

LEARNING OBJECT RECOGNITION AND
OBJECT CLASS SEGMENTATION WITH
DEEP NEURAL NETWORKS ON GPU

DISSERTATION

zur Erlangung des Doktorgrades (Dr. rer. nat.)
der Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

SCHULZ, HANNES
aus Leipzig

Bonn, 2015

Angefertigt mit Genehmigung der Mathematisch-
Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-
Wilhelms-Universität Bonn

Erster Gutachter

Prof. Dr. Sven Behnke

Zweiter Gutachter

Prof. Dr.-Ing. Christian Bauckhage

Tag der Promotion

9. September 2016

Erscheinungsjahr

2017

ZUSAMMENFASSUNG

Allgegenwärtige Kameras und preiswerter Internetspeicher erzeugen einen großen Bedarf an Algorithmen für maschinelles Sehen. Die vorliegende Dissertation adressiert zwei Teilbereiche dieses Forschungsfeldes: Erkennung von Objekten und Objektklassensegmentierung. Der methodische Schwerpunkt liegt auf dem Lernen von tiefen Modellen („Deep Learning“). Diese haben in den vergangenen Jahren einen enormen Einfluss auf maschinelles Lernen allgemein und speziell maschinelles Sehen gewonnen. Dabei behandeln wir drei Themenfelder.

Der erste Teil der Arbeit beschreibt ein GPU-basiertes Softwaresystem für Deep Learning. Dessen hierarchische Struktur erlaubt schnelle GPU-Berechnungen, einfache Spezifikation komplexer Modelle und interaktive Modellanalyse. Damit liefert es das Fundament für die folgenden Kapitel. Teile des Systems finden Verwendung in einer Echtzeit-GPU-Bibliothek für Random Forests, die wir ebenfalls vorstellen und evaluieren.

Der zweite Teil der Arbeit beleuchtet Greedy-Lernalgorithmen für halbüberwachtes Lernen. Hier werden hierarchische Modelle schrittweise aus Modulen wie Autokodierern oder restricted Boltzmann Machines (RBMs) aufgebaut. Wir verbessern die Repräsentationsfähigkeiten von RBMs auf Bildern durch Einführung lokaler und lateraler Verknüpfungen und liefern empirische Erkenntnisse zur Bewertung von RBM-Lernalgorithmen. Wir zeigen zudem, dass die in Autokodierern verwendeten einschichtigen Kodierer komplexe Zusammenhänge ihrer Eingaben nicht erkennen können und schlagen stattdessen einen hybriden Kodierer vor, der sowohl komplexe Zusammenhänge erkennen, als auch weiterhin einfache Zusammenhänge einfach repräsentieren kann.

Im dritten Teil der Arbeit stellen wir neue neuronale Netzarchitekturen und Trainingsmethoden für die Objektklassensegmentierung vor. Wir zeigen, dass neuronale Netze mit überwachtem Vortrainieren, wiederverwendeten Ausgaben und Histogrammen Orientierter Gradienten (HOG) als Eingabe den aktuellen Stand

der Technik auf mehreren RGB-Datenmengen erreichen können. Anschließend erweitern wir unsere Methoden in zwei Dimensionen, sodass sie mit Tiefendaten (RGB-D) und Videos verarbeiten können. Dazu führen wir zunächst Tiefennormalisierung für Objektklassensegmentierung ein um die Skala zu fixieren, und erlauben expliziten Zugriff auf die Höhe in einem Bildausschnitt. Schließlich stellen wir ein rekurrentes konvolutionales neuronales Netz vor, das einen großen räumlichen Kontext einbezieht, hochaufgelöste Ausgaben produziert und Videosequenzen verarbeiten kann. Dadurch verbessert sich die Bildsegmentierung relativ zu vergleichbaren Methoden, etwa auf der Basis von Random Forests oder CRFs. Wir zeigen dann, dass pixelbasierte Ausgaben in neuronalen Netzen auch benutzt werden können um die Position von Objekten zu detektieren. Dazu kombinieren wir Techniken des strukturierten Lernens mit Konvolutionsnetzen. Schließlich schlagen wir eine objektzentrierte Einfärbungsmethode vor, die es ermöglicht auf RGB-Bildern trainierte neuronale Netze auf RGB-D-Bildern einzusetzen. Dieser Transferlernansatz erlaubt es uns auch mit stark reduzierten Trainingsmengen noch bessere Ergebnisse beim Schätzen von Objektklassen, -instanzen und -orientierungen zu erzielen.

Wir werten die von uns vorgeschlagenen Methoden auf den öffentlich zugänglichen MNIST, MSRC, INRIA Graz-02, NYU-Depth, Pascal VOC, und Washington RGB-D Objects Datenmengen aus.

ABSTRACT

As cameras are becoming ubiquitous and internet storage abundant, the need for computers to understand images is growing rapidly. This thesis is concerned with two computer vision tasks, recognizing objects and their location, and segmenting images according to object classes. We focus on deep learning approaches, which in recent years had a tremendous influence on machine learning in general and computer vision in particular. The thesis presents our research into deep learning models and algorithms. It is divided into three parts.

The first part describes our GPU deep learning framework. Its hierarchical structure allows transparent use of GPU, facilitates specification of complex models, model inspection, and constitutes the implementation basis of the later chapters. Components of this framework were used in a real-time GPU library for random forests, which we present and evaluate.

In the second part, we investigate greedy learning techniques for semi-supervised object recognition. We improve the feature learning capabilities of restricted Boltzmann machines (RBM) with lateral interactions and auto-encoders with additional hidden layers, and offer empirical insight into the evaluation of RBM learning algorithms.

The third part of this thesis focuses on object class segmentation. Here, we incrementally introduce novel neural network models and training algorithms, successively improving the state of the art on multiple datasets. Our novel methods include supervised pre-training, histogram of oriented gradient DNN inputs, depth normalization and recurrence. All contribute towards improving segmentation performance beyond what is possible with competitive baseline methods. We further demonstrate that pixel-wise labeling combined with a structured loss function can be utilized to localize objects. Finally, we show how transfer learning in combination with object-centered depth colorization can be used to identify objects.

We evaluate our proposed methods on the publicly available MNIST, MSRC, INRIA Graz-02, NYU-Depth, Pascal VOC, and Washington RGB-D Objects datasets.

ACKNOWLEDGMENTS

I would like to thank my advisor Sven Behnke for the opportunity to pursue my studies in his department, his support, encouragement, and fertile discussions. I would also like to thank my second advisor, Christian Bauckhage, for agreeing to review my thesis.

During my time at the University Bonn, I was fortunate to work on a large range of topics, from Boltzmann machines to object detections, but also from soccer field localization to plant root segmentation. Whatever it was, it was a pleasure to discuss and develop with my co-workers in the autonomous systems group. I am particularly grateful to Dr. Andreas Müller (NY University) for his constructive advice.

This thesis would not have been possible without the hard work of the University Bonn students, especially Benedikt Waldvogel, Max Schwarz, Niko Höft, and Mircea Pavel.

I would also like to say thanks to Tapani Raiko (Aalto University) and Kyunghyun Cho (NY University) for hosting me in Helsinki during my all too brief stay.

Last but not least, I would like to thank Sarah for her unrelenting support and patience during this strenuous time.

Contents

1	INTRODUCTION	1
1.1	Key Contributions	4
1.2	Publications	5
2	BACKGROUND ON DEEP LEARNING	7
2.1	The Argument for Many-Layered Neural Networks	7
2.2	Notation	9
2.3	Difficulties in Learning Deep Architectures	11
2.4	Greedy Layer-wise Training with RBMs	12
2.4.1	Greedy Training with Auto-Encoders	14
2.5	Regularization in Unsupervised Learning	15
2.5.1	Limiting Representation Size	15
2.5.2	Enforcing Representation Sparsity	15
2.5.3	Infinite Training Data	16
2.6	Convolutional Neural Networks	16
2.6.1	Sparse Parameters	17
2.6.2	Weight Sharing	18
2.6.3	Pooling	19
2.6.4	Margin Handling	19
2.7	Cross-Dataset Transfer Learning	19
2.8	Conclusion	21
3	GPU-BASED MACHINE LEARNING	23
3.1	Background on the Backpropagation Algorithm	23
3.1.1	Forward Pass	24
3.1.2	Backward Pass	24
3.1.3	Weight Update	25
3.2	GPU Considerations	26
3.3	CUV: N-Dimensional Arrays in CUDA	27

3.4	Generic Model Optimization with CUVNET	31
3.4.1	Requirements of Deep Learning Algorithms	31
3.4.2	Fast GPU Implementation	31
3.4.3	User Interface	32
3.4.4	Complex Models	33
3.4.5	Differentiation Graph	35
3.4.6	Special Copy-on-write Semantics	35
3.4.7	Monitoring and Visualization	37
3.4.8	Testing	38
3.4.9	Other Major Features	39
3.5	Comparison with other Neural Network Software	39
3.6	Acknowledgments	40
3.7	Applications	41
4	CURFIL: FAST RANDOM FORESTS IN CUDA	43
4.1	Related Work	45
4.2	Random Forests	46
4.3	Visual Features for Node Tests	47
4.4	CURFIL Software Package	49
4.4.1	GPU Kernels	53
4.4.2	Global Memory Limitations	56
4.4.3	Extensions	57
4.5	Experimental Results	58
4.5.1	Datasets	58
4.5.2	Training and Prediction Time	59
4.5.3	Classification Accuracy	60
4.5.4	Incorporating Novel Features	62
4.5.5	Random Forest Parameters	63
4.6	Conclusion	63
5	UNSUPERVISED METHODS FOR IMAGE CATEGORIZATION	65
5.1	Exploiting Local Structure in Boltzmann Machines	65
5.1.1	Background on Boltzmann Machines	66
5.1.2	Local Impact Semi-Restricted Boltzmann Machines ⁺	67
5.1.3	Related Work	69
5.1.4	Experimental Results	70
5.1.5	Conclusions	73

5.2	Investigating Convergence of RBM Learning	73
5.2.1	Background on Restricted Boltzmann Machines	76
5.2.2	Experimental Setup	79
5.2.3	Results	83
5.2.4	Conclusions	85
5.3	Two-Layer Encodings for Semi-supervised Learning	86
5.3.1	Related Work	88
5.3.2	Background	89
5.3.3	Where Pre-Training of One-Layer Encoders Fails	93
5.3.4	Two-Layer Encoders and Contractive Regularization	95
5.3.5	Two-Layer Encoders and Shortcut Connections	96
5.3.6	Experiments	96
5.3.7	Discussion	105
5.3.8	Conclusions	106
6	LEARNING OBJECT-CLASS SEGMENTATION	109
6.1	Learning Object Class Segmentation with CNN	110
6.1.1	Methods	110
6.1.2	Results	114
6.1.3	Related Work	117
6.1.4	Conclusion	119
6.2	Encoding Depth Information for CNN	119
6.2.1	Methods	120
6.2.2	Experiments	121
6.2.3	Related Work	122
6.2.4	Conclusion	124
6.3	Depth and Height Aware Semantic Perception	125
6.3.1	Related Work	127
6.3.2	Methods	128
6.3.3	Experiments	130
6.3.4	Conclusion	133
6.4	Recurrent Networks for Depth Perception	133
6.4.1	Model Description	135
6.4.2	Related Work	139
6.4.3	Experiments	141
6.4.4	Conclusion	151
6.5	Chapter Summary	152

- 7 STRUCTURED PREDICTION FOR OBJECT DETECTION 153
 - 7.1 Structured Prediction for Object Detection 154
 - 7.2 Experiments 158
 - 7.3 Related Work 160
 - 7.4 Conclusion 162

- 8 TRANSFER LEARNING FOR RGB-D OBJECT RECOGNITION 165
 - 8.1 Related Work 167
 - 8.2 CNN Feature Extraction Pipeline 169
 - 8.2.1 RGB Image Preprocessing 169
 - 8.2.2 Depth Image Preprocessing 170
 - 8.2.3 Image Feature Extraction 174
 - 8.3 Learning Method 175
 - 8.3.1 Object Classification 175
 - 8.3.2 Object Pose Estimation 175
 - 8.4 Evaluation 176
 - 8.4.1 Evaluation Protocol 176
 - 8.4.2 Results 177
 - 8.5 Conclusion 183

- 9 CONCLUSION 185
 - 9.1 Future Directions 187

- A ACRONYMS AND SYMBOLS 189

- BIBLIOGRAPHY 195

INTRODUCTION

How can a machine see? This question becomes increasingly important as ubiquitous smartphones, wearables, self-driving cars and other autonomous robots are equipped with cameras. While interpreting images and videos is an easy feat for humans, modern computer systems still struggle with this task. One reason for this struggle may be the output specification — what does it mean to “interpret” an image? —, but even for clearly defined subtasks such as object recognition, many questions are unsolved.

An image taken by a consumer camera is a two-dimensional array of pixels between 0 and 255. These values can change dramatically depending on, e.g., environmental conditions, depicted objects and instances, and their pose. Due to this complexity, interpretation of images usually relies heavily on models about statistical properties of images in general and the specific entities required for a task.

Specifying models manually is tedious and difficult. Instead, we can learn models from datasets which contain labeled information. The research field of DNNs has gained much attention in the past years. Recently, its initial success on toy problems carried over to challenges of the computer vision community. Today, major parts of this community focus on DNN-based models. This achievement is largely attributed to the availability of large amounts of training data, high-throughput GPU processing power, and a large, accumulated body of insights into how to build, initialize, and train DNN models.

DNN models are functions on images, which transform the image in multiple steps, using a limited set of operations. This structure is inspired by the processing of visual inputs in the visual cortex of primates. Every operation in the chain has parameters, which must be either set by the model designer (hyper-parameters), or learned from data.

Specifying a deep model’s structure, providing adequate pre-processing and output encoding, and learning its millions of parameters using various gradient methods, learning stages, and transfer learning is a still largely unexplored playground. This thesis examines a few of its corners.

In this thesis, we cover three main topics. After an introduction to deep learning in general (Chapter 2), we are first concerned with the technical challenges of model learning. It requires time-consuming experiments on large datasets, which we (and most other researchers) delegate to a graphics processing unit (GPU). We start by explicating technical requirements in Sections 3.1 and 3.2 and proceed to describe three libraries which we created for this thesis. CUV (Section 3.3) provides a high-level interface to multi-dimensional arrays on GPU. This library is the workhorse of CUVNET (Section 3.4), which in turn offers automatic differentiation, multiple abstractions over operations to simplify specification of complex models, and a machine learning eco-system that allows to determine hyper-parameters and perform experiments on a variety of tasks. CUV is also used in another library, CUDA random forests for image labeling (CURFIL, Chapter 4). Random forests are another computer vision technique, which is exceptionally fast and yields good results. Our implementation is real-time capable even on mobile GPUs. We return to the image labeling task in Chapter 6 using DNNs.

The second main topic of this thesis is semi-supervised learning for image categorization. It comprises a number of techniques where the model is initialized with task-unspecific properties from a dataset, and afterwards fine-tuned using labeled data. To this end, features are learned sequentially in a hierarchy of learning blocks, restricted Boltzmann machines (RBMs) or auto-encoders. We explore extensions of RBMs, generative models with hidden variables, for modeling images in Section 5.1. While our extensions produce good results, we show in Section 5.2 that empirically, learning an RBM is a matter of luck, since the exact gradient is too expensive to compute, and known partition function approximations do not perform well even on small problems. In Section 5.3, we turn to auto-encoders, which can be learned directly with gradient descent on the objective function. Here,

we show that the commonly used shallow auto-encoders cannot learn certain classes of functions, construct a dataset where common auto-encoders fail, and provide an improved approach using two-layer contractive encoders with shortcut connections. The two-layer structure allows for detection of complex features, while shortcuts allow simple features to be represented as in single-layer encoders.

The third main topic of the thesis is the development of models for object class segmentation. In Section 6.1, we introduce a model and a training procedure for object class segmentation with DNNs. HOG descriptors are an established tool in computer vision, which we adapt for DNNs. To improve accuracy, we also propose to diverge from the linear structure of common feed-forward neural networks, by introducing multi-scale inputs and reusing of pre-trained outputs. We then supplement our approach to work with three-dimensional data provided by an RGB-D sensor in Section 6.2. We show that with histogram of oriented depth (HOD) descriptors and depth normalization, our CNN outperforms the state-of-the-art convolutional network as well as random forest and CRF-based approaches. Last but not least in Section 6.4, we extend our network in the time domain using a recurrent convolutional neural network (RNN) for video processing. This network is able to retain a state over time, track and measure movement, retain uncertainty, and produces better segmentations than all previous methods.

Another class of problems that can be cast into pixel-wise predictions is object localization. In Chapter 7, we suggest to combine deep learning techniques with structured learning. Structured learning allows to produce outputs in a space which is not enumerable, but structured. Here, a point in this space is an arbitrary number of object bounding boxes in an image. We propose a neural network which produces output maps, from which bounding boxes of objects in the image can be inferred efficiently. This inference procedure is used as part of the neural network loss.

In Chapter 8, we return to the object classification task and combine it with depth perception. We show that object-centered colorization schemes of depth allow transfer learning from net-

works pre-trained on RGB image classification tasks. Transfer-learning allows us to produce superior results even with small datasets, which is essential for robotics applications. We apply this technique to RGB-D object classification and pose-estimation. Chapter 9 concludes the thesis.

1.1 KEY CONTRIBUTIONS

This thesis contains the following contributions:

- Providing open-source GPU-based software frameworks for deep learning and random forests.
- Introducing methods for learning laterally and locally connected RBMs, which are better suited for images exhibiting weak long-range dependencies between pixels.
- Demonstrating empirically that RBM training success is not predictable. We use annealed importance sampling (AIS), the standard method for model selection on unnormalized multimodal probability distributions, and find that it is not reliable even for very simple tasks.
- Showing that non-linear relations of variables can be a problem in semi-supervised deep learning due to the fact that some stable features cannot be detected. To recover, we propose a two-layer auto-encoder with shortcuts, which combines simplicity of single-layer encoders with the representational power of two-layer encoders.
- Introducing a DNN model for object class segmentation. Our approach is fast and achieves very good performance on a number of benchmark datasets.
- Extending the object class segmentation model to the depth modality, using histograms of oriented depth, depth normalization and height-awareness.
- Extending object class segmentation models for images to video processing, using recurrent convolutional neural net-

works. While slower and more difficult to train, prediction accuracy of these models increases strongly over non-recurrent models.

- Combining structured prediction with deep learning for object detection. This approach allows optimization of the PASCAL VOC measure of detection quality directly during learning.
- Demonstrating that transfer learning can be a powerful tool for robotics, where datasets are typically small. We use depth-coloring and pre-trained CNN to recognize objects and their pose in RGB-D images in small datasets.

1.2 PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

- Schulz, H., A. Müller, and S. Behnke (2010b). “Investigating Convergence of Restricted Boltzmann Machine Learning.” In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.
- Schulz, H., A. Müller, and S. Behnke (2011). “Exploiting Local Structure in Boltzmann Machines.” In: *Neurocomputing* 74:9. Supersedes Schulz et al. (2010a).
- Schulz, H. and S. Behnke (2012a). “Deep Learning: Layer-wise Learning of Feature Hierarchies.” In: *Künstliche Intelligenz* 26.4: *Neural Learning Paradigms*.
- Schulz, H. and S. Behnke (2012b). “Learning Object-Class Segmentation with Convolutional Neural Networks.” In: *European Conference on Neural Networks (ESANN)*.
- Höft, N., H. Schulz, and S. Behnke (2014). “Fast Semantic Segmentation of RGB-D Scenes with GPU-Accelerated Deep Neural Networks.” In: *German Conference on Artificial Intelligence (KI)*. Lecture Notes in Computer Science (LNCS) 8736. Springer.

- Schulz, H. and S. Behnke (2014). “Structured Prediction for Object Detection in Deep Neural Networks.” In: *International Conference on Artificial Neural Networks (ICANN)*.
- Schulz, H., K. Cho, T. Raiko, and S. Behnke (2014). “Two-Layer Contractive Encodings for Learning Stable Nonlinear Features.” In: *Neural Networks: Deep Learning of Representations*. Ed. by Y. Bengio and H. Lee. Supersedes Schulz and Behnke (2012c) and Schulz et al. (2013).
- Schwarz, M., H. Schulz, and S. Behnke (2014). “RGB-D Object Recognition and Pose Estimation based on Pre-trained Convolutional Neural Network Features.” In: *International Conference on Robotics and Automation (ICRA)*.
- Pavel, M. S., H. Schulz, and S. Behnke (2015). “Recurrent Convolutional Neural Networks for Object-Class Segmentation of RGB-D Video.” In: *International Joint Conference on Neural Networks (IJCNN)*.
- Schulz, H., N. Höft, and S. Behnke (2015a). “Depth and Height Aware Semantic RGB-D Perception with Convolutional Neural Networks.” In: *European Conference on Neural Networks (ESANN)*.
- Schulz, H., B. Waldvogel, R. Sheikh, and S. Behnke (2016). “CURFIL: A GPU library for Image Labeling with Random Forests.” In: *Computer Vision, Imaging and Computer Graphics. Theory and Application*. Communications in Computer and Information Science. Springer. Supersedes Schulz et al. (2015b).

BACKGROUND ON DEEP LEARNING

2.1 THE ARGUMENT FOR MANY-LAYERED NEURAL NETWORKS

Supervised learning tasks, such as assigning a class label to images, are given as a set of example input-output pairs where the output must be predicted. Different learning architectures, such as support vector machines, neural networks, decision trees, and memory-based methods (e. g. k nearest neighbors) can be used to approximate the desired classification function not only for the given examples, but also for unseen test images.

Frequently, classification is not done directly on the raw pixel input, but an intermediate representation—a vector of *features*—is extracted first, which is then classified, resulting in a two-stage computation. This approach performs very well if the features represent the essential information needed for classification. Obviously, feature extraction is not free of parameters and depends on the type of data and the task. For example, we might expect a different set of features is useful for detecting cars in images than for detecting people. Perhaps surprisingly, however, some types of features, like localized edges for natural images, can be adopted to a range of tasks (Bottou, 2014). These features seem to be generic representations of the input signal. Finding these generic features is difficult though—often feature extractors are hand-crafted and selected by testing them on many similar learning problems.

Other methods, such as feed-forward neural networks with a single hidden layer, learn features from the training set by optimizing the parameters of feature extraction and the classifier simultaneously. While such networks can in principle represent almost any function (Cybenko, 1989), the number of required feature detectors in the hidden layer can be exponential in the number of inputs. This property is a generalization of the circuit complexity result that any Boolean function can be represented

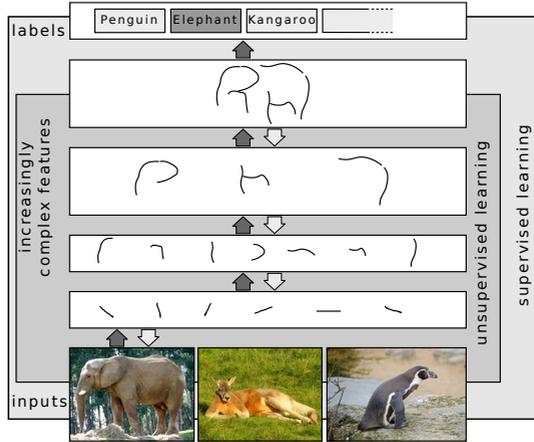


Figure 2.1: Schematic overview of layer-wise learning of feature hierarchies. Increasingly complex features are determined from input using unsupervised learning. The features can be used for supervised task learning.

by two layers of conjunction and disjunction of inputs (Shannon, 1949). Often however, Boolean functions can be represented more space-efficiently by multi-stage binary decision diagrams that are less wide and hence need less logic units. The gain in efficiency is made possible by reusing the results of lower-level circuits at higher levels. Applying this finding to the feed-forward neural network context, we can save space—and time in sequential processing—by combining lower-level features to more abstract features when we allow multiple hidden layers and create a *feature hierarchy*.

Hierarchical neural networks for object categorization in images have a long history. They are motivated by the hierarchical structure of the ventral stream of the visual cortex, where a sequence of retinotopic representations is computed that represents the field-of-view with decreasing spatial resolution, but increasing feature complexity. This network structure reflects the typical hierarchical structure of images, where edges can be grouped to parts, which form objects (Fig. 2.1).

One of the earliest hierarchical neural networks for object recognition was the Neocognitron proposed by Fukushima (1980). In a sequence of feature extracting and pooling layers—which create invariance to local shifts—the network was able to recognize handwritten digits and letters, even if they had been deformed. Other prominent hierarchical neural networks for object recognition include LeNet by LeCun et al. (1989), the HMAX network by Riesenhuber and Poggio (1999), and the Neural Abstraction Pyramid by Behnke (2003b). Hierarchical features are also learned in the hierarchy of parts proposed by Fidler and Leonardis (2007). Finally, hierarchy is a key feature in state-of-the-art architectures for object recognition. While shallow, extremely wide architectures perform very well (Coates et al., 2010), the best-performing classifiers for standard datasets are currently very deep with ten (Cireşan et al., 2012b) or even twenty (Szegedy et al., 2015) levels of features.

Despite the above examples, difficulties in learning the feature hierarchies often prevented better performance of deep architectures, as compared to the convex optimization of shallow models like the Support Vector Machine. To overcome these difficulties, Hinton et al.—who coined the term deep learning—proposed to initialize supervised training of deep networks with a feature hierarchy that was learned in an unsupervised way, layer by layer from the data (Hinton and Salakhutdinov, 2006; Hinton et al., 2006). The impressive performance of this method—together with massively parallel computation by GPUs—triggered a revival of neural networks research.

2.2 NOTATION

To ease the following discussion, let us first define the terms. A deep architecture (as shown in Fig. 2.2) with depth L is a function $\mathbf{h}^{(L)}$ with

$$\mathbf{h}^{(l)}(\mathbf{x}) = \sigma \left(W^{(l)} \mathbf{h}^{(l-1)}(\mathbf{x}) + \mathbf{b}^{(l)} \right), \quad \mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}. \quad (2.1)$$

Here, $\mathbf{x} \in \mathbb{R}^N$ is the input vector, $\sigma(\cdot)$ is an elementwise sigmoid function such as $\sigma_i(\mathbf{z}) = (1 + \exp(-z_i))^{-1}$. The weight matrices

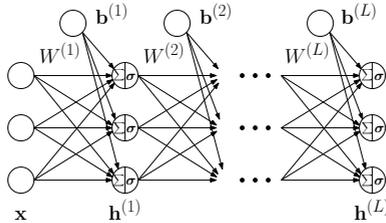


Figure 2.2: Visualization of a deep architecture

$W^{(l)} \in \mathbb{R}^{N_l \times N_{l-1}}$ and the biases $\mathbf{b}^{(l)} \in \mathbb{R}^{N_l}$ constitute the layer parameters $\theta^{(l)}$. Intermediate $\mathbf{h}^{(l)}(\mathbf{x}) \in \mathbb{R}^{N_l}$ are referred to as hidden layers. When unambiguous, we omit the dependency on \mathbf{x} and write $\mathbf{h}^{(l)}$. When discussing properties of a specific layer, we will also skip the layer index (l) for simplicity. While Eq. (2.1) is based on dot products of input and weight vectors, it is also possible to rely on differences between the two vectors instead and set $\sigma(\cdot)$ to a radial basis function.

Our features correspond to the rows of $W^{(l)}$ and can be determined by learning. We first formalize the task using a loss function which is minimal when the task is solved. *Learning* is then to find parameters such that the loss function is minimal on some training data \mathcal{D} . For example, we might choose the mean square loss

$$\ell_{\text{MSE}}(\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(L)}, \mathcal{D}) = \sum_{d=1}^D \sum_{n=1}^N \left(h_n^{(L)}(\mathbf{x}^d) - x_n^d \right)^2 \quad (2.2)$$

for an i.i.d. dataset $\mathcal{D} = \{\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^D\}$ with Gaussian noise model. This is an unsupervised task where the input is reconstructed from the features. If for some $l < L$ we have $N_l < N_0$, this requires learning to compress and decompress the input¹. Supervised tasks provide some desired output or label in addition to the input data. If we assume binary labels $\mathbf{y} \in \{0, 1\}^M$ with $\sum_{m=1}^M y_m = 1$, our dataset is $\mathcal{D} = \{(\mathbf{x}^1, \mathbf{y}^1), (\mathbf{x}^2, \mathbf{y}^2), \dots, (\mathbf{x}^D, \mathbf{y}^D)\}$,

¹ In fact, an auto-encoder without non-linearity is learning a projection to the subspace spanned by the first N_l principal components of the data (Bourlard and Kamp, 1988), which is optimal w.r.t. the squared reconstruction loss.

and we learn a classification task by minimizing the cross-entropy loss

$$\ell_{\text{CE}}(\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(L)}, \mathcal{D}) = - \sum_{d=1}^D \sum_{m=1}^M \left[y_m^d \cdot \log \frac{\exp(h_m^{(L)}(\mathbf{x}^d))}{\sum_{m'=1}^M \exp(h_{m'}^{(L)}(\mathbf{x}^d))} \right]. \quad (2.3)$$

In networks with at least one non-linear hidden layer, both losses are non-convex in the parameters. They are typically minimized by gradient descent, where the gradients are efficiently (linearly in the number of parameters) computed with the backpropagation algorithm (Section 3.1).

2.3 DIFFICULTIES IN LEARNING DEEP ARCHITECTURES

The principles of learning architectures with many levels have been widely known since the popularization of the backpropagation algorithm for multi-layer perceptron (MLP) by Rumelhart et al. (1986). In practice, however, using more than one hidden layer was neither common nor successful. The challenges of DNN learning include

1. increasing depth increases the probability of training stopping in a poor local minimum of the non-convex loss (Erhan et al., 2009),
2. training the lower layers (close to the input) is more difficult than training the higher layers (ibid.). A reason might be vanishing gradients, comparable to gradient propagation problems in recurrent neural networks (Hochreiter et al., 2001). Finally,
3. learning DNNs requires more training data than other methods, since many parameters have to be estimated.

Approaches to deal with these problems have been changing a lot in recent years, which is also reflected by the later chapters of this thesis. The main focus has been on finding good priors—

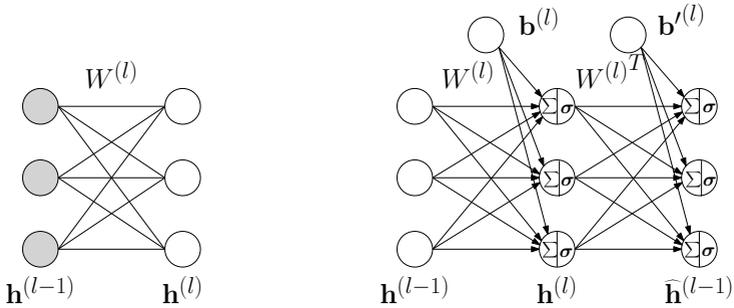


Figure 2.3: Building blocks for greedy pre-training. (a) A restricted Boltzmann machine (RBM) is an undirected graphical model where variables are depicted as circles. Gray circles signify *observed* variables. (b) Auto-encoder network, reconstructing $\mathbf{h}^{(l-1)}$ with $\hat{\mathbf{h}}^{(l-1)}$.

especially for the parameters in the lower layers of the network—and using more training data. More precisely,

1. *greedy layer-wise training* uses multiple unsupervised learning problems starting at the input, to find a good initialization for the supervised problem,
2. *regularization* attempts to prevent overfitting on small datasets,
3. the *network structure* can be adjusted reduce the number of parameters and comply with dataset properties, e.g. invariance to location in images with the help of convolutional neural network (CNN),
4. features learned from other (e.g. larger) datasets can be *transferred* to novel datasets, and

Since these strategies will play a central role in later chapters, we discuss them in more detail in the following sections.

2.4 GREEDY LAYER-WISE TRAINING WITH RBMS

In their influential work on data reduction with neural networks, Hinton and Salakhutdinov (2006) introduced a first solution to

the problems stated in Section 2.3. Before minimizing the loss of the deep network with L levels, they optimized a sequence of $L - 1$ single-layer problems using restricted Boltzmann machines (RBM). An RBM is a graphical model displayed in Fig. 2.3 (a) that represents the log-linear probability distribution

$$\begin{aligned} p(\mathbf{h}^{(l-1)}, \mathbf{h}^{(l)}) \\ = \frac{1}{Z} \cdot \exp \left(\mathbf{b}^{(l-1)T} \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)T} \mathbf{h}^{(l)} + \mathbf{h}^{(l)T} W^{(l)} \mathbf{h}^{(l-1)} \right), \end{aligned} \quad (2.4)$$

where Z is the *partition function* which ensures that

$$\int \int p(\cdot, \cdot) d\mathbf{h}^{(l-1)} d\mathbf{h}^{(l)} = 1.$$

Here, $\mathbf{h}^{(l)}$ and $\mathbf{h}^{(l-1)}$ denote vectors of binary random variables. The parameters are chosen such that, when we marginalize out $\mathbf{h}^{(l)}$, they minimize the negative log-likelihood of the data distribution

$$\ell_{DD}(\theta^{(l)}, \mathcal{D}) = - \sum_{d=1}^D \log \sum_{\mathbf{h}^{(l)}} p \left(\mathbf{h}^{(l-1)}(\mathbf{x}^d), \mathbf{h}^{(l)} \right). \quad (2.5)$$

At first glance, we note that this loss is very different from Eqs. (2.2) and (2.3). Determining the gradient of Eq. (2.5) analytically is usually unfeasible since calculating the partition function Z in $p(\cdot, \cdot)$ scales exponentially with $\min(N_l, N_{l-1})$. Instead, approximations such as contrastive divergence (Hinton, 2002; Tieleman, 2008) are used. Due to the factorization of the graphical model, however, the expected conditional probability of $\mathbf{h}^{(l)}$ given $\mathbf{h}^{(l-1)}$ can be calculated analytically as

$$p \left(\mathbf{h}^{(l)} \mid \mathbf{h}^{(l-1)} \right) = \sigma \left(W^{(l)} \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)} \right) \quad (2.6)$$

— which closely resembles Eq. (2.1)! After minimizing $\ell_{DD}(\theta^{(l)}, \mathcal{D})$, Hinton and Salakhutdinov transform \mathcal{D} using Eq. (2.6) and iterate the process for $\theta^{(l+1)}$. Once all parameters of a feature hierarchy have been *pre-trained* in this way, they are *fine-tuned* with the original objective (e. g. Eq. (2.3)). While the precise definitions of pre-training and fine-tuning loss vary, this general approach is prevalent in the deep learning literature.

2.4.1 Greedy Training with Auto-Encoders

Arguably, auto-encoders (Bengio et al., 2006) are more prevalent for pre-training than RBM, since their gradient can be calculated exactly². An auto-encoder (Fig. 2.3 b) is a function

$$\hat{\mathbf{h}}^{(l-1)} = W^{(l)T} \mathbf{h}^{(l)}(\mathbf{x}) + \mathbf{b}'^{(l-1)}. \quad (2.7)$$

In its hidden layer $\mathbf{h}^{(l)}$, it creates a feature representation (encoding) of its input $\mathbf{h}^{(l-1)}$. The encoding is used for two purposes. Firstly, we optimize ℓ_{MSE} to reconstruct $\mathbf{h}^{(l-1)}$ from the features. Secondly, similar to RBMs, the features are used as input for $\mathbf{h}^{(l+1)}$, where the next-level auto-encoder is trained in the same fashion. The close connection between RBMs and auto-encoders has been investigated by Vincent (2011).

Why can we get away with changing the loss function between pre-training and fine-tuning seemingly at will? There are at least two reasons which have been identified:

1. The discussed pre-training methods identify generic features of the input, which resemble largely independent constituents. Higher-level features detect common co-occurrence patterns of lower level features. Intuitively, such features are likely to play a role in many objectives, termed the *structure assumption* in Weston et al. (2008).
2. Pre-training can be seen as a *regularization* of fine-tuning. It moves the weights to a region in parameter space that has better generalization properties, and can therefore be seen as a form of semi-supervised learning (Erhan et al., 2010a).

Additionally, by training only one layer at a time, we solve simpler problems with fewer local minima and cleaner gradients (as discussed in Section 2.3)—and postpone dealing with the complete, hard problem to the fine-tuning phase.

Other local learning methods that have been applied to learn feature hierarchies include competitive learning (Fukushima

² without non-linearity, their object is even convex (Baldi and Hornik, 1989)

1980 and Behnke 1999), slow feature analysis (Wiskott and Sejnowski, 2002), non-negative matrix factorization (Behnke, 2003a), and deconvolutions (Zeiler et al., 2011).

2.5 REGULARIZATION IN UNSUPERVISED LEARNING

For a successful application of deep learning, additional precautions have to be taken. An important requirement is that we should not learn trivial features of the input. Commonly, data is normalized by centering and sometimes (ZCA) whitening (Hyvärinen and Oja, 2000; Welling and Hinton, 2002). Still, if the number of features is large enough, the identity function can be learned, which does not yield new insights to be used in higher layers.

2.5.1 *Limiting Representation Size*

One way to enforce learning of novel features in auto-encoders is to keep the number of features small with respect to the number of inputs, such that for all $l < L : N_{l+1} < N_l$. Intuitively, we learn to represent the input using fewer bits and minimize the information loss.

2.5.2 *Enforcing Representation Sparsity*

Sometimes it is useful to learn highly overcomplete ($N_l \gg N_{l-1}$) feature hierarchies, e. g. for decomposing the signal for use in linear classifiers (Boureau et al., 2010). In this case, we can amend the auto-encoder loss function by approximately minimizing the number of non-zero entries of the hidden representation, resulting in

$$\ell_{MSE+S}(\theta^{(l)}, \mathcal{D}) = \ell_{MSE}(\theta^{(l)}, \mathcal{D}) + \lambda \sum_{d=1}^D \left\| \mathbf{h}^{(l)}(\mathbf{x}^d) \right\|_1.$$

Optimizing this objective is related to sparse coding (Kavukcuoglu et al., 2010).

2.5.3 *Infinite Training Data*

Large datasets allow better generalization to unseen test data from the same distribution. We can artificially introduce new data to profit from this effect, even if the generated data is not strictly i.i.d. In auto-encoders, this is achieved by adding noise to the input, and reconstructing the original, noise-free data (Vincent et al., 2010). This also requires to learn non-trivial features, since any single input is likely to be corrupted by noise. The learning algorithms for RBMs have built-in random sampling, which has a similar effect (Vincent, 2011).

We can also inject noise by setting random units in the hidden layers of a DNN to zero. This technique, especially when combined with the rectifying linear unit (ReLU) nonlinearity, is known as *dropout* (Srivastava et al., 2014). Since dropout randomly disables units in a DNN, it effectively samples one of exponentially many network structures every time an example is presented. During prediction, no noise is applied and the network implicitly “averages” over the possible network structures, which can be interpreted as a form of bagging.

Another approach to increase training data size is to use so-called adversarial examples (Szegedy et al., 2014; Goodfellow et al., 2014). These examples are created by determining the minimal change necessary to misclassify an example from the training data. Even examples without relation to the training data, but are classified with high confidence might be useful (Nguyen A, 2015).

2.6 CONVOLUTIONAL NEURAL NETWORKS

If inputs are natural images, we can further exploit their two-dimensional structure to regularize learning. Pixels have much stronger correlation with their immediate neighbors than with far-away ones (Huang and Mumford, 1999). In natural images, we can assume that these correlations are roughly similar for all image locations. Convolutional neural network (CNN) have been developed to exploit these statistics by efficiently leveraging spar-

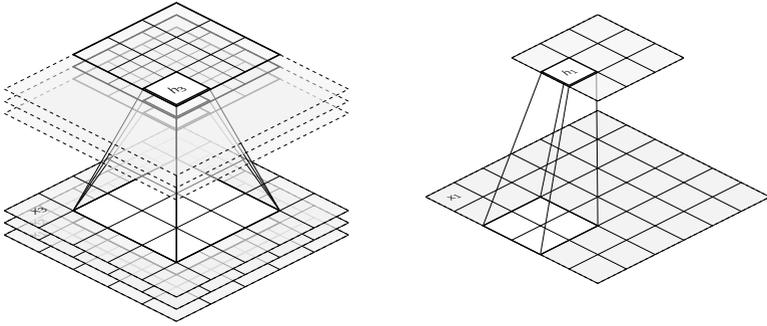


Figure 2.4: *Left*: Convolution layer of a convolutional neural network. A hidden unit h_m in an output map m has access to a rectangular region (receptive field) in a number of input maps x_s . The location of the receptive field corresponds to the location of the hidden unit in the output map. Size of the receptive field and number of input maps determine the number of weights. *Right*: Pooling layer of a convolutional neural network. A hidden unit has access to a receptive field in a single input map, and computes The number of output maps is the same as the number of input maps. Typically, only averages or maxima are computed and no parameters need to be learned.

sity in the weight matrices and enforced weight sharing, as well as local invariance to deformations.

2.6.1 Sparse Parameters

To reflect the strong local correlations, image features learned e.g. by auto-encoders are localized, i. e. mostly zero. These zeros need not be learned, instead, we can use local weight matrices which—for each feature—allow a set of non-zero weights only in a small image region. The responses of localized feature detectors can be arranged in the same coordinates as the original image. Thus, every layer of the CNN contains a set of *maps*, one for each learned feature. Formally, instead of full connectivity between layers

$$h_i = \sigma \left(\sum_j w_{ij} x_j \right), \quad (2.8)$$

we transform input maps $s \in \{1, \dots, S\}$ into output maps $m \in \{1, \dots, M\}$, such that

$$h_{ijm} = \sigma \left(\sum_s \sum_{(k,l) \in \mathcal{R}} w_{msijkl} x_{s,i+k,j+l} \right). \quad (2.9)$$

Here, location indices i and j describe the location in an image. The non-zero region of the feature, a square $(\varphi \times \varphi)$ receptive field (\mathcal{R}) is enumerated with indices k and l . In Section 5.1, we will show that not only CNN, but also semi-restricted Boltzmann machines can profit from this receptive field concept.

2.6.2 Weight Sharing

Even with local weight matrices, we can anticipate that the filters will be very redundant. Edge features, to choose a simple example, are found when applying sparse coding to image patches. They are fundamental property of natural images (Olshausen et al., 1996) and therefore biological vision (Olshausen and Field, 1997), and would need to be learned for all image locations. LeCun et al. (1998a) introduced *weight sharing* between locations to eliminate this redundancy. To realize weight sharing, we drop the image location indices on the weights, such that

$$h_{ijm} = \sigma \left(\sum_s \sum_{(k,l) \in \mathcal{R}} w_{mskl} x_{s,i+k,j+l} \right). \quad (2.10)$$

If every feature detector is applied at all image locations, this operation is equivalent to the sum of multiple convolution operations instead of the matrix multiplication in Eq. (2.8), which we can also write as

$$\mathbf{h}_m = \sigma \left(\sum_s W_{ms} * \mathbf{x}_s \right). \quad (2.11)$$

A convolutional layer applies the same M feature detectors at all image locations. If location-dependent filters are required for a task, they can be incorporated by adding non-convolutional layers at later stages in the hierarchy.

2.6.3 Pooling

The second essential component of CNN is the *pooling* layer. Similar to a convolutional layer, a pooling layer operates on receptive fields and conserves image topology. In contrast to convolutional layers, pooling is typically applied with a stride ζ of more than one pixel, does not have learnable parameters, and pools over the contents of a single map only. Thus, pooling layers serve two purposes: Reducing spatial resolution and increasing invariance to local deformations. Scherer et al. (2010) compared various pooling schemes and found that the local maximum gives the best results. For pooling with stride ζ , we have

$$h_{m,i/\zeta,j/\zeta} = \max_{(l,k) \in \mathcal{R}} x_{m,i+l,j+k} \quad , \quad (2.12)$$

with, following Scherer et al. (ibid.),

$$\frac{\partial h_{m,i/\zeta,j/\zeta}}{\partial x_{m,i+l,j+k}} = \begin{cases} 1 & \text{if } x_{m,i+l,j+k} = h_{m,i/\zeta,j/\zeta} \\ 0 & \text{else.} \end{cases} \quad (2.13)$$

The maximum operation discards location information within the receptive field. It can be regarded as a disjunction over the applications of the feature detector m at the locations in \mathcal{R} .

2.6.4 Margin Handling

Valid convolutions decrease the image size by a margin of $\lfloor \varphi/2 \rfloor$ on all sides, where φ is the filter size. To retain correspondence between input and output map coordinates, the input maps are often padded (explicitly or implicitly) with zeros. To retain the same correspondence for pooling operations, it is common to either choose the size of the input maps such that it is divisible by φ or reduce the pool sizes at the border.

2.7 CROSS-DATASET TRANSFER LEARNING

Another reason why learning gets stuck in local optima is that the datasets are often small in comparison to the number of pa-

rameters of deep neural networks. This property gives neural networks the ability to *overfit*, i.e. to learn the data distribution by heart at the expense of its generalization abilities³. This problem can be overcome if enough unlabeled data is available, with semi-supervised learning using RBMs or auto-encoders as discussed in Section 2.4.

A second approach to the small dataset problem, is to profit from other datasets entirely, if they share important characteristics with the target dataset. The most notable and recent example of this idea are image datasets. The hierarchical features learned by the CNN of Krizhevsky et al. (2012) and subsequent works on the ImageNet large scale visual recognition challenge (ILSVRC) dataset exhibit remarkable generalization abilities for a large number of vision tasks, as shown by Girshick et al. (2014), Razavian et al. (2014), and Donahue et al. (2014) and our own work detailed in Chapter 8. These CNN were trained for a supervised image classification task on a very large dataset (1.2 million images) and outperformed other approaches by a large margin. More importantly, the CNN have learned features which reflect generic properties of natural images. While their precise nature is still being investigated (Zeiler and Fergus, 2014; Springenberg et al., 2014; Szegedy et al., 2014), they were proven to provide state-of-the-art results even on non-image-classification tasks e.g. action recognition (Simonyan and Zisserman, 2014; Gkioxari et al., 2014), object-class segmentation (Girshick et al., 2014; Eigen and Fergus, 2015) and pose estimation (Schwarz, 2014; Gkioxari et al., 2014). ILSVRC-trained CNN can even be used to process depth data (Schwarz, 2014). The ability to process and learn from vast amounts of data, discovering generic features that yield superior performance on many tasks, has become a central argument for deep learning.

With similar impact, Mikolov et al. (2013) use neural networks trained to predict whether a word-context pair can appear together in a written text. The hidden encoding for the words have surprising and useful properties. For example, similar words have similar encodings and allow simple semantic vector arith-

³ also named “large p , small n problem” in the statistics literature

metic. The embedding, called “word2vec”, simplifies many tasks in the language domain similar to the visual CNN-based encodings discussed above.

2.8 CONCLUSION

We gave a brief overview of the ideas behind deep learning, a field of machine learning creating and analyzing the building blocks of feature hierarchies.

Feature hierarchies provide a space and time-efficient decomposition of inputs, which can be useful in various tasks such as classification, denoising, and compression. For a long time it was not clear how feature hierarchies can be learned. We discussed the problems encountered during learning—the large number of effective local minima and gradient dilution.

One deep learning solution to learning feature hierarchies is to solve a sequence of simple shallow problems first. In each step, deep methods learn a new level of features—gaining new insights into the input data distribution on the way. The resulting feature hierarchy can finally be adopted to an arbitrary (usually supervised) task. Another solution is to train a comparably simple task such as whole-image classification on a large dataset and then use the weights of the trained network in unrelated tasks such as segmentation or pose estimation.

While deep learning has progressed tremendously over the last years, many challenges remain. The described building blocks of deep learning are often difficult to optimize (Section 5.2). They are also restricted and cannot represent arbitrary features, since encoders are too simplistic. We believe that unsupervised pre-training of two-layer encoders (Section 5.3) may provide a remedy here. Currently, research focuses on supervised and transfer learning techniques, however. The set of learned invariances enforced through convolutions is limited, allowing only for translations and local deformations. To understand image sequences, it might be required to explicitly include transformations (Taylor et al., 2010; Memisevic, 2011) and hierarchies of transformations, so that image sequences can be modeled beyond captioning of videos (Donahue et al., 2015). Behnke (2003b) provides a possible

base architecture which we investigate in Section 6.4. Another open problem is to model the 3D structure of scenes to deal with occlusions and deal efficiently with multi-modal input (in this thesis, we incorporate depth information in Section 6.2 and Chapter 8). More context must be incorporated and world knowledge must be learned so that long texts, videos and camera streams can be interpreted.

Deep learning has become an ever-growing number of concepts and techniques. However, it is much less theoretically understood than e.g. convex optimization techniques. Training procedures and model architectures are changing quickly depending on the task and as new “tricks” are discovered. Deep learning frameworks provide tools to research, distribute and verify novel techniques. Since they have become the essential instrument for us, the following chapter gives an overview of what they must offer and how these features might be implemented, using our own CUVNET as an example.

In the last chapter, we introduced a large number of deep learning concepts and methods. While only touching on most of them, the list was by no means exhaustive. Nevertheless, deep learning research and applications depend on most of them, and new ideas are proposed every day. Deep learning software frameworks relieve the user from tedious re-implementation. For this thesis, we developed a deep learning framework which was used for the experiments in most of the chapters, as well as numerous student theses and as a teaching device.

This chapter describes how we structured the framework. We start by recapitulating the backpropagation algorithm in Section 3.1 and derive GPU-specific requirements in Section 3.2. We then describe design choices, which allow our framework to be both fast using GPU operations (Section 3.3) and flexible, using high-level model specification, learning algorithm description and debugging facilities (Section 3.4).

3.1 BACKGROUND ON THE BACKPROPAGATION ALGORITHM

The backpropagation algorithm for MLP was popularized by Rumelhart et al. (1986), with previous work by Werbos (1974) and Parker (1985). Before, only algorithms for one-layered neural networks were known, which suffered from the problem of not being able to solve the exclusive OR (XOR) problem (see also Section 5.3). The backpropagation algorithm has three stages, the forward pass, the backward pass, and the weight update, which we briefly discuss in the following, with a focus on the requirements and properties relevant to a GPU implementation.

3.1.1 Forward Pass

For all (\mathbf{x}, \mathbf{y}) in \mathcal{D} , compute the loss ℓ . We can subdivide the forward pass in two steps, where the first step computes the model prediction $\hat{\mathbf{y}}$, which the second step then evaluates in the loss function ℓ .

3.1.2 Backward Pass

For all weights w_{ij} and biases b_j , compute $\partial\ell/\partial w_{ij}$ and $\partial\ell/\partial b_j$. The backpropagation algorithm mainly provides an computationally efficient method to evaluate these gradients, with two important principles which will play a major role for the software design choices later in this chapter: Sharing computations and saving intermediate results.

SHARED COMPUTATIONS When using the chain rule, the chain from ℓ to the weights of some layer l ,

$$\frac{\partial\ell}{\partial w_{ij}^{(l)}} = \frac{\partial\ell}{\partial \mathbf{h}^{(L)}} \cdots \frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \frac{\partial \mathbf{h}^{(l)}}{\partial w_{ij}^{(l)}} \quad (3.1)$$

shares most of the chain links with the partial derivative computation for $W^{(l-1)}$

$$\frac{\partial\ell}{\partial w_{ij}^{(l-1)}} = \frac{\partial\ell}{\partial \mathbf{h}^{(L)}} \cdots \frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{h}^{(l-1)}} \frac{\partial \mathbf{h}^{(l-1)}}{\partial w_{ij}^{(l-1)}} \quad (3.2)$$

Due to the simple nature of the chain rule, shared chain links can be identified automatically as long as ℓ is given as a traversable data structure.

Shared links also trivially exist within a single layer: in Eq. (3.1), bold (vector) symbols can be substituted with any of their scalar vector components. This has two consequences: (a) all partial derivative computations in a single chain link share the same path, which only needs to be computed once, and (b) for sufficiently large vectors, this means that derivatives of a single layer can be computed in parallel using the single instruction, multiple

data (SIMD) mechanism. The CUDA architecture heavily depends on SIMD for speed.

SAVING INTERMEDIATE RESULTS The derivative of matrix multiplications, common sigmoidal non-linearities, and many other operations, has the important property that intermediate results or arguments of the forward pass also appear in the partial derivative. Thus, software implementations of DNN can save time by remembering these intermediate values. For example, the multiplication operator $\mathbf{z} = W\mathbf{x}$ needs to have access to its inputs W and \mathbf{x} to compute its partial derivatives, whereas the $\tanh(\cdot)$ operator needs to “remember” its result to compute $\partial \tanh(x) / \partial x = 1 - \tanh^2(x)$. Similarly, the mean squared loss $\ell(\mathbf{y}, \mathbf{h}^{(L)}) = \sum_i (y_i - h_i^{(L)})^2$ can remember the result of the difference operation to compute $\partial \ell / \partial h_i^{(L)} = 2(y_i - h_i^{(L)})$, and the pooling operator of Eq. (2.12) can remember the indices required to compute Eq. (2.13).

3.1.3 *Weight Update*

After computing derivatives for all parameters, we use them to update weights and biases. The simplest method here, suggested by Rumelhart et al. (1986), is gradient descent. Higher order methods (e.g. Schaul et al., 2013) have been shown to require fewer updates but require more computational effort and make possibly unwarranted assumptions about the nature of the loss function. Updating the weights typically requires at least a learnrate hyper-parameter, which needs to be grid-searched with cross-validation (see Bergstra et al., 2011, for a discussion of the efficiency of grid-search and its alternatives in this context). Sometimes, different layers of the network can require higher or lower learnrates. Adaptive learning rate schemes such as RPROP (Riedmiller and Braun, 1993), RMSProp (Dauphin et al., 2015) or AdaGrad (Duchi et al., 2011) can be used to reduce the time-consuming tuning and cross-validation steps, or enable learning in case of recurrent neural networks (Pavel et al., 2015). Per-layer learning rates can also be obsoleted by clever initialization

schemes (e.g. Glorot and Bengio, 2010; He et al., 2015), which attempt to equalize the initial magnitude of neuron activations and gradients throughout the DNN.

3.2 GPU CONSIDERATIONS

When implementing DNN backpropagation for the GPU, we have to take the peculiarities of the parallel GPU hardware and CUDA programming model into account. In the following, we discuss a non-exhaustive list of such considerations.

MINIBATCH LEARNING All three steps of the backpropagation algorithm can be computed for the whole dataset (called batch learning), a small subset of the dataset (minibatch learning), or single elements in the dataset (online learning). Batch learning computes the exact empirical gradient on the training set. For larger datasets, this requires a lot of computation time for a single weight update. The gradients for subsets of the dataset are typically good approximations to the dataset gradient, and it is not necessary to spend much time to compute it exactly, resulting in a tradeoff between number of weight updates per second and gradient precision. While e.g. LeCun et al. (1998b) recommend the relatively noisy online-learning, the advancement of GPUs with the possibility to compute gradients for many examples in parallel resulted in minibatch learning as a quasi-standard, with mini-batch sizes chosen to maximize GPU throughput and memory use (e.g. Krizhevsky et al., 2012).

LIMITED MEMORY The GPU has its own, usually superior but small, on-board memory, which — in current CUDA versions — has to be managed explicitly by the programmer. For the GPUs used in this thesis, the on-board memory is comparatively small. Transfers to and from the GPU memory are slow and must be kept to a minimum. A typical approach is to copy the data from the dataset to the GPU memory, then compute forward and backward pass, and update the weights — all on the GPU. Finally, statistics for monitoring learning progress, such as the loss value, are copied to the main memory. During the forward pass, intermedi-

ate values must be remembered for the backward pass. Especially when working with high resolution or multi-resolution images (e.g. with CNN, Section 2.6), these intermediate values can be very large. In our implementation we need to make sure that:

1. Results of a single operation can be read by multiple consecutive operations without being copied. For example, the DNN prediction is required to compute (a) the logistic loss and (b) compute the zero-one loss (i.e. the number of misclassifications).
2. If possible, we want to be able to reuse memory of intermediate results that are not required anymore. For example, the $\tanh(\cdot)$ operator only needs to remember $\mathbf{h}^{(1)} = \tanh(\mathbf{x})$ for the backward pass, since $\partial \mathbf{h}^{(1)} / \partial \mathbf{x} = (1 - \tanh^2(\mathbf{x}))$. Thus, it can directly overwrite \mathbf{x} . This should not be possible, however, if (a) \mathbf{x} appears somewhere else in the loss in an operator which has not yet been evaluated, or (b) \mathbf{x} is needed for another partial derivative in the backward pass.

Note that the two goals—avoiding copying and allowing memory reuse—can be in conflict, as the memory reuse example shows. This problem is related to register reuse and is typically addressed by compiler optimization. While this kind of optimization is outside the scope of this thesis, we will show in the following that for DNNs, simple heuristics can produce the same effect.

3.3 CUV: N-DIMENSIONAL ARRAYS IN CUDA

To realize the GPU requirements of the previous section, we implemented two dependent libraries, CUV and CUVNET. The CUV library does all the heavy lifting on GPU, whereas CUVNET provides a high-level DNN machine learning API.

Current GPU hardware and programming models such as CUDA and OpenCL rely on the SIMD principle (Flynn, 1972) for speed. Similarly to interpreted languages like MATLAB and Python, the CUV API focuses on providing data structures for bundling data and vectorized functions to operate on them. Vectorization groups element-wise operations together (van der Walt et al.,

Algorithm 1 demonstration of CUV features

```

typedef ndarray<float, dev_memory_space> dev_ndarray;      1
typedef ndarray<float, host_memory_space> host_ndarray;    2
dev_ndarray v(extents[5]), m(extents[5][7]); // Declaration  3
host_ndarray hm;                                           4
                                                                 5
m = 2.f; // m[:] = 2.0                                     6
m += m; // copy to RAM                                    7
reduce_to_col(v, m); //  $v_i := \sum_j m_{ij}$ , dispatched to GPU 8
hm = m; // copy to RAM                                    9
hm = hm[indices[range(0,3)]]; // keep first 3 rows, no copy 10
ofstream os("m.ser");                                       11
binary_oarchive oa(os);                                     12
oa << hm << m; // serialization                            13

```

2011), and in our case, hides the GPU implementation details from the programmer. Algorithm 1 shows CUV usage, demonstrating the most relevant features.

DATASTRUCTURES The main CUV datastructure is a multi-dimensional array. Similar datastructures form the basis of e.g. MATLAB¹ and NumPy (van der Walt et al., 2011). The CUV library borrows its concepts mostly from NumPy and Ron Garcia’s Boost.MultiArray². Multi-dimensional arrays allow for intuitive specification and passing of tabular data (Lines 3 and 8, respectively). A common example is a number of B images of size $H \times W$ with M channels each. In CUV, this can be declared as

```

ndarray<float, host_memory_space> data(extents[B][M][H][W
↪]);

```

, where the dimensions — the *shape* — are stored in data. The dimensions do not carry inherent meaning, such that e.g. weight matrices from M input maps to S output maps with filter dimension $\zeta \times \zeta$ can be declared with exactly the same syntax:

¹ <http://www.mathworks.com>

² http://www.boost.org/doc/libs/release/libs/multi_array

```
ndarray<float, host_memory_space> weights(extents[M][S][ξ][
↪ξ]);
```

The shape can be changed independent of the underlying memory and the dimensions can be referenced by their index in the shape array, e.g. to determine the sum of all maps in all images:

```
data.reshape(extents[I][B][H*W]); // collapse last two dimensions
data.sum(2);                       // sum over pixels
```

Another essential property of multi-dimensional arrays is slicing. Slicing allows to select a part of the data structure — without touching the underlying memory — and continue processing it. Line 10 shows an example where a subset of the rows of a matrix is sliced. Finally, the `ndarray` supports serialization using the `Boost.Serialization`³ interface. This format allows `ndarrays` to be transparently saved as part of more complex DNN data structures (Section 3.4).

TAG-DISPATCHED FUNCTION CALLS The implementation of the multi-dimensional array type `ndarray` is generic with respect to its `value.type` and the memory where it resides, i.e. GPU or central processing unit (CPU) memory — associated with the tags `dev_memory_space` and `host_memory_space`, respectively. All operations where memory is modified (e.g., Line 8) are then selected using the tag given when declaring an `ndarray` variable. Dispatching on tags is necessary since the compiler cannot distinguish GPU and CPU pointer types under the CUDA programming model. Additionally, the tags allow the user to write template functions which e.g. perform operations and create temporaries in the same memory space as their parameters.

EFFICIENT, INSTANTIATED KERNELS All operations on `ndarrays` are implemented for GPU and CPU. Tests ensure that the outcome is the same up to machine precision. Depending on value type, memory layout and data size of the input, one of multiple kernels may be chosen to run, which is transparent to the user.

³ <http://www.boost.org/doc/libs/release/libs/serialization>

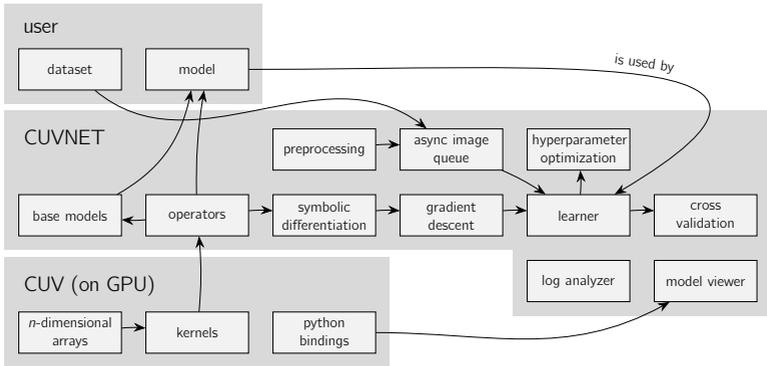


Figure 3.1: CUVNET design. Arrows represent a “used by” relation.

Due to explicit template instantiations, the C++ header files of CUV are kept free of CUDA includes. This allows the use of arbitrary IDEs without CUDA support and faster compilation of programs using CUV.

MEMORY POOLING In the CUDA programming model, kernel calls are asynchronous. This enables interleaved CPU and GPU computations. Apart from explicit synchronization calls, the GPU is synchronized with the main thread only for memory copies, allocation, and deallocation. We use the CPU mainly to determine the next kernel call. This small application overhead can be completely hidden by the kernel running time, but it often involves allocation of temporaries and deallocation of variables going out of scope, both of which would force synchronization. The CUV library implements its own memory pooling to work around this CUDA limitation. Allocation and deallocation are handled by the memory pool, which only involves the CUDA API if it is unable to handle the request itself, effectively removing allocation and deallocation-triggered synchronization of the GPU thread.

3.4 GENERIC MODEL OPTIMIZATION WITH CUVNET

In this section, we describe how the CUVNET library built on top of CUV facilitates model optimization and deep learning research. An overview over the library is given in Fig. 3.1

3.4.1 *Requirements of Deep Learning Algorithms*

Before considering the design choices in CUVNET, we list essential requirements for deep learning libraries.

A fast GPU implementation is a necessity, since we need to be able to (a) train large models fast, and (b) try many (smaller) models simultaneously.

The user must be able to build models from reliable, tested building blocks. Models can be quickly combined to build larger models, creating abstraction layers. This enables the user to (a) write down loss functions easily; (b) store and load models at high level, and (c) build repetitive models by chaining abstract base models.

Standard machine learning tools must be readily available to allow exploring the huge space of hyper-parameters in deep learning. To test hypotheses quickly, we should provide at least (a) access to multiple datasets; (b) multiple gradient methods; (c) standardized training protocols, and (d) extensive debugging and logging tools

3.4.2 *Fast GPU Implementation*

CUVNET realizes this requirement by relying on the optimized CUV for non-trivial computations. The generic nature of the `ndarray` class also allowed us to wrap and switch between third-party functions, e.g. convolutions from Alex Krizhevsky's `cuda-convnet` library (Krizhevsky, 2015) or the CUDNN library⁴ developed by NVidia.

⁴ <https://developer.nvidia.com/cuDNN>

3.4.3 *User Interface*

Many deep learning frameworks, such as `pylearn2` (Goodfellow et al., 2013a), `cuda-convnet` (Krizhevsky, 2015), or `caffe` (Jia et al., 2014), rely mainly on declarative languages (i.e. configuration files) to specify network architecture and training parameters. Declarative languages can be very concise, but do not scale well:

1. Defaults and documentation are not available via IDE when writing custom language specifications.
2. High-level abstractions are key when specifying complex models. Examples are code reuse, e.g. for combinations of convolution, pooling, non-linearity operations, or looping, e.g. for networks repeated over time steps in a recurrent neural network. Specifying these require either complex declarative language—which the mentioned libraries avoid—or configuration files generated in another programming language.
3. Training of DNN can be complicated. For example, unsupervised learning may be done on a different dataset than supervised learning, with different learning algorithms (RBM vs. gradient descent), different loss functions. Advanced fine-tuning methods add the validation dataset to the training set after early-stopping and continue training until a specified loss is reached. These complex algorithms are much easier expressed in a programming language.
4. Hyper-parameters of architecture (e.g. number of neurons in a layer) and training (e.g. learnrates) can be tuned automatically using cross-validation and grid search. Again, configuration files complicate this process avoidably.

CUVNET takes the approach of a library, i.e. all functionality is written in C++, and it is available in C++. CUVNET also provide a python interface with the same functionality. Specifications of network architecture is as concise as possible.

The basic user interface of CUVNET is shown in Algorithm 2. Line 4 constructs a linear transformation of a 1000 sample dataset with 20 variables $x \in \mathbb{R}^{1000,20}$. This estimator is used in Line 5

Algorithm 2 Demonstration of basic CUVNET interface.

```

x = input(extents[1000][20]);           1
y = input(extents[1000][10]);         2
W = input(extents[20][10]);           3
est = prod(x, W);                       4
loss = mean(logistic_loss(est, y));    5
gradient_descent gd(loss, {W}, learnrate); 6
// [...] put data in x, y, randomize W  7
gd.batch_learning(n_epochs);          8
ofstream os("model.ser");              9
binary_oarchive oa(os);                10
oa << est << m; // Model serialization  11

```

to build a multinomial logistic loss function, interpreting the output of `est` as log-likelihoods for one of 10 classes for each sample, and takes the mean over the dataset. Line 6 builds a gradient descent object which minimizes `loss` with respect to `W` using plain gradient descent with the given `learnrate`. The final lines show how learning is executed, and the model is saved to a file with the same mechanism used for CUV in Algorithm 1.

The gradient descent object can be modified using a signal-slot mechanism. E.g., in every iteration, new input can be loaded into the network and saved after processing. Every K epochs, the model itself can be saved to a file, or it can be evaluated on a validation set for early stopping. A monitor can calculate summary statistics on activations. A large number of such signals and slots is provided with CUVNET.

3.4.4 Complex Models

The model class is an abstraction around loss functions, which provides convenient loss function constructors, access to input ops, learnable parameter operators, loss function and prediction function, but most importantly, hierarchical structure.

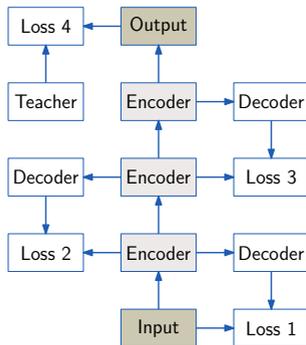


Figure 3.2: Multistage model training in a simplified pre-training/fine-tuning model. The complete model graph is constructed first, then the four loss functions are minimized consecutively.

METAMODELS The need for hierarchical structure becomes evident when we analyze typical DNN models. While the learned features become more and more abstract, the functions computed in the layers of the models are usually the same. All layers of the classical MLP, for example, have the same internal structure. To reflect this observation, CUVNET adopts the notion of metamodels. Metamodels are models which—by default—inherit the parameters, inputs, and monitoring variables of the models they subsume. A common use-case is to build an MLP given a set of pre-trained chainable encoders as described in Section 2.4.

MULTISTAGE MODELS The greedy layerwise training introduced in Section 2.4 can be re-realized in CUVNET as shown in Fig. 3.2. First, a model graph is constructed which contains multiple loss functions. The loss functions are then optimized in succession. During pre-training, the models being optimized are the auto-encoders closest to the respective loss function, while all previous (lower) encoder weights are fixed. These lower encoders can be automatically substituted by CUVNET with input nodes which supply a transformed version of the original dataset, avoiding costly reevaluation.

3.4.5 *Differentiation Graph*

We saw in Section 3.1 if the loss is given as a traversable graph data structure, the gradient of the loss function with respect to the weights can be computed automatically and efficiently. Fig. 3.3 shows how the loss graph is constructed in CUVNET. The loss is a directed acyclic tripartite graph, with node types operator, parameter, and result. Each operator may have zero, one, or more parameters. Operators without parameters are called inputs. Inputs can be dataset mini-batches, but also weight matrices or bias vectors. Similarly, operators can have zero, one, or more results. Operators without results are called sinks. Sinks store values that can be retrieved after evaluating the graph. The result of an operator can be used as the parameter of another operator. For symmetry reasons, we also allow that multiple results can project to a single parameter, interpreted as an implicit addition. The symmetry lies in the fact that the derivative of multiple use of the same result also produces an addition in the derivative.

A topological sorting (Kahn, 1962) of the graph yields the evaluation order. The execution order differs between forward and backward pass, since during the backward pass, we only need to follow paths which lead to learnable parameters.

Complex pointer-based data structures need special care to avoid memory leaks. In CUVNET, references to operator nodes are stored in reference-counting pointers. These references are kept internally (by neighboring graph nodes) and by the user. Internally, reference-counting pointers are only used in the direction opposite to the graph edge direction. Thus, every operator node that is not stored explicitly by the user will automatically be destroyed, which will recursively trigger the destruction of non-referenced child nodes. References in graph edge direction are kept as “weak” pointers, which are not reference-counted (Karls-son, 2002).

3.4.6 *Copy-on-write with Single Reference Handling*

Every node has functions to compute the forward pass and the backward pass. The the computed values are passed through the

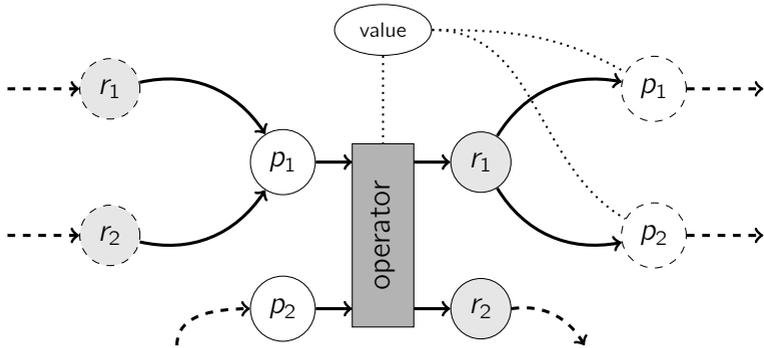


Figure 3.3: Detail of a CUVNET graph structure. Arrows depict information flow. Operators have an arbitrary number of parameters (p_i) and results (r_i). Results can be used as parameters of other operators. Data are passed as CUV ndarrays from results to parameters via copy-on-write pointers. Overwriting of data is allowed if only one reference to them exists.

graph wrapped by a copy-on-write pointer with special semantics. The concept is illustrated in Fig. 3.3 for the forward pass. After result r_1 is computed, it is stored in `value` and references to it (dotted lines) are pushed to all parameters p_i of the operators that need it. If multiple operators need the result, the first operator cannot overwrite it. If it tries to overwrite `value` nonetheless, it needs to be copied. Only the last operator keeping a reference to `value` is allowed to overwrite it. These constraints are transparently handled by the `cow_ptr` class of CUVNET. With their help, many computations can be performed without new allocations, as shown in Fig. 3.4, at the cost of implementing three versions of the operator,

1. a version that overwrites an already allocated target
2. a version that adds to a target set by another operator, and
3. a version that pushes a newly allocated result to multiple targets.

The first two versions are optional, since they can be emulated by the third. After the forward-pass, operators can decide whether to surrender their right to read a variable, or retain it for the

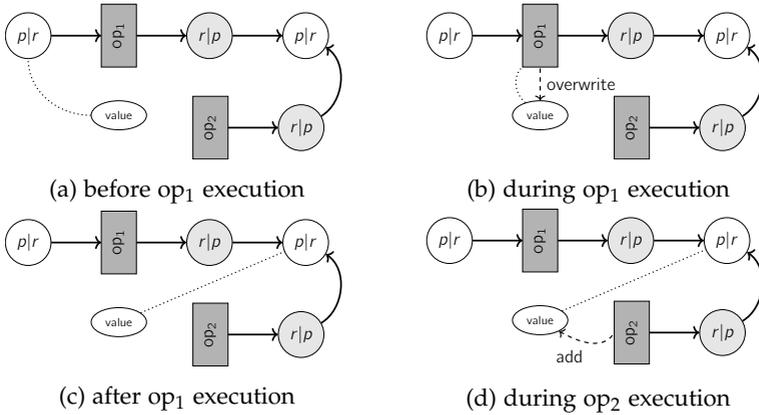


Figure 3.4: Memory-efficient computation: a value stored in our own copy-on-write pointer is changed and passed through without allocations. If at an point two pointers to value had existed, a copy would have been created in the memory pool before writing. Due to symmetry of forward/backward pass, result (r) and parameter (p) nodes are interchangeable.

backward pass. These semantics prevent undesired interaction between operator calls and unnecessary copies at the same time, as desired in Section 3.2.

By rearranging the evaluation order of the operators, memory efficiency could be further improved. Theano (Bergstra et al., 2010) and frameworks based on it take this approach, by optimizing the python-specified execution graph and generating C++ and CUDA code. This type of optimization is outside the scope of this thesis.

3.4.7 Monitoring and Visualization

Due to the lengthy training process of neural networks, it is crucial to monitor key indicators of training progress. Examples, besides loss on and training set to measure generalization, are vari-

ance of weights and activations (Glorot and Bengio, 2010), and the average relative weight change,

$$\frac{\eta}{\langle w_i \rangle_i} \left\langle \left| \frac{\partial \ell}{\partial w_i} \right| \right\rangle_i. \quad (3.3)$$

CUVNET realizes monitoring either with nodes inside the loss graph (intrusive) or by adding sink nodes to an existing graph. Models register sinks with a monitor instance, which computes summary statistics and logs the information to an XML file. The CUVNET logviewer script can interpret these files and plot them. It is an essential tool when comparing different training runs.

After training and at critical stages (e.g. when switching from pre-training to fine-tuning), the model is serialized. Serialized models can be loaded to continue the training process with different parameters.

The serialized model can also be loaded using an interactive GraphViz display, which allows to localize errors in the network graph construction. Most importantly, it can be used to inspect the processing at every node using histograms, weight visualizations, or feature map visualizations. For this purpose, the user loads new inputs from a dataset into the model and clicks on the node to see activations. Modifiers allow inspection of gradients at the node.

3.4.8 Testing

A strong motivation for developing libraries for deep learning is testing all components in isolation before building more complex models. CUV and CUVNET test the correctness and speed:

- CUV's CPU functions should produce correct results for small examples,
- CUV's GPU functions should return the same as corresponding CPU functions on larger examples,
- A CUVNET-computed operator Jacobian should be the same as the Jacobian computed via finite differences.

The last item can be non-trivial if, for example,

- the computation involves random numbers,
- the computation is numerically unstable (e.g. involving the exp function for unbounded inputs)
- the operator is not differentiable, e.g. the popular $\max(\cdot)$ non-linearity,
- the Jacobian matrix does not fit into the memory or takes a long time to compute.

The CUVNET test suite provides support for some of these cases, and attempts work around the others to produce a large code coverage for a variety of inputs.

3.4.9 *Other Major Features*

CUVNET implements multiple gradient methods (e.g. momentum, Nesterov accelerated (Sutskever, 2013), RPROP (Riedmiller and Braun, 1993), RMSProp (Dauphin et al., 2015), AdaGrad (Duchi et al., 2011)). It also provides multiple regularization methods (e.g. L1 and L2 weight decay, early stopping, dropout, learnrate scheduling based on fixed or adaptive schedules).

In CUVNET, all concepts and learning algorithms are strictly written in C++. The user, too, specifies the model in this language. This facilitates debugging and lowers complexity. However, analysis of the resulting models is written in Python, since much better tools for visualization and plotting are available. For this purpose, CUV and CUVNET both provide a Python interface, to cooperate e.g. with NumPy and make the model graph structure traversable.

3.5 COMPARISON WITH OTHER NEURAL NETWORK SOFTWARE

While in 2014, a new deep learning framework was released every 47 days on average, that number dropped to 22 days in 2015⁵. Some of these frameworks are backed by large companies, such as TensorFlow (Google) or torch (facebook). They range from fairly low-level (theano, Bergstra et al. 2010) to high-level (Pylearn2, Goodfellow et al. 2013a, built on top of theano). They

⁵ <https://goo.gl/BnS4J5>

may be largely single-language (Caffe, Jia et al. 2014, or mixed (theano, TensorFlow). Some even forego the use of GPU (convnetjs, or DistBelief, Dean et al. 2012). These frameworks are open source, while e.g. the price-winning framework of Cireşan et al. (2013) and Cireşan et al. (2012a) is not, and OverFeat (Sermanet et al., 2013) is only partially. Most software frameworks rely on a small number of implementations for operations which dominate running time, such as matrix multiplications (e.g. NVidia’s cuBLAS) and convolutions (e.g. NVidia’s cuDNN).

Any comparison with these other software will quickly become obsolete. CUVNET is by no means perfect, but adheres to some principles which — at least in this combination — we think are distinguishing:

- single-language implementation with identical GPU and CPU operations, which facilitates debugging,
- model specification in the same language, with high-level declarative syntax, and without requiring knowledge about GPU programming.
- allowing procedural building of models and metamodels. Custom languages such as configuration files often do not provide this flexibility.
- a formal way of keeping statistics during training
- interactive model introspection GUI and visualization capabilities through Python-bindings
- focus on parallel online pre-processing to create infinite datasets

However, CUV/CUVNET also lacks important features provided by other frameworks, such as a speed-optimized CPU runtime for camera streams, symbolic function optimization, arbitrary length sequence inputs for RNN, and more.

3.6 ACKNOWLEDGMENTS

Apart from the author of this thesis, CUV and CUVNET also owe much to Andreas Müller and University Bonn students Benedikt Waldvogel, Mircea Pavel, Nico Höft, Lukas Koliogiannis, Tobias

Hartmann, and Yi Huang. Their contributions and comments helped to shape the current general-purpose system.

3.7 APPLICATIONS

Apart from the publications listed in Section 1.2⁶, elements of CUVNET and CUV were used as a teaching device in University Bonn courses, in the theses of Tobias Hartmann, Lukas Koliogiannis, Yi Huang, Nico Höft, and Mircea Pavel, as well as in publications by Kyunghyun Cho, Müller et al. (2010), Höft et al. (2014), Müller and Behnke (2014), and Stückler et al. (2014).

An important use of CUV is the CURFIL software package, which is built around the CUV data structures. CURFIL was built with one application in mind: image labeling, or classifying pixels according to object classes. The next chapter describes CURFIL with a focus on the GPU challenges.

⁶ with the exception of Chapter 8

Random forests are ensemble classifiers that are popular in the computer vision community. Random decision trees are used when the hypothesis space at every node is huge, so that only a random subset can be explored during learning. This restriction is countered by constructing an ensemble of independently learned trees—the random forest.

Variants of random forests were used in computer vision to improve e. g. object detection or image segmentation. One of the most prominent examples is the work of Shotton et al. (2011), who use random forests in Microsoft’s Kinect system for the estimation of human pose from single depth images. Here, we are interested in the more general task of object class segmentation (sometimes called image labeling), i. e. determining a label for every pixel in an RGB or RGB-D image.

The real-time applications such as the ones presented by Lepetit et al. (2005) and Shotton et al. (2011) require fast prediction in few milliseconds per image. This is possible with parallel architectures such as GPUs, since every pixel can be processed independently. Random forest training for object class segmentation, however, is not as regular—it is a time consuming process. To evaluate a randomly generated feature candidate in a single node of a single tree, a potentially large number of images must be accessed. With increasing depth, the number of pixels in an image arriving in the current node can be very small. It is therefore essential for the practitioner to optimize memory efficiency in various regimes, or to resort to large clusters for the computation. Furthermore, changing the visual features and other hyperparameters requires a re-training of the random forest, which is costly and impedes efficient scientific research.

This work describes the architecture of our open-source GPU implementation of random forests for image labeling (CURFIL). CURFIL provides optimized CPU and GPU implementations for the

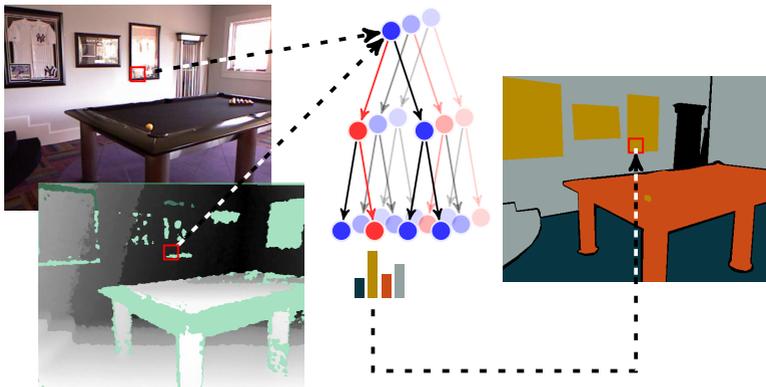


Figure 4.1: Overview of object class segmentation with random forests: Every pixel (RGB and depth) is classified independently based on its context by the trees of a random forest. The leaf distributions of the trees determine the predicted label.

training and prediction of random forests. Our library trains random forests up to 26 times faster on GPU than our optimized multi-core CPU implementation. Prediction is possible in real-time speed on a single mobile GPU.

In short, our contributions are as follows:

1. We describe how to efficiently implement random forests for object class segmentation on GPU,
2. We describe a method which allows to train on horizontally flipped images at significantly reduced cost,
3. we show that our GPU implementation is up to 26 times faster for training (up to 48 times for prediction) than an optimized multi-core CPU implementation,
4. We show that simply by the now feasible optimization of hyper-parameters, we can improve performance in two object class segmentation tasks, and
5. we make our documented, unit-tested, and MIT-licensed source code publicly available¹.

¹ <https://github.com/deeplearningais/curfil/>

The remainder of this chapter is organized as follows. After discussing related work, we introduce random forests and our node tests in Sections 4.2 and 4.3, respectively. We describe our optimizations in Section 4.4. Section 4.5 analyzes speed and accuracy attained with our implementation.

4.1 RELATED WORK

Random forests were popularized in computer vision by Lepetit et al. (2005). Their task was to classify patches at pre-selected keypoint locations, not — as in this work — all pixels in an image. Random forests proved to be very efficient predictors, while training efficiency was not discussed. Later work focused on improving the technique and applying it to novel tasks.

Lepetit and Fua (2006) use random forests to classify keypoints for object detection and pose estimation. They evaluate various node tests and show that while training is increasingly costly, prediction can be very fast.

The first GPU implementation for our task was presented by Sharp (2008), who implements random forest training and prediction for Microsoft's Kinect system that achieves a prediction speed-up of 100 and training speed-up factor of eight on a GPU, compared to a CPU. This implementation is not publicly available and uses Direct3D which is only supported on the Microsoft Windows platform.

An important real-world application of object class segmentation with random forests is presented by Shotton et al. (2011). Human pose estimation is formulated as a problem of determining pixel labels corresponding to body parts. The authors use a distributed CPU implementation to reduce the training time, which is nevertheless one day for training three trees from one million synthetic images on a 1,000 CPU core cluster. Their implementation is also not publicly available.

Several fast implementations for general-purpose random forests are available, notably in the scikit-learn machine learning library (Pedregosa et al., 2011) for CPU and CudaTree (Liao et al., 2013) for GPU. General random forests cannot make use of texture caches optimized for images though, i.e., they treat

all samples separately. GPU implementations of general-purpose random forests also exist, but due to the irregular access patterns when compared to object class segmentation problems, their solutions were found to be inferior to CPU (Slat and Lapajne, 2010) or focused on prediction (Van Essen et al., 2012). For some use cases, the general-purpose implementation of Jansson et al. (2014) seems promising.

The prediction speed and accuracy of random forests facilitates applications interfacing computer vision with robotics, such as semantic prediction in combination with self localization and mapping (Stückler et al., 2012) or 6D pose estimation (Rodrigues et al., 2012) for bin picking.

CURFIL was successfully used by Stückler et al. (2014) to predict and accumulate semantic classes of indoor sequences in real-time, and by Müller and Behnke (2014) to significantly improve segmentation accuracy on a benchmark dataset.

4.2 RANDOM FORESTS

Random forests — also known as random decision trees or random decision forests — were independently introduced by Ho (1995) and Amit and Geman (1997). Breiman (2001) coined the term “random forest”. Random decision forests are ensemble classifiers that consist of multiple decision trees — simple, commonly used models in data mining and machine learning. A decision tree consists of a hierarchy of questions that are used to map a multi-dimensional input value to an output which can be either a real value (regression) or a class label (classification). Our implementation focuses on classification but can be extended to support regression.

To classify input x , we traverse each of the K decision trees \mathcal{T}_k of the random forest \mathcal{F} , starting at the root node. Each inner node defines a test with a binary outcome (i. e. true or false). We traverse to the left child if the test is positive and continue with the right child otherwise. Classification is finished when a leaf node $l_k(x)$ is reached, where either a single class label or a distribution $p(c | l_k(x))$ over class labels $c \in \mathcal{C}$ is stored.

The K decision trees in a random forest are trained independently. The class distributions for the input x are collected from all leaves reached in the decision trees and combined to generate a single classification. Various combination functions are possible. We implement majority voting and the average of all probability distributions as defined by

$$p(c | \mathcal{F}, x) = \frac{1}{K} \sum_{k=1}^K p(c | l_k(x)).$$

A key difference between a decision tree and a random decision tree is the training phase. The idea of random forests is to train multiple trees on different random subsets of the dataset and random subsets of features. In contrast to normal decision trees, random decision trees are not pruned after training, as they are less likely to overfit (*ibid.*). Breiman’s random forests use CART as tree growing algorithm and are restricted to binary trees for simplicity. The best split criterion in a decision node is selected according to a score function measuring the separation of training examples. CURFIL supports information gain and normalized information gain (Wehenkel and Pavella, 1991) as score functions.

A special case of random forests are random ferns, which use the same feature in all nodes of a hierarchy level. While our library also supports ferns, we do not discuss them further in this chapter, as they are neither faster to train nor did they produce superior results.

4.3 VISUAL FEATURES FOR NODE TESTS

Our selection of features was inspired by Lepetit et al. (2005)—the method for visual object detection proposed by Viola and Jones (2001). We implement two types of RGB-D image features as introduced by Stückler et al. (2012). They resemble the features of Sharp (2008) and Shotton et al. (2011)—but use depth-normalization and region averages instead of single pixel values. Shotton et al. (2011) avoid the use of region averages to keep computational complexity low. For RGB-only datasets, we employ the same fea-

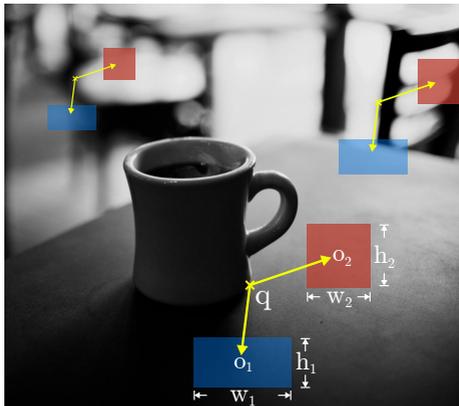


Figure 4.2: Sample visual feature at three different query pixels. Feature response is calculated from difference of average values in two off-set regions. Relative offset locations \mathbf{o}_i and region extents w_i, h_i are normalized with the depth $d(\mathbf{q})$ at the query pixel \mathbf{q} .

tures but assume constant depth. The features are visualized in Fig. 4.2.

For a given query pixel \mathbf{q} , the image feature φ_θ is calculated as the difference of the average value of the image channel χ_i in two rectangular regions R_1, R_2 in the neighborhood of \mathbf{q} . Size w_i, h_i and 2D offset \mathbf{o}_i of the regions are normalized by the depth $d(\mathbf{q})$:

$$\varphi_\theta(\mathbf{q}) := \frac{1}{|R_1(\mathbf{q})|} \sum_{\mathbf{p} \in R_1} \chi_1(\mathbf{p}) - \frac{1}{|R_2(\mathbf{q})|} \sum_{\mathbf{p} \in R_2} \chi_2(\mathbf{p})$$

$$R_i(\mathbf{q}) := \left(\mathbf{q} + \frac{\mathbf{o}_i}{d(\mathbf{q})}, \frac{w_i}{d(\mathbf{q})}, \frac{h_i}{d(\mathbf{q})} \right). \quad (4.1)$$

CURFIL optionally fills in missing depth measurements using the colorization method of Levin et al. (2004). We use integral images to efficiently compute region sums. The large space of eleven feature parameters—region sizes, offsets, channels, and thresholds—requires to calculate feature responses on-the-fly since pre-computing all possible values in advance is not feasible.

Algorithm 3 Training of random decision tree

Require: \mathcal{D} training instances
Require: F number of feature candidates to generate
Require: P number of feature parameters
Require: T number of thresholds to generate
Require: stopping criterion (e. g. maximal depth)

- 1: $D \leftarrow$ randomly sampled subset of \mathcal{D} ($D \subset \mathcal{D}$)
- 2: $N_{\text{root}} \leftarrow$ create root node
- 3: $C \leftarrow \{(N_{\text{root}}, D)\}$ ▷ initialize candidate nodes
- 4: **while** $C \neq \emptyset$ **do**
- 5: $C' \leftarrow \emptyset$ ▷ initialize new set of candidate nodes
- 6: **for all** $(N, D) \in C$ **do**
- 7: $(D_{\text{left}}, D_{\text{right}}) \leftarrow \text{EVALBESTSPLIT}(D; F, P, T)$
- 8: **if** $\neg \text{STOP}(N, D_{\text{left}})$ **then**
- 9: $N_{\text{left}} \leftarrow$ create left child for node N
- 10: $C' \leftarrow C' \cup \{(N_{\text{left}}, D_{\text{left}})\}$
- 11: **if** $\neg \text{STOP}(N, D_{\text{right}})$ **then**
- 12: $N_{\text{right}} \leftarrow$ create right child for node N
- 13: $C' \leftarrow C' \cup \{(N_{\text{right}}, D_{\text{right}})\}$
- 14: $C \leftarrow C'$ ▷ continue with new set of nodes

4.4 CURFIL SOFTWARE PACKAGE

CURFIL's speed is the result of careful optimization of GPU memory throughput. This is a non-linear process to find fast combinations of memory layouts, algorithms and exploitable hardware capabilities. In the following, we describe the most relevant aspects of our implementation.

USER API The CURFIL software package includes command line tools as well as a library for random forest training and prediction. Inputs consist of images for RGB, depth, and label information. Outputs are forests in JSON format for training and label-images for prediction. Datasets with varying aspect ratios are supported.

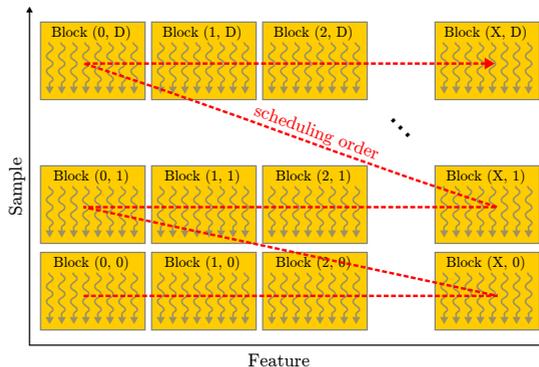


Figure 4.3: Feature Response Kernel. Two-dimensional grid layout of the feature response kernel for D samples and F features. Each block contains n threads. The number of blocks in a row, X , depends on the number of features. $X = \lceil F/n \rceil$. Feature responses for a given sample are calculated by the threads in one block row. The arrow (red dashes) indicates the scheduling order of blocks.

Our source code is organized such that it is easy to improve and change the existing visual feature implementation. It is developed in a test-driven process. Unit tests cover major parts of our implementation.

CPU IMPLEMENTATION Our CPU implementation is based on a refactored, parallelized and heavily optimized version of the Tuwo Computer Vision Library² by Nowozin. Our optimizations make better use of CPU cache by looping over feature candidates and thresholds in the innermost loop, and by sorting the dataset according to image ID before learning. Since feature candidate evaluations do not depend on each other, we can parallelize over the training set and make use of all CPU cores even when training only a single tree.

GPU IMPLEMENTATION Evaluation of the optimized random forest training on CPU (Algorithm 3) shows that the vast majority of time is spent in the evaluation of the best split feature. This is

² <http://www.nowozin.net/sebastian/tuwo/>

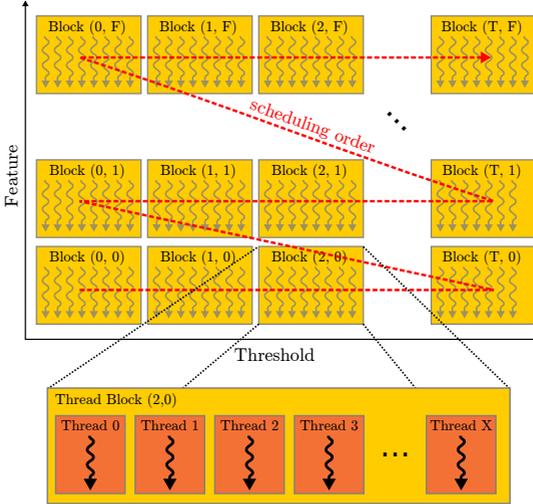


Figure 4.4: Histogram Aggregation Kernel. Thread block layout of the histogram aggregation kernel for F features and T thresholds. One thread block per feature and per threshold. X threads in block aggregate histogram counters for D samples in parallel. Every thread iterates over at most $\lceil D/X \rceil$ samples.

Algorithm 4 CPU-optimized feature evaluation

Require: D samples

Require: $\mathbf{F} \in \mathbb{R}^{F \times P}$ random feature candidates

Require: $\mathbf{T} \in \mathbb{R}^{F \times T}$ random threshold candidates

- 1: $\mathbf{H} \in \mathbb{R}^{F \times T} \leftarrow$ zeros
 - 2: initialize histograms for every feature/threshold
 - 3: **for all** $d \in D$ **do**
 - 4: **for all** $f \in 1 \dots F$ **do**
 - 5: $\varphi_{fd} \leftarrow \text{CALCRESPONSE}(\mathbf{f}_{f,}, d)$
 - 6: **for all** $t \in 1 \dots T$ **do**
 - 7: **if** $\varphi_{fd} < \mathbf{t}_{ft}$ **then**
 - 8: $h_{ft} \leftarrow h_{ft} + 1$
 - 9: calculate impurity scores for all histograms
 - 10: **return** histogram with best score
-

to our benefit when accelerating random forest training on GPU. We restrict the GPU implementation efforts to the relatively short feature evaluation algorithm (Algorithm 4) as a drop-in replacement and leave the rest of the CPU computation unchanged. We use the CPU implementation as a reference for the GPU and ensure that results are the same in both implementations.

Split evaluation can be divided into the following four phases that are executed in sequential order:

1. Random feature and threshold candidate generation,
2. Feature response calculation,
3. Histogram aggregation for all features and threshold candidates, and
4. impurity score (information gain) calculation.

Each phase depends on results of the previous phase. As a consequence, we cannot execute two or more phases in parallel. The CPU can prepare data for the launch of the next phase, though, while the GPU is busy executing the current phase.

4.4.1 GPU Kernels

RANDOM FEATURE AND THRESHOLD CANDIDATE GENERATION A significant amount of training time is used for generating random feature candidates. The total time for feature generation increases per tree level since the number of nodes increases as trees are grown.

The first step in the feature candidate generation is to randomly select feature parameter values. These are stored in a $F \times 11$ matrix for F feature candidates and eleven feature parameters of Eq. (4.1). The second step is the selection of one or more thresholds for every feature candidate. Random threshold candidates can either be obtained by randomly sampling from a distribution or by sampling feature responses of training instances. We implement the latter approach, which allows for greater flexibility if features or image channels are changed. For every feature candidate generation, one thread on the GPU is used and all T thresholds for a given feature are sampled by the same thread.

In addition to sorting samples according to the image they belong to, feature candidates are sorted by the feature type, channels used, and region offsets. Sorting reduces branch divergence and improves spatial locality, thereby increasing the cache hit rate.

FEATURE RESPONSE CALCULATION On GPU, we implement a similar optimization technique as the one used on CPU, where loops in the feature generation step are rearranged in order to improve caching.

We used one thread to calculate the feature response for a given feature and a given training sample. Section 4.4 shows the thread block layout for the feature response calculation. A row of blocks calculates all feature responses for a given sample. A column of blocks calculates the feature responses for a given feature over all samples. The dotted red arrow indicates the order of thread block scheduling. The execution order of thread blocks

is determined by calculating the Block ID bid . In the two-dimensional case, it is defined as

$$bid = \text{blockIdx}.x + \underbrace{\text{gridDim}.x}_{\text{blocks in row}} \cdot \underbrace{\text{blockIdx}.y}_{\text{sample ID}}.$$

The number of features can exceed the maximum number of threads in a block, hence, the feature response calculation is split into several thread blocks. We use the x coordinate in the grid for the feature block to ensure that all features are evaluated before the GPU continues with the next sample. The y coordinate in the grid assigns training samples to thread blocks. Threads reconstruct their feature ID f using block size, thread and block ID by calculating

$$f = \text{threadIdx}.x + \underbrace{\text{blockDim}.x}_{\text{threads in block row}} \cdot \underbrace{\text{blockIdx}.x}_{\text{block index in grid row}}.$$

After sample data and feature parameters are loaded, the kernel calculates a single feature response for a depth or color feature by querying four pixels in an integral image and carrying out simple arithmetic operations to calculate the two regions sums and their difference.

HISTOGRAM AGGREGATION Feature responses are aggregated into class histograms. Counters for histograms are maintained in a four-dimensional matrix of size $F \times T \times C \times 2$ for F features, T thresholds, C classes, and the two left and right children of a split.

To compute histograms, the iteration over features and thresholds is implemented as thread blocks in a two-dimensional grid on GPU; one thread block per feature and threshold. This is depicted in Section 4.4. Each thread block slices samples into partitions such that all threads in the block can aggregate histogram counters in parallel.

Histogram counters for one feature and threshold are kept in the shared memory, and every thread gets a distinct region in the memory. For X threads and C classes, $2XC$ counters are allocated. An additional reduction phase is then required to reduce

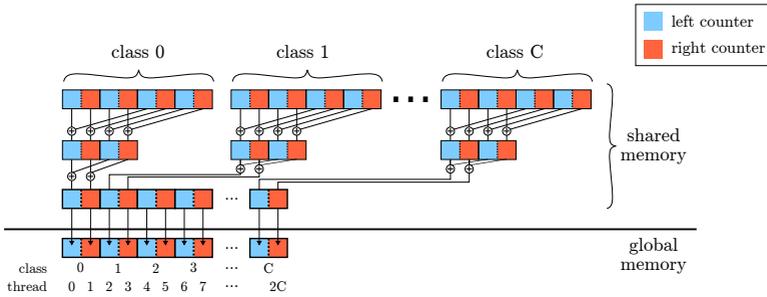


Figure 4.5: Reduction of histogram counters. Every thread sums to a dedicated left and right counter (indicated by different colors) for each class (first row). Counters are reduced in a subsequent phase. The last reduction step stores counters in shared memory, such that no bank conflicts occur when copying to global memory.

the counters to a final sum matrix of size $C \times 2$ for every feature and threshold.

Figure 4.5 shows histogram aggregation and sum reduction. Every thread increments a dedicated counter for each class in the first phase. In the next phase, we iterate over all C classes and reduce the counters of every thread in $O(\log X)$ steps, where X is the number of threads in a block. In a single step, every thread calculates the sum of two counters. The loop over all classes can be executed in parallel by $2C$ threads that copy the left and right counters of C classes.

The binary reduction of counters (Fig. 4.5) has a constant runtime overhead per class. The reduction of counters for classes without samples can be skipped, as all counters are zero in this case.

IMPURITY SCORE CALCULATION Computing impurity scores from the four-dimensional counter matrix is the last of the four training phases that are executed on GPU.

In the score kernel computation, 128 threads per block are used. A single thread computes the score for a different pair of features and thresholds. It loads $2C$ counters from the four-dimensional

counter matrix in global memory, calculates the impurity score and writes back the resulting score to global memory.

The calculated scores are stored in a $T \times F$ matrix for T thresholds and F features. The matrix is then finally transferred from device to host memory space.

UNDEFINED VALUES Image borders and missing depth values (e.g. due to material properties or camera disparity) are represented as not a number (NaN), which automatically propagates and causes comparisons to produce false. This is advantageous, since no further checks are required and the random forest automatically learns to deal with missing values.

4.4.2 Global Memory Limitations

SLICING OF SAMPLES Training arbitrarily large datasets with many samples can exceed the storage capacity of global memory. The feature response matrix of size $D \times F$ scales linearly in the number of samples D and the number of feature candidates F . We cannot keep the entire matrix in global memory if D or F is too large. For example, training a dataset with 500 images, 2000 samples per image, 2000 feature candidates and double precision feature responses (64 bit) would require $500 \cdot 2000 \cdot 2000 \cdot 64 \text{ bit} \approx 15 \text{ GB}$ of global memory for the feature response matrix in the root node split evaluation.

To overcome this limitation, we split samples into partitions, sequentially compute feature responses, and aggregate histograms for every partition. The maximum possible partition size depends on the available global memory of the GPU.

IMAGE CACHE Given a large dataset, we might not be able to keep all images in the GPU global memory. We implement an image cache with a last recently used (LRU) strategy that keeps a fixed number of images in memory. Slicing samples ensures that a partition does not require more images than can be fit into the cache.

MEMORY POOLING To avoid frequent memory allocations, we reuse memory that is already allocated but no longer in use. Due to the structure of random decision trees, evaluation of the root node split criterion is guaranteed to require the largest amount of memory, since child nodes always contain less or equal samples than the root node. Therefore, all data structures have at most the size of the structures used for calculating the root node split. With this knowledge, we are able to train a tree with no memory reallocation.

4.4.3 *Extensions*

HYPER-PARAMETER OPTIMIZATION Cross-validating all the hyper-parameters is a requirement for model comparison, and random forests have quite a few hyper-parameters, such as stopping criteria for splitting, number of features and thresholds generated, and the feature distribution parameters.

To facilitate model comparison, CURFIL includes support for cross-validation and a client for an informed search of the best parameter setting using Hyperopt (Bergstra et al., 2011). This allows to leverage the improved training speed to run many experiments serially and in parallel.

IMAGE FLIPPING To avoid overfitting, the dataset can be augmented using transformations of the training dataset. One possibility is to add horizontally flipped images, since most tasks are invariant to this transformation. CURFIL supports training horizontally flipped images with reduced overhead.

Instead of augmenting the dataset with flipped images and doubling the number of pixels used for training, we horizontally flip each of the two rectangular regions used as features for a sampled pixel. This is equivalent to computing the feature response of the same feature for the same pixel on an actual flipped image. Histogram counters are then incremented following the binary test of both feature responses. The implicit assumption here is that the samples generated through flipping are independent.

The paired sample is propagated down a tree until the outcome of a node binary test is different for the two feature responses,

Table 4.1: Comparison of random forest *training* time (in minutes) on a quadcore CPU and two non-mobile GPUs. Random forest parameters were chosen for best accuracy.

Device	NYUD		MSRC-21	
	time	factor	time	factor
i7-4770K	369	1.0	93.2	1.0
Tesla K20c	55	6.7	5.1	18.4
GTX Titan	24	15.4	3.4	25.9

indicating that a sample and its flipped counterpart should split into different directions. A copy of the sample is then created and added to the samples list of the other node child.

This technique reduces training time since choosing independent samples from actually flipped images requires loading more images in memory during the best split evaluation step. Since our performance is largely bounded by memory throughput, dependent sampling allows for higher throughput at no cost in accuracy.

4.5 EXPERIMENTAL RESULTS

We evaluate our library on two common object class segmentation tasks, the NYUD and the MSRC-21 dataset. We focus on the processing speed, but also discuss the prediction accuracies attained. Note that the speed between datasets is not comparable, since dataset sizes differ and the forest parameters were chosen separately for best accuracy.

4.5.1 Datasets

The NYUD by Silberman et al. (2012) contains 1449 densely labeled pairs of aligned RGB-D images from 464 indoor scenes. We focus on the semantic classes ground, furniture, structure, and props defined by Silberman et al.

Table 4.2: Random forest *prediction* time on RGB-D images at original resolution, comparing speed on a recent quadcore CPU and various GPUs. Random forest parameters are chosen for best accuracy.

Device	NYUD		MSRC-21	
	time (ms)	factor	time (ms)	factor
i7-440K	477	1	409	1
GTX 675M	28	17	37	11
Tesla K20c	14	34	10	41
GTX Titan	12	39	9	48

To evaluate our performance without depth, we use the MSRC-21 dataset³. Here, we follow the literature in treating rarely occurring classes horse and mountain as void and train/predict the remaining 21 classes on the standard split of 335 training and 256 test images.

4.5.2 Training and Prediction Time

Tables 4.1 and 4.2 show random forest training and prediction times, respectively, on an Intel Core i7-4770K (3.9 GHz) quadcore CPU and various Nvidia GPUs. Note that the CPU version is using all cores.

For the RGB-D dataset, training speed is improved from 369 min to 24 min, which amounts to a speed-up factor of 15. Dense prediction improves by factor of 39 from 477 ms to 12 ms.

Training on the RGB dataset is finished after 3.4 min on a GTX Titan, which is 26 times faster than CPU (93 min). For prediction, we achieve a speed-up of 48 on the same device (9 ms vs. 409 ms).

Prediction is fast enough to run in real time even on a mobile GPU (GTX 675M, on a laptop computer fitted with a quadcore i7-3610QM CPU), with 28 ms (RGB-D) and 37 ms (RGB).

³ <http://jamie.shotton.org/work/data.html>

Table 4.3: Segmentation accuracies on NYUD of our random forest compared to state-of-the-art methods. We used the same forest as in the training/prediction time comparisons of Tables 4.1 and 4.2.

Method	Accuracy (%)	
	Pixel	Class
Silberman et al. (2012)	59.6	58.6
Coupric et al. (2013)	63.5	64.5
Our random forest*	68.1	65.1
Our random forest* (with height, cf. Section 4.5.4)	69.6	66.5
Stückler et al. (2014)**	70.6	66.8
Hermans et al. (2014)	68.1	69.0
Müller and Behnke (2014)**	72.3	71.9

* see main text for hyper-parameters used

** based on our random forest prediction

4.5.3 Classification Accuracy

Our implementation is fast enough to train hundreds of random decision trees per day on a single GPU. This fast training enabled us to conduct an extensive parameter search with five-fold cross-validation to optimize segmentation accuracy of a random forest trained on the NYUD (Silberman et al., 2012). Table 4.3 shows that we outperform other state-of-the-art methods simply by using a random forest with optimized parameters. The resulting model and the fast CURFIL prediction were used in two publications which improved the results further by 3D accumulation of predictions in real time (Stückler et al., 2014) and superpixel CRFs (Müller and Behnke, 2014). This shows that efficient hyper-parameter search is crucial for model selection. Example segmentations are displayed in Figs. 4.6 and 4.7.

Methods on the established RGB-only MSRC-21 benchmark are so advanced that their accuracy cannot simply be improved by a random forest with better hyper-parameters. Our pixel and class

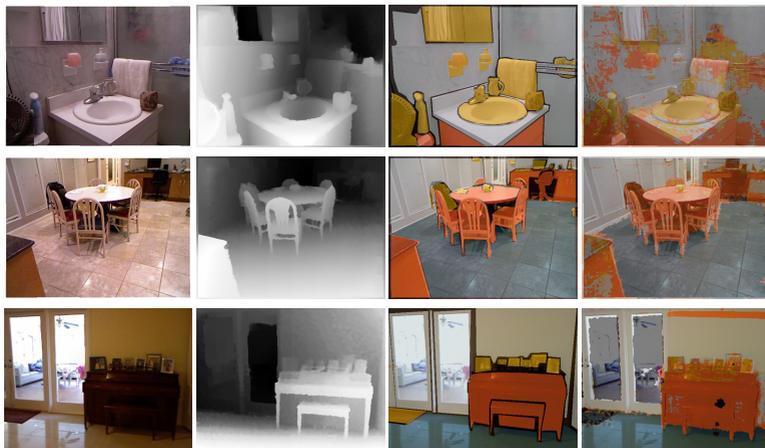


Figure 4.6: Segmentation examples on NYUD. Left to right: RGB image, depth visualization, ground truth, random forest segmentation.

accuracies for MSRC-21 are 59.2% and 47.0%, respectively. This is still higher than other published work using random forests as the baseline method, such as 49.7% and 34.5% by Shotton et al. (2008). However, as Shotton et al. and the above works show, random forest predictions are fast and constitute a good initialization for other methods such as conditional random fields.

Finally, we trained the MSRC-21 dataset by augmenting the dataset with horizontally flipped images using the naïve approach and our proposed method. The naïve approach doubles both the total number of samples and the number of images, which quadruples the training time to 14.4 min. Accuracy increases to 60.6% and 48.6% for pixel and class accuracy, respectively. With paired samples (introduced in Section 4.4.3), we reduce the runtime by a factor of two (to now 7.48 min) at no cost in accuracy (60.9% and 49.0%). The remaining difference in speed is mainly explained by the increased number of samples, thus the training on flipped images has very little overhead.

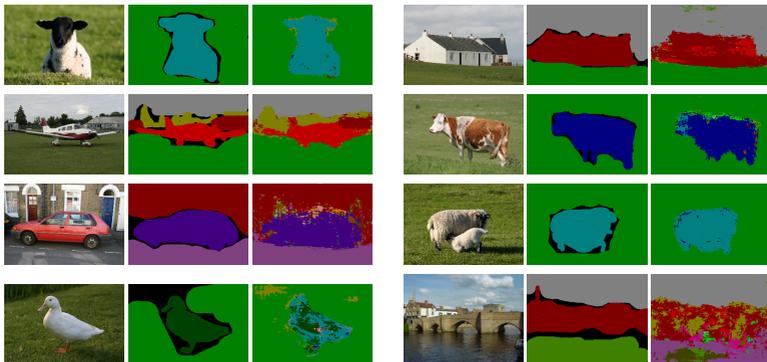


Figure 4.7: Segmentation examples on the MSRC-21 dataset. In groups of three: input image, ground truth, random forest segmentation. Last row shows typical failure cases

4.5.4 Incorporating Novel Features

With few changes in code, CURFIL allows to incorporate novel features. To demonstrate this, we chose height above ground, which is an important cue for indoor scene classification, and has been used in multiple other studies (Gupta et al., 2014; Müller and Behnke, 2014; Schulz et al., 2015a). On a robot with known camera pose, height above ground can be inferred directly. To generate this information for the NYUD—where camera poses are not available—we proceed as suggested by Müller and Behnke (2014). We extract normals in the depth images, find ten clusters in normal space with k -means and determine the cluster that is most vertical. We then project all points to this normal and subtract the height of the lowest point.

We add the height image as an additional depth channel. Instead of computing region differences as in Eq. (4.1), we determine the average height above ground in R_1 , such that

$$f_{\text{height},\theta}(\mathbf{q}) := \frac{1}{|R_1(\mathbf{q})|} \sum_{\mathbf{p} \in R_1} \chi_{\text{height}}(\mathbf{p}). \quad (4.2)$$

Using the same hyperparameters as without height, the classification accuracy improves significantly by 1.5 and 1.3 percentage points for class and pixel accuracy, respectively (Table 4.3). Analysis of the learned forest shows that overall, height above ground is used in roughly 12%, depth differences in 38%, and color in 50% of the split nodes. These numbers reflect the statistics of the feature proposal distribution.

4.5.5 *Random Forest Parameters*

The hyper-parameter configurations for which we report our timing and accuracy results were found with global parameter search and cross-validation on the training set. The cross-validation outcome varies between datasets.

For the NYUD, we used three trees with 4537 samples / image, 5729 feature candidates / node, 20 threshold candidates, a box radius of 111 px, a region size of 3, tree depth 18 levels, and minimum samples in leaf nodes 204.

For MSRC-21 we found 10 trees, 4527 samples / image, 500 feature candidates / node, 20 threshold candidates, a box radius of 95 px, a region size of 12, tree depth 25 levels, and minimum samples in leaf nodes 38 to yield best results.

4.6 CONCLUSION

We provide an accelerated random forest implementation for image labeling research and applications. Our implementation achieves real-time dense pixel-wise classification of VGA images on a GPU. Training is accelerated on GPU by a factor of up to 26 compared to an optimized CPU version. The experimental results show that our fast implementation enables effective parameter searches that find solutions which outperform state-of-the-art methods. CURFIL prepares the ground for scientific progress with random forests, e. g. through research on improved visual features.

UNSUPERVISED METHODS FOR IMAGE CATEGORIZATION

In this chapter, we turn to one of the main deep learning methods introduced in Chapter 2, the combination of unsupervised pre-training and supervised fine-tuning, specifically the “simple” building blocks used in the unsupervised pre-training: RBMs and auto-encoders. Section 5.1 discusses how we can endow RBMs with local receptive fields. In Section 5.2, we empirically investigate the limits of RBM learning. Finally, in Section 5.3 we show that auto-encoders are limited in the type of features they can encode and propose a solution using two-layer encoders with shortcut connections.

5.1 EXPLOITING LOCAL STRUCTURE IN BOLTZMANN MACHINES

One of the main aims of unsupervised learning is modeling the data distribution. Generative graphical models, such as Restricted Boltzmann Machines (RBM, Hinton et al. 2006) are a popular choice for this purpose. RBMs model correlations of observed variables by introducing binary latent variables (features) which are assumed to be conditionally independent given the observed variables (Eq. (2.6)). This restriction is useful because, in contrast to general Boltzmann Machines, a fast learning algorithm exists (Contrastive Divergence, *ibid.*). RBMs are generic learning machines and have been applied to many domains, including text, speech, motion data, and images. In the most commonly used form, however, they do not take advantage of the topology of the input space. Especially when applied to image data, fully connected RBMs model long-range dependencies which are known to be weak in natural images (Huang and Mumford, 1999).

One way to deal with this problem is to completely remove long-range parameters from the model. The advantage of this approach is two-fold: first, local connectivity serves as a prior that

matches well to the properties of natural images and, second, the drastically reduced number of parameters makes learning in larger models feasible. The downside of local connectivity is that weaker long-distance interactions cannot be modeled at all. In this chapter, we propose to compensate for this disadvantage by introducing direct or indirect lateral interactions between the local features.

While local receptive fields are well-established in discriminative learning, their counterpart in the generative case, which we call “impact area”, is not well understood. In this chapter, we investigate the capabilities of stacked RBMs and RBM-like graphical models with local impact area and lateral connections. We train our architecture on the well-known MNIST database of handwritten digits (LeCun et al., 1998a) and demonstrate the efficiency of learning. The hidden representations can then be used for classification. With a similar number of model parameters, we find that models which exploit image structure perform better for classification. We also show that models with local impact area can generate globally consistent images. Finally, the data probability under our model compares favorably with the data probability of a fully connected RBM.

5.1.1 Background on Boltzmann Machines

A Boltzmann Machine (BM) is an undirected graphical model with binary observed variables $\mathbf{v} \in \{0, 1\}^n$ (visible nodes) and latent variables $\mathbf{h} \in \{0, 1\}^m$ (hidden nodes). The energy function of a BM is given by

$$E(\mathbf{v}, \mathbf{h}, \theta) = -\mathbf{v}^T W \mathbf{h} - \mathbf{v}^T I \mathbf{v} - \mathbf{h}^T L \mathbf{h} - \mathbf{b}^T \mathbf{v} - \mathbf{a}^T \mathbf{h},$$

where $\theta = (W, I, L, \mathbf{b}, \mathbf{a})$ are the model parameters, namely pairwise visible-hidden, visible-visible and hidden-hidden interaction weights, respectively, and \mathbf{b} , \mathbf{a} are the biases of visible and hidden activation potentials. The diagonal elements of I and L are always zero. This yields a probability distribution $p(v)$

$$p(\mathbf{v}; \theta) = \frac{1}{Z(\theta)} p^*(\mathbf{v}; \theta) = \frac{1}{Z(\theta)} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h}, \theta)},$$

where $Z(\theta)$ is the normalizing constant (partition function) and $p^*(\cdot)$ denotes unnormalized probability.

In restricted BMs (RBMs), I and L are set to zero. Consequently, the conditional distributions $p(\mathbf{v}|\mathbf{h})$ and $p(\mathbf{h}|\mathbf{v})$ factorize completely. This makes exact inference of the respective posteriors possible. Their expected values are given by

$$\langle \mathbf{v} \rangle_p = \sigma(W\mathbf{h} + \mathbf{b}) \quad \text{and} \quad \langle \mathbf{h} \rangle_p = \sigma(W\mathbf{v} + \mathbf{b}), \quad (5.1)$$

where σ denotes element-wise application of the sigmoid function. In practice, Contrastive Divergence (CD, Hinton et al. 2006) is used to approximate the true parameter gradient

$$\frac{\partial \ln p(\mathbf{v})}{\partial w_{i,j}} = \langle \mathbf{v}^T \mathbf{h} \rangle_+ - \langle \mathbf{v}^T \mathbf{h} \rangle_- \quad (5.2)$$

by a Markov chain Monte Carlo (MCMC) algorithm. Here, $\langle \cdot \rangle_+$ and $\langle \cdot \rangle_-$ refer to the expected values with respect to the data distribution and model distribution, respectively. Approximating these quantities is called the positive and negative phase. Tieleman (2008) proposed a faster alternative, called Persistent Contrastive Divergence (PCD), which employs a persistent Markov chain to approximate $\langle \cdot \rangle_-$. We use PCD throughout this chapter.

When I is not zero, the model is called a semi-restricted Boltzmann machine (SRBM) (Osindero and Hinton 2008). This model can be trained with a variant of CD by approximating $p(\mathbf{v}|\mathbf{h})$ using a few damped mean-field updates instead of many sequential rounds of Gibbs sampling. We will refer to this model as SRBM^- , as the lateral connections only play a role in the negative phase. We will later introduce our model, the SRBM^+ , where lateral connections influence both the positive and the negative phase.

As described in Section 2.4, RBMs can be stacked to build hierarchical models. The training of stacked models proceeds layer-wise by training the high-level models using the activations of the hidden nodes of the layer below as input.

5.1.2 Local Impact Semi-Restricted Boltzmann Machines⁺

We now introduce two modifications to the architectures introduced in Section 5.1.1. Firstly, we restrict the impact area of each

hidden node. To this end, we arrange the hidden nodes in multiple grids, each of which resembles the visible layer in its topology. As a result, each hidden node h_j can be assigned a position $x(h_j) \in \mathbb{N}^2$ in the input (pixel) coordinate system. This approach is similar to the common approach in convolutional neural networks (LeCun et al., 1998a). We then allow w_{ij} to be non-zero only if $|\text{pos}(v_i) - \text{pos}(h_j)| < r$ for a small constant r , where $\text{pos}(v_i)$ is the pixel coordinate of v_i . In contrast to the convolutional procedure, we do not require the weights to be equal for all hidden units within one grid. We call this modification of the RBM the Local Impact RBM or, shorter, LIRBM.

Secondly, we allow l_{ij} to be non-zero if $i \neq j$. Learning in this model proceeds as follows. In the positive phase the visible activations are given by the input and the hidden activations are calculated using damped mean-field updates. The mean-field updates are determined by

$$\mathbf{h}^{(0)} = \sigma(W^T \mathbf{v} + \mathbf{b}), \quad \mathbf{h}^{(k+1)} = \sigma(W^T \mathbf{v} + \mathbf{b} + L^T \mathbf{h}^{(k)}). \quad (5.3)$$

In the negative phase, we continue our persistent Markov chain from the current state $(\mathbf{v}_p^0, \mathbf{h}_p^0)$ by sampling \mathbf{v}_p^1 from

$$p(\mathbf{v}_p^0 | \mathbf{h}_p^0) = \sigma(W \mathbf{h}_p^0 + \mathbf{a}). \quad (5.4)$$

We then generate a new hidden state by sampling from

$$p(\mathbf{h}^{k+1} | \mathbf{v}_p^1) = \sigma(W^T \mathbf{v}_p^1 + \mathbf{b} + L^T \mathbf{h}_p^k). \quad (5.5)$$

We call this model SRBM⁺ since, in contrast to SRBM⁻, the lateral connections play a crucial role in the positive phase.

A common problem in training convolutional RBMs is that the overcomplete representation of the visible nodes by the hidden nodes enables the filters to learn a trivial identity (Lee et al., 2009; Norouzi et al., 2009). The proposed SRBM⁺ does not suffer from this problem for two reasons: First, we use PCD for training. This means even if a training example can be perfectly reconstructed, there is still a non-zero learning signal. This signal stems from the dependency of the approximated gradient on the state of the persistent Markov chain. Second, due to not sharing weights, for a trivial solution to occur, all filters have to learn the identity separately.

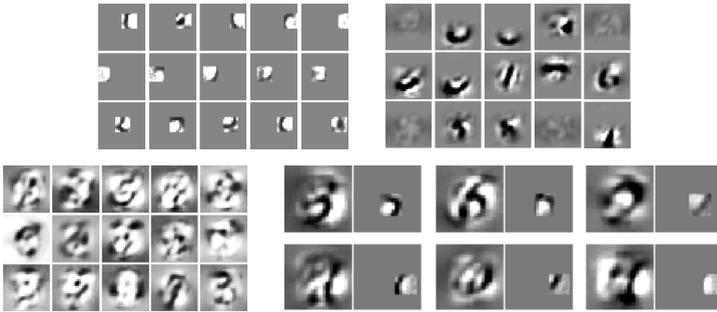


Figure 5.1: Visualization of direct and indirect lateral interaction. Activations of randomly selected hidden nodes in LIRBM, projected down from the first, second and third layer, respectively. *Bottom right*: Visualizations of lateral interactions in LIRBM⁺, see text in Section 5.1.4 for details.

5.1.3 Related Work

In the context of generative models of images, little work has been done to exploit local structure. A well known supervised learning approach that makes use of the local structure of images is the convolutional neural network by LeCun et al. (1998a). Another approach to exploit local structure has been suggested by Behnke (2003b). LeCun’s ideas were transferred to the field of generative graphical models by Lee et al. (2009) and Norouzi et al. (2009). Their models, which employ weight sharing and max-pooling, discard global image statistics. Our model does not suffer from this restriction. For example, when training landscapes, our model would be able to learn, even on the lowest layer, that there is always sky depicted in the upper half of the image.

To achieve globally consistent representations in spite of the local impact area, we make use of lateral connections between the latent variables. Such connections can be modelled indirectly using stacked RBMs as in Deep Belief Networks (DBN) (Hinton et al., 2006) or Deep Boltzmann Machines (DBM, Salakhutdinov and Hinton 2009). Stacks of more than two RBMs, however, are not guaranteed to improve the data likelihood. In fact, even stacks of two RBMs do not improve a lower likelihood bound empirically

Table 5.1: Test log-likelihood on MNIST for plain RBMs (RBM-51, RBM-392) and Local Impact RBMs with different impact area sizes, measured in nats. Local impact RBMs perform better than standard RBM with the same number of parameters.

	Global connectivity		Local connectivity	
	RBM-51	RBM-392	LIRBM (11×11)	LIRBM (7×7)
Impact area	n/a	n/a		
Log-probability	-125.58	-101.69	-108.65	-109.95
# Parameters	40 819	308 504	39 818	27 058

(Salakhutdinov, 2009a). On the other hand, stacking locally connected RBMs yields larger effective impact area for higher layers and thus can enforce more global constraints.

A more direct way to model lateral interaction is to introduce pairwise potentials between observed or between latent variables. The former case, without restrictions to local impact areas, was studied by (Osindero and Hinton, 2008). Furthermore, (Salakhutdinov, 2008) trained general BMs with lateral interactions between both observed and latent variables on small problems. Due to long training times, this general approach seems to be infeasible for larger problem sizes. Furthermore, stacking of BMs would yield two kinds of lateral interactions in each layer.

In this work, we employ lateral interactions to implicitly extend the impact area of latent variables.

5.1.4 *Experimental Results*

In our experiments we analyze the effects of local impact areas and lateral interaction terms on RBMs. As a first step, we trained models with and without local impact areas on the MNIST database of handwritten digits (LeCun et al., 1998a).

The standard RBM model had 51 hidden units, so that it had slightly higher number of parameters than the locally connected model with a hidden layer consisting of two grids of size 14×14 ,

and an impact area of size 12×12 . Throughout the chapter, we use 500 epochs for fully connected models and 80 epochs for locally connected models. Please note that the reduced number of epochs is not due to longer training times of locally connected models. On the contrary, fewer parameters have to be updated and the smaller parameter space results in faster convergence.

We then approximated the likelihood assigned to the test data under these models using annealed importance sampling (AIS). The likelihood is measured in “nats”, meaning the natural logarithm of the unit-less probabilities. The results are summarized in Table 5.1. The locally connected model had a test log-likelihood of -108.65 nats whereas the standard RBM had a test log-likelihood of -125.58 nats, showing a clear advantage of our model. For comparison, a fully connected RBM with as many hidden units as the local model achieves a test log-likelihood of -101.69 nats. However, this model has eight times more parameters. Further decreasing the number of parameters by reducing the size of the impact area by factor of 0.65 does not significantly reduce the test log-likelihood.

In a second experiment, we examine the suitability of the hidden representations, which were learned without supervision, for classification. For fair comparison, we employ a k -nearest neighbor classifier with $k = 3$. We find that features learned with LIRBM are more useful (3.12%) for classification than features learned by a plain RBM (5.24%) with a similar number of parameters. As above, we also trained an RBM with 392 hidden units (2.65%) for comparison. Furthermore, we observe that our LIRBM⁺ model yields slightly better (although insignificantly so) results (3.07%) when compared to LIRBM. Surprisingly, our LIRBM⁻ performs much worse than all other models. It could be that the whitening operation implicitly performed by the SRBM⁻ (Osinero and Hinton, 2008) hurts classification performance while it is helpful for generative purposes.

Next, we evaluated the influence of direct and indirect lateral interactions in the generative context. To this end, we visualized filters and fantasies generated by LIRBM and LIRBM⁺. The left three images in Fig. 5.1 show weights in a LIRBM with three layers. In the first image, filters of the lowest hidden layer are projected

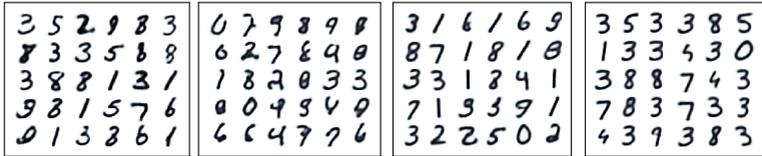


Figure 5.2: Fantasies generated by our models from random noise. Left: LIRBM⁺, one layer, 1000 steps in Markov chain. Center left: LIRBM, one layer, 1000 steps. Center right: LIRBM, two layers, 500 steps. Right: LIRBM, three layers, 250 steps. Lateral connections as well as stacking enforce global consistency.

down to the visible layer. We observe that small parts of lines and line-endings are learned. The second and third figure display filters from the second and third hidden layer, projected down to the visible layer. These filters are clearly more global. With their size, the size of the recognized structure increases as well. The fourth image visualizes lateral interactions in an LIRBM⁺. Even columns show randomly selected filters h_j of the first hidden layer, while the patch to the left of a filter depicts a linear combination of all other filters h_i , weighted by their pairwise potential l_{ij} . Note that h_j does not contribute to this sum, since $l_{jj} = 0$. We observe that through lateral interaction the filter is not only replicated, it is even extended beyond its impact area to a more global feature.

Figure 5.2 shows fantasies generated by LIRBM⁺ and a stack of LIRBM. Markov chains for fantasies were started using binary noise in the visible layer. To show quick convergence to model distribution, less iterations were used on the deeper models. It is clear that fantasies produced by a single-layer LIRBM are only locally consistent and one can observe that stacking gradually improves global consistency. Lateral connections in the hidden layer significantly improve consistency even for a flat model.

These finds strongly support our initial claim that lateral interaction compensates for the negative effects of the local impact areas.

5.1.5 Conclusions

In this chapter, we present a novel variation of the Restricted Boltzmann Machine for image data, featuring only local interactions between visible and hidden nodes. While learning in this model is fast and few parameters yield comparably good data probabilities and classification performance, the model does not enforce global consistency constraints. We showed that this effect can be compensated for by adding lateral interactions between the latent variables, which we model directly by pairwise potentials or indirectly through stacking.

Due to the small number of parameters and its computational efficiency our architecture has the potential to model images of a much larger size than commonly used forms of RBMs.

5.2 INVESTIGATING CONVERGENCE OF RBM LEARNING

Restricted Boltzmann Machines (RBMs, Smolensky, 1986) have been widely used as generative models, for unsupervised feature extraction and as building blocks of deep belief networks (Bengio, 2009; Salakhutdinov and Hinton, 2009). Applications range from image processing (Ranzato and Hinton, 2010) and classification (Hinton et al., 2006) to collaborative filtering (Salakhutdinov et al., 2007). Despite this success RBM training remains a problematic task. For even medium-sized RBMs likelihood maximization is not possible because the true gradient of the likelihood is not tractable.

Most applications instead rely on a fast MCMC approximation to the gradient, called contrastive divergence (CD). CD was shown to work well in practice in a number of tasks, even though it is not a good approximation to the likelihood gradient (Salakhutdinov and Murray, 2008).

There are a number of variants of CD, notably persistent contrastive divergence (PCD), fast persistent contrastive divergence (FPCD), Tempered Transitions (Salakhutdinov, 2009b), and Parallel Tempering (Desjardins et al., 2010; Cho et al., 2010). Most of these come with a variety of hyperparameters in addition to the established

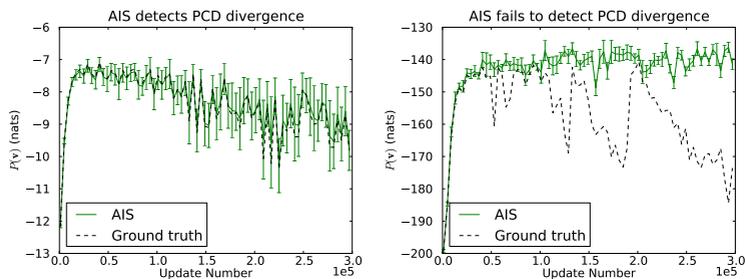


Figure 5.3: AIS (solid green with average uncertainty) vs. ground truth (black, dashed). *Left.* AIS follows true likelihood even during unstable learning. *Right.* AIS fails to detect PCD learning divergence.

heuristics of weight-decay, momentum, and learnrate schedules. Since exact evaluation of the objective function is infeasible for interesting datasets, it is not clear which heuristic to choose and how to set the hyperparameters. Empirical evaluations exist (Desjardins et al., 2010; Tieleman and Hinton, 2009) but are scarce on realistic datasets.

Fischer and Igel (2010), Desjardins et al. (2010) and others even observed that CD (and PCD) training diverges on the training set after an initial increase in likelihood. Fischer and Igel (2010) conclude that the right choice of hyperparameters can solve this problem. Due to the problem of calculating the partition function, however, it is not clear how the best training method and hyperparameters could be chosen.

Please also note that the effects discussed in this section are not related to the common “overfitting” phenomenon. This can be dealt with by choosing a validation set and monitoring the ratio of the unnormalized probabilities, causing the partition functions to cancel. What we are looking at in this section is whether the training method actually follows the gradient on the training set, as opposed to generalization on a validation set.

To our knowledge, there are two main methods in use today to evaluate the learning progress of RBMs. One is the so-called “reconstruction error”, the other is annealed importance sampling

(AIS). In the *practical guide to training RBMs*, Hinton (2012) both refers researchers to the reconstruction error but also warns them to rely on it. The “reconstruction error” is the difference between a data point and the “reconstruction”,

$$\ell_{\text{recons}}(\mathcal{D}, \theta) = \sum_{\mathbf{x} \in \mathcal{D}} \left\| \mathbf{x} - E[\mathbf{v} \mid E[\mathbf{h}|\mathbf{v}; \theta]; \theta] \right\|_2^2. \quad (5.6)$$

Fischer and Igel (2010) found that this measure is truly dangerous on some toy problems, since it does not correlate with the objective function of RBM training and in particular does not detect the divergence of likelihood. In this section, we can confirm this observation for more realistically sized RBMs.

The second commonly used method to evaluate RBMs is Annealed Importance Sampling (AIS, Neal, 2001; Salakhutdinov and Murray, 2008). AIS is an MCMC method that can be used to approximate the partition function of an RBM with the help of a baseline model. We investigate the use of AIS, not only to judge the final result of learning but also to find good hyperparameters and as an indicator when to stop learning to prevent divergence.

The main observations in this section can be summarized as follows: By analyzing detailed learning curves on medium-sized RBMs, we find that using PCD and a simple update rule suffices to produce high likelihood values. In particular, it is not necessary to tune many hyperparameters or to find the right learning schedule. Prevention of divergence remains difficult, however, since results on AIS approximations are rather mixed. In most cases, the behaviour of the true likelihood was reproduced accurately but in other cases serious divergence was not detected at all. Theoretical work by Yuille (2005) shows that CD is guaranteed to converge to a local maximum when an appropriate learnrate schedule is used. Whether this can be used in practice is not clear, as too conservative learnrate schedules result in convergence to low likelihood values ((Fischer and Igel, 2010) and our results in Section 5.2.3). Long and Servedio (2010) showed that it is NP-hard to approximate the likelihood of a given RBM to a certain precision (ibid.). We therefore suggest more research in the direction of early stopping and detection of divergence.

5.2.1 Background on Restricted Boltzmann Machines

A Restricted Boltzmann Machine (RBM) is an undirected graphical model with binary observed variables $\mathbf{v} \in \{0, 1\}^n$ (visible nodes) and binary latent variables $\mathbf{h} \in \{0, 1\}^m$ (hidden nodes). The energy function of an RBM is given by

$$E(\mathbf{v}, \mathbf{h}, \theta) = -\mathbf{v}^T W \mathbf{h} - \mathbf{b}^T \mathbf{v} - \mathbf{a}^T \mathbf{h}, \quad (5.7)$$

where $\theta = (W, \mathbf{b}, \mathbf{a})$ are the model parameters, namely pairwise visible-hidden interaction weights and biases of visible and hidden activation potentials, respectively. This yields a probability distribution

$$p(\mathbf{v}; \theta) = \frac{1}{Z(\theta)} p^*(\mathbf{v}; \theta) = \frac{1}{Z(\theta)} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h}, \theta)}, \quad (5.8)$$

where $Z(\theta)$ is the normalizing constant (partition function) and $p^*(\cdot)$ denotes unnormalized probability. The conditional distributions

$p(\mathbf{v}|\mathbf{h})$ and $p(\mathbf{h}|\mathbf{v})$ factorize completely, making exact inference of the respective posteriors possible. Their expected values are given by

$$\langle \mathbf{v} \rangle_p = \sigma(W \mathbf{h} + \mathbf{b}) \quad \text{and} \quad \langle \mathbf{h} \rangle_p = \sigma(W \mathbf{v} + \mathbf{a}). \quad (5.9)$$

Here, σ denotes element-wise application of the logistic sigmoid:

$$\sigma(x) = (1 + \exp(-x))^{-1}. \quad (5.10)$$

In practice, contrastive divergence (CD) or one of its variants is used to approximate the true parameter gradient

$$\frac{\partial \ln p(\mathbf{v})}{\partial W} = \langle \mathbf{v} \mathbf{h}^T \rangle_+ - \langle \mathbf{v} \mathbf{h}^T \rangle_- \quad (5.11)$$

by an MCMC algorithm. Here, $\langle \cdot \rangle_+$ and $\langle \cdot \rangle_-$ refer to the expected values with respect to the data distribution and model distribution, respectively. The expected value of the data distribution is approximated in the “positive phase”, while the expected values of the model distribution are approximated in the “negative

phase". For CD in RBMs, $\langle \cdot \rangle_+$ can be calculated in closed form, while $\langle \cdot \rangle_-$ is estimated using k steps of a Markov chain started at the training data.

Recently, Tieleman (2008) proposed a faster alternative to CD, called Persistent Contrastive Divergence (PCD), which employs a persistent Markov chain to approximate $\langle \cdot \rangle_-$. This is done by maintaining a set of "fantasy particles" $\mathbf{v}_-, \mathbf{h}_-$ during the whole training. The chains are also governed by the transition operator in Eq. (5.9) and are used to calculate $\langle \mathbf{v}\mathbf{h}^T \rangle_-$ as the expected value with respect to the Markov chains $\langle \mathbf{v}_- \mathbf{h}_-^T \rangle$.

If the learnrate is small enough, the chains \mathbf{v}_- and \mathbf{h}_- should mix faster than the model changes. Therefore they form a better estimate of the model distribution than a k -step Gibbs sampling as performed by CD. As a side effect, PCD removes k from the set of hyperparameters to be adjusted and again emphasizes the importance of the learnrate.

RBMs can be stacked to build hierarchical models. The training of stacked models proceeds greedily layer-wise. After training an RBM, one calculates the expected values $\langle \mathbf{h} \rangle_{p(\mathbf{h}|\mathbf{v})}$ of its hidden variables given the training data. Keeping the parameters of the first RBM fixed, we can then train another RBM using $\langle \mathbf{h} \rangle_{p(\mathbf{h}|\mathbf{v})}$ as its input. We do not directly investigate stacking but concentrate on the learning of a single layer, as the results can be directly applied to the stacked setting.

Annealed importance sampling (AIS) can be used to obtain an approximation of the partition function of an RBM. It is an algorithm to estimate the ratio of two normalization constants, and builds upon the following fact. Let $p_A(v) = p_A^*(v)/Z_A$ and $p_B(v) = p_B^*(v)/Z_B$ be two distributions such that $p_A(v) \neq 0$ if $p_B(v) \neq 0$. Then:

$$\frac{Z_B}{Z_A} = \frac{\int p_B^*(\mathbf{v})d\mathbf{v}}{Z_A} = \int \frac{p_B^*(\mathbf{v})}{p_A^*(\mathbf{v})} p_A(\mathbf{v})d\mathbf{v} = \left\langle \frac{p_B^*(\mathbf{v})}{p_A^*(\mathbf{v})} \right\rangle_{p_A} \quad (5.12)$$

If it is possible to draw independent samples from p_A , then this expected value can be approximated using a Monte Carlo approach. This only gives good approximations if p_A is very close to p_B .

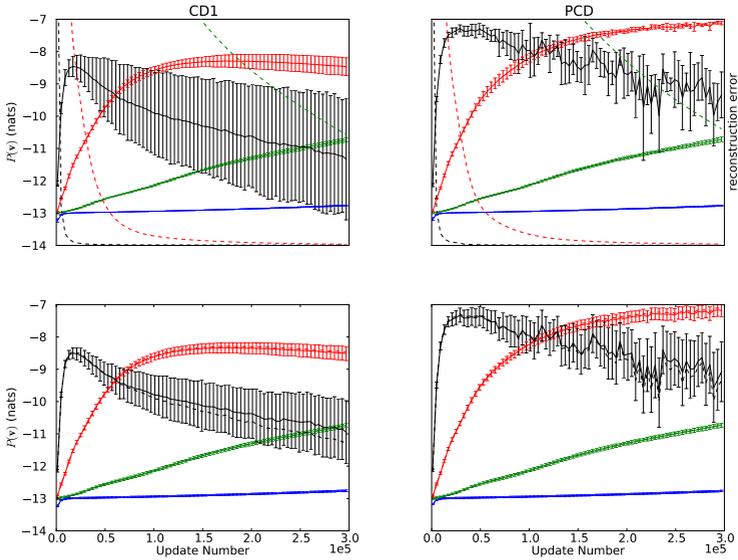


Figure 5.4: Learning curves for Shifter dataset: *Top row.* Exact likelihood during training, with standard deviation for different random initializations. Dashed lines show “reconstruction error”. *Bottom row.* AIS approximations to likelihood with error bars showing mean uncertainty. Colors indicate different learnrates η : black $\eta = 0.1$, red $\eta = 0.01$, green $\eta = 0.001$, blue $\eta = 0.0001$. See main text for a detailed discussion.

AIS overcomes this weakness by introducing an annealing chain of distributions p_n such that $p_0 = p_A$, $p_N = p_B$ and p_k is very close to p_{k+1} . Calculating the ratios of all intermediate normalization functions—which can be done efficiently using an MCMC algorithm—then yields the desired ratio Z_B/Z_A .

As detailed by Salakhutdinov and Murray (2008), AIS can be applied to calculating the partition function of an RBM by setting p_B to a distribution for which the normalization constant can be computed efficiently and setting p_A to the distribution modeled by the RBM.

5.2.2 *Experimental Setup*

We use three datasets in this section, which we chose for comparability with the literature.

SHIFTER. Labeled Shifter Ensemble (Fischer and Igel, 2010) is a 19-dimensional data set containing 768 samples. The samples are generated in the following way: The states of the first eight visible units are set uniformly at random. The states of the following eight units are cyclically shifted copies of the first eight. The shift can be zero, one unit to the left, or one to the right and is indicated by the last three units. Average log-likelihood is $\log \frac{1}{768} \approx -6.64$ if the distribution of the data set is modeled perfectly.

BARS AND STRIPES. This dataset also stems from Fischer and Igel (*ibid.*) and has 16 visible units. Each pattern corresponds to a square of 4×4 units and is generated by first randomly choosing an orientation, vertical or horizontal with equal probability, and then picking the state for all units of every row or column uniformly at random. Since each of the two completely uniform patterns can be generated in two ways, the upper bound of the average log-likelihood is -3.21 .

MNIST. Finally, we use the MNIST database of handwritten digits (MNIST)¹, which is more realistic dataset than the first two. If not by itself, it certainly has gained relevance through heavy use for evaluation of new learning algorithms.

¹ <http://yann.lecun.com/exdb/mnist/>

DETERMINING THE LOG-LIKELIHOOD. Due to the symmetric structure of the RBM energy function with respect to \mathbf{v} and \mathbf{h} , the likelihood can be factored in two different ways:

$$\begin{aligned} \log p(\mathbf{v}; \theta) &= -Z(\theta) + \log \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h}; \theta)) \\ &= -Z(\theta) + (\mathbf{b}^T \mathbf{v}) + \sum_{j=1}^n \log \left(1 + \exp \left(a_j + \sum_{i=1}^m w_{ij} v_j \right) \right) \end{aligned} \quad (5.13)$$

$$= -Z(\theta) + (\mathbf{a}^T \mathbf{h}) + \sum_{j=1}^m \log \left(1 + \exp \left(b_j + \sum_{i=1}^n w_{ji} h_j \right) \right) \quad (5.14)$$

Depending on the dimensions of W , the larger of \mathbf{h} and \mathbf{v} is summed out (Eq. (5.13) or (5.14), respectively). For

$$Z(\theta) = \sum_{\mathbf{v}} \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h}; \theta)), \quad (5.15)$$

all possible values of $\mathbf{v} \in \{0, 1\}^n$ or $\mathbf{h} \in \{0, 1\}^m$ must be considered.

DETAILS ON LEARNING PROCEDURE. Since the Shifter and Bars and Stripes datasets are quite small, we use true batch learning. Minibatch learning gave similar results which are not shown here. For the MNIST dataset we use minibatches of size 400. Other batch sizes gave similar results, small batch sizes just result in less stable learning curves. No weight-decay, learnrate schedule, momentum or sparsity bias was used in either case.

For PCD, we used as many chains as there were samples in a batch. This was mainly done for convenient implementation. Since varying the batch size did not have significantly influence on our main observations, PCD does not seem to be very sensitive to the particular number of persistent chains used.

Details on Annealed Importance Sampling.

As a base model for AIS, we used the standard procedure of modeling each visible unit as independent. This corresponds to a energy model consisting only of the bias term $E(\mathbf{v}, \theta) = -\mathbf{b}^T \mathbf{v}$.

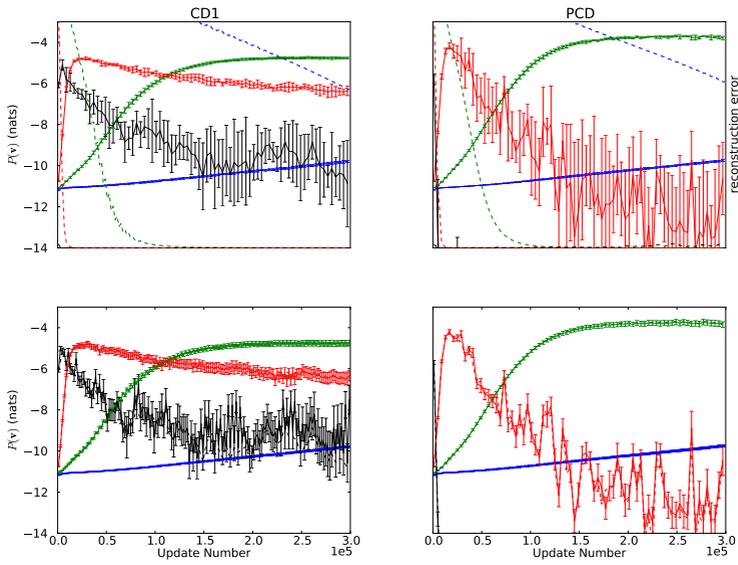


Figure 5.5: Learning curves for Bars and Stripes. Color coding as in Fig. 5.4.

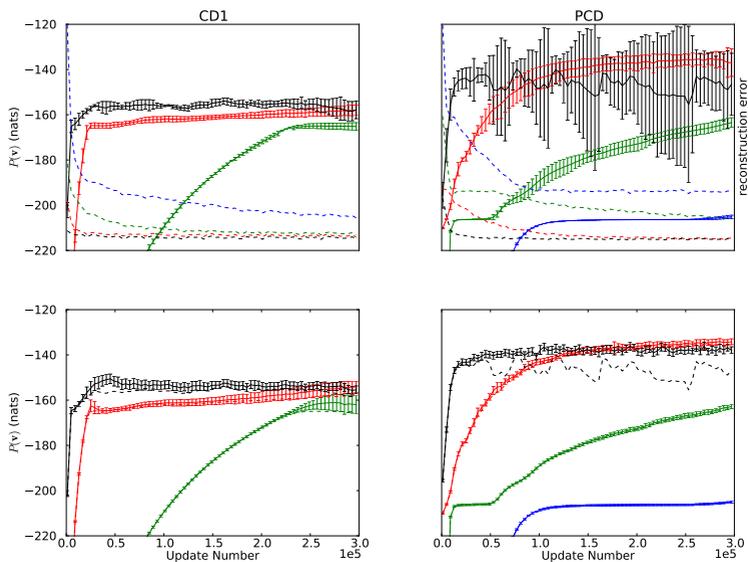


Figure 5.6: Learning curves for MNIST for varying learnrates. Color coding as in Fig. 5.4. Note especially the divergence of AIS and the ground truth likelihood in the plot on the lower right.

The maximum likelihood solution for \mathbf{b} is then given by: $\mathbf{b} = \log(\bar{\mathbf{v}}) - \log(1 - \bar{\mathbf{v}})$, where $\bar{\mathbf{v}}$ is the mean of \mathbf{v} over the dataset.

We initialized \mathbf{b} to a smoothed version of the maximum likelihood solution, by using $\bar{\mathbf{v}}' = \bar{\mathbf{v}} + 0.1$. This heuristic was chosen on MNIST and kept unchanged for the other datasets. The schedule employed for the parameter β was proposed by Salakhutdinov and Murray (2008), where β was taken uniformly spaced in three intervals. Specifically, we use 500 $\beta \in [0, 0.5[$, then 4000 $\beta \in [0.5, 0.9[$ and 10 000 $\beta \in [0.9, 1.0]$ for a total of 14 500 intermediate distributions. Using β that were uniformly distributed in $[0, 1]$ did not produce significantly different results. We used 512 parallel Markov chains for a stable approximation.

Empirical Verification of Implementation.

Since the calculations for training and evaluation of the true and approximate (AIS) log-likelihood are quite demanding, we parallelize them using the NVIDIA CUDA framework and the CUV python library. Training and evaluation jobs are further distributed over a cluster of 25 GPUs on eight computers. A complete run for MNIST with 25 hidden units can be computed in about ten minutes. Two adjustments were made to ensure numerical stability. Firstly, the large sums in Eqs. (5.13) and (5.14) are calculated using the Kahan algorithm on CPU and logarithmic summing on GPU. Secondly, we set $\log(1 + \exp(x)) := x$ if $x > -\log(\epsilon)$, where ϵ is machine precision for double (CPU) or float (GPU). We calculated the true average log-likelihood of MNIST for a trained RBM both on GPU and on CPU and consistently find that the results differ only after the fifth decimal place.

5.2.3 Results

During training, we save the weights every 4000 weight updates and determine the ground truth log-likelihood, the estimated log-likelihood using AIS, norm of weight matrix and the reconstruction error. All data shown is for RBMs with 25 (24) hidden units for MNIST (Bars and Stripes and Shifter). We repeated each experiment with five different random initializations except the special

case for MNIST where AIS does not follow the ground truth, which was repeated 10 times to ensure that this is a repeatable observation.

LIKELIHOOD DEVELOPMENT OVER TRAINING. In both, PCD and contrastive divergence a single Markov chain step (CD1), learning curves were strongly dependent on the learnrate. Further parameters (learnrate schedules, weight decay) are hard to set and may lead to convergence to suboptimal results (Fischer and Igel, 2010). We find that we can obtain results which compare favorably with the literature without the use of these methods.

With regards to learnrates, we observe that for all datasets (top rows of Figs. 5.4 to 5.6) learning speed correlates with learnrate. In contrast to, for example, neural networks, large learnrates in RBM training do not result in unstable learning and bad optima but in divergence of the log-likelihood function after achieving good likelihood values. This divergence of RBM training has been observed before (e.g. Fischer and Igel, 2010; Desjardins et al., 2010) and we could reproduce this effect in larger RBMs on MNIST (Fig. 5.6, top row). The divergence effect is more pronounced for high learnrates. This is expected for PCD, since PCD requires “small” learnrates, so that the persistent chains stay close to the current model distribution. Still, large learnrates reached good solutions quickly before training diverged. We also observed divergence for smaller learnrates when training was carried on long enough (data not shown).

In general, learning with higher learnrates is more dependent on the seed (see e.g. the upper left plot of Fig. 5.4).

The optimum in PCD learning is consistently larger than the optimum in CD1 learning, which is conforming to the literature (e.g. Tieleman, 2008).

AIS APPROXIMATION OF LOG-LIKELIHOOD.

Generalizing the above observations and accepting the divergence of RBM training algorithms, we would like to choose a learnrate which is large and stop before the likelihood starts to diverge. As can be observed in all plots, the “reconstruction error” (dashed lines in top row) mentioned before is of no help at

all. The value always decreases. We therefore consider AIS as an evaluation method of the learning process.

For the toy datasets (Bars and Stripes, Shifter) and also for most cases in the MNIST dataset, AIS approximates the ground truth likelihood accurately. This is even the case when learning is very unstable (e. g. left plot in Fig. 5.3). We also measured the uncertainty of AIS. The error bars in the bottom rows of Figs. 5.4 to 5.6 show the uncertainty of AIS averaged over learning trials with varying random seeds. As long as learning is stable, the uncertainty is very small. Surprisingly, however, for various trials ground truth likelihood dropped dramatically, while AIS completely failed to either capture the change in likelihood or to increase uncertainty (see right of Fig. 5.3 for single run and bottom right of Fig. 5.6 for the average over trials). We also observed this behavior for smaller learnrates (which diverge later) on MNIST, but not at all on the toy datasets. Therefore generalizations from small models to larger ones should be taken with a grain of salt. To investigate the reason for the drop in likelihood, we examined the persistent chains at the problematic update steps, but found no obvious deficiency in the mixing. Therefore, AIS, while it models easy cases perfectly and some hard cases very well should be used cautiously for such purposes as stopping learning, finding hyper-parameters and evaluating new learning algorithms.

5.2.4 *Conclusions*

While RBMs are successful learning machines, their training remains a tricky task. We evaluated training methods with minimal parameter sets on small to medium-sized problems to analyze the behavior of CD1 and PCD training with respect to parameter selection and divergence. Our findings suggest that often a simple setup provides good results, provided one finds a suitable learnrate. We can confirm divergence of CD1 and PCD learning algorithms and therefore investigated AIS as a stopping and evaluation method. The presented results suggest that AIS is often, but not always, a good measure of the training progress and we suggest further investigation into alternative criteria.

Following the publication of the contents in this section, alternative learning schemes, stopping criteria and evaluation metrics were proposed. Cho et al. (2011b) and Cho et al. (2013), e.g., worked on Gaussian-Bernoulli RBM (GRBM), which are even more sensitive to the chosen learnrate and its schedule. The authors propose adaptive step sizes, which are automatically tuned according to likelihood approximations. An upper bound on the step size ensures that the likelihood approximation is stable. Buchaca et al. (2013) set out from our—rather negative—result to search for stopping criteria for RBM training other than the reconstruction error. They proposed an estimator which relates the probability of the training set to the probability of an artificially created dataset. At least for the small Shifter and Bars and Stripes datasets, this seems promising, but our results show that generalization to larger models is not straightforward. Finally, Grosse et al. (2013) propose an alternative to AIS, where the annealing chain is improved using moment averaging. Their experiments on slightly smaller models than ours suggest that their method has significantly fewer problems than AIS.

5.3 TWO-LAYER CONTRACTIVE ENCODINGS FOR SEMI-SUPERVISED LEARNING

In Chapter 2, we have seen how deep neural networks can be initialized using unsupervised learning—pre-training—of feature hierarchies. This initialization was found to improve results for many-layered—so-called *deep*—neural networks (Hinton et al., 2006; Ranzato et al., 2007; Bengio et al., 2006) and has spurred research on understanding and improving feature learning methods (e.g. Erhan et al., 2009; Cho et al., 2011a; Glorot and Bengio, 2010; Rifai et al., 2011c). Classical pre-training for a multi-layer perceptron is performed layer-wise greedily, that is, after training the parameters of one layer, they are fixed for the training of the next higher layer parameters. After pre-training, all parameters are *fine-tuned* jointly in a supervised manner on the task.

The greedy initialization steps successively build more complex features and at the same time avoid problems occurring when training deep architectures directly. Erhan et al. (2009)

argue that many-layered architectures have more local minima and that gradients are becoming less informative when passing through many layers. In contrast, commonly employed auto-encoders (AE) and restricted Boltzmann machines (RBM) are shallow. They have fewer local minima and gradients are not diluted.

In some cases, layer-wise pre-training might not help fine-tuning, e.g. when extracted features bear no relation with the desired output. Recently, Rifai et al. (2011c) showed that *stable* features of the training data are useful for supervised training on wide range of datasets. These features do not change when the input varies slightly. The failure mode we address in this section is when these stable features cannot be recognized by the pre-training method. Both, AEs and RBMs yield encoders which consist of a linear projection followed by a smooth thresholding function. This is a highly restricted function class. In fact, while the deep MLP can learn almost arbitrary functions (Hornik et al., 1989), Minsky and Papert (1969) showed that a one-layer neural network as is used for AE and RBM is not able to learn the class of not linearly separable functions, of which the well-known XOR problem is the simplest example.

We argue here that deep architectures pre-trained with the common one-layer encoders often fail to discover features which are of the XOR class (which we shall refer to as *non-linear* features), and that fine-tuning may not repair this defect. We construct problems that cannot profit from pre-training and show that pre-training may even be counter-productive in these cases.

These problem cases can be solved for auto-encoders by introducing a hidden layer in the encoder, yielding a compromise between the advantages of increased expressiveness and disadvantages of increased depth.

To remedy the problem of increased depth, we propose to extend contractive regularization (Rifai et al., 2011c) to two-layer auto-encoder pre-training. We further propose to add shortcuts to the auto-encoder, which helps learning a combination of simple and complex features. Our training procedure employs the method of linearly transforming perceptrons, recently proposed by Raiko et al. (2012).

We show that contractive regularization can resolve the constructed cases and performs better than greedy pre-training on benchmark datasets. Finally, we evaluate the proposed two-layer encoding with shortcuts method on the task of semi-supervised classification of handwritten digits and show that it achieves better generalization than greedy pre-training methods when only a few labeled examples are available.

5.3.1 *Related Work*

The representational power of deep architectures has been thoroughly analyzed (Le Roux and Bengio, 2008; Bengio and Delalleau, 2011). Le Roux and Bengio (2008) showed that, in principle, any distribution can be represented by an RBM with $M + 1$ hidden units, where M is the number of input states with non-zero probability. The question of which *features* can be represented, is not addressed, however. Bengio and Delalleau (2011) analyzed the representational power of deep architectures and showed that they can represent some functions with exponentially fewer units than shallow architectures. It is not clear, however, whether these representations can be learned greedily.

There is considerable evidence that the performance of deep architectures can be improved when the greedy initialization procedure is relaxed. Salakhutdinov and Hinton (2009) report advantages when performing simultaneous unsupervised optimization of all layer parameters of a stack of RBMs as a deep Boltzmann machine. The authors rely on a greedy initialization, however, which we demonstrate here might establish a bad starting point. Deep Boltzmann machines could also be initialized using two-layer encoders, using the two-stage pretraining method of Cho et al. (2012). Ngiam et al. (2011) train a deep belief network without greedy initialization and also report good results. Their approach might not scale to many-leveled hierarchies though, and relies on a variant of contrastive divergence (cf. Section 5.2) to approximate the gradient. More recently, Cireřan et al. (2012b), Krizhevsky et al. (2012), and Hinton et al. (2012) obtained top results on a number of image classification data sets with deep neural networks. They did not rely on pre-training, but used other

regularization methods, such as convolutional network structure with max-pooling, generation of additional training examples through transformations, bagging, and dropout.

5.3.2 Background

In this section, we discuss each of the three methods combined on our approach—auto-encoders, contractive encodings and linear transformations—in more detail.

Auto-Encoders

An auto-encoder consists of an encoder and a decoder. The encoder typically has the form

$$\mathbf{h} = \mathbf{f}_{\text{enc}}(\mathbf{x}) = \sigma(\mathbf{o}) = \sigma(W\mathbf{x}), \quad (5.16)$$

where σ is a component-wise sigmoid non-linearity, e.g. $\sigma_i(\mathbf{o}) = \tanh(o_i)$. The encoder transforms the input $\mathbf{x} \in \mathbb{R}^N$ to a hidden representation $\mathbf{h} \in \mathbb{R}^M$ via the (learned) matrix $W \in \mathbb{R}^{M \times N}$. The decoder is then given by

$$\hat{\mathbf{x}} = \mathbf{f}_{\text{dec}}(\mathbf{h}) = W'\mathbf{h} \in \mathbb{R}^N, \quad (5.17)$$

where we restrict ourselves to the symmetric case $W' = W^\top$. Even though biases are commonly used, we omit them here for clarity. The main objective for auto-encoders is to determine W such that $\hat{\mathbf{x}}$ is similar to \mathbf{x} . For binary \mathbf{x} , this amounts to minimizing the cross-entropy loss

$$\ell_{\text{bin}}(\mathbf{x}, W) = -\sum_i^N (x_i \log(\hat{x}_i) + (1 - x_i) \log(1 - \hat{x}_i)). \quad (5.18)$$

Auto-encoders have gained popularity as a method for pre-training of deep neural networks (e.g. Bengio et al., 2006). In deep neural networks, gradient information close to the input layer is “diluted” since it passed through a series of randomly initialized nonlinear layers. This effect makes it difficult to learn good high-level representations (Erhan et al., 2010a). pre-training

moves weights to an area where they relate to the input and therefore allow for cleaner gradient propagation. Bengio (2009) and Bengio et al. (2013) further hypothesize that stacking of unsupervised neural networks disentangles factors of variations and that the untangled representations make supervised learning easier.

Pre-training typically proceeds greedily. Consider a deep network for a classification task, $\mathbf{y} = W^{(L)}\mathbf{h}^{(L)}$, with $\mathbf{h}^{(l+1)} = \sigma(W^{(l)}\mathbf{h}^{(l)})$, where \mathbf{y} is the output layer, $\mathbf{h}^{(l)}$, $l \in 1, \dots, L$ are hidden layers, and $\mathbf{h}^{(0)} = \mathbf{x}$ is the input layer. Then, $L - 1$ auto-encoders are learned in succession, minimizing $\ell.(\mathbf{h}^{(l)}, W^{(l)})$ and keeping $W^{(l' < l)}$ fixed. It is frequently observed that following this protocol, successively higher-level representations of the inputs are attained (Bengio et al., 2006). This pre-training is, however, greedy, and a joint optimization of all layers might yield superior results in principle. In practice, however, joint optimization without pre-training suffers from the same gradient dilution problem as originally addressed by the deep-learning methodology and often yields bad performance. After pre-training, the supervised classification objective is optimized jointly with respect to all parameters $(W^{(1)}, \dots, W^{(L)})$. This final step is called *fine-tuning*.

Contractive Encodings

To avoid overfitting the training data, or in order to obtain the overcomplete representations, it is common to regularize the auto-encoder learning. A common regularizer for MLPs is the L_2 penalty on the weight matrices. This regularizer is well-motivated for linear methods (e.g. ridge regression or logistic regression), where it penalizes strong dependence of \mathbf{y} on few variables in \mathbf{x} , and thus ensures invariance of \mathbf{y} to small changes in \mathbf{x} .

For MLPs, which contain saturating non-linearities, this desirable property can be achieved with both strongly positive and negative weights. Rifai et al. (2011b) and Rifai et al., 2011a show that the generalization of the L_2 penalty in the presence of saturating non-linearities is the contractive penalty.

The contractive penalty penalizes the squared Frobenius norm of the Jacobian $J_{\mathbf{f}_{\text{enc}}}$ of \mathbf{f}_{enc} with respect to the input \mathbf{x} , where the Jacobian is defined by

$$J_{\mathbf{f}_{\text{enc}}} = \begin{bmatrix} \frac{\partial f_{\text{enc}}^1}{\partial x_1} & \dots & \frac{\partial f_{\text{enc}}^1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{\text{enc}}^M}{\partial x_1} & \dots & \frac{\partial f_{\text{enc}}^M}{\partial x_N} \end{bmatrix}.$$

By minimizing the Jacobian $J_{\mathbf{f}_{\text{enc}}}$, the hidden representation computed by the encoder becomes more invariant to small changes in the input \mathbf{x} , resulting in robust representations.

The combined objective is given as

$$\ell_{CAE}(\mathbf{h}^{(l)}, W^{(l)}) = \ell(\mathbf{h}^{(l)}, W^{(l)}) + \lambda \|J_{\mathbf{f}_{\text{enc}}}\|^2, \quad (5.19)$$

where λ is the regularization strength. Rifai et al. (2011b) and Rifai et al. (2011a) demonstrate that training a stack of simple auto-encoders with the contractive penalty produces features which identify the data manifold, which later on helps fine-tuning.

Linear Transformations in Perceptrons

Raiko et al. (2012) proposed a method of linearly transforming perceptrons in a deep MLP network to avoid difficulties in training a deep neural network without pre-training.

Let us focus on a single hidden layer within a possibly deep MLP network. The inputs to this layer are denoted \mathbf{x}^k and its outputs are \mathbf{y}^k , where k is the sample index. We allow short-cut connections that by-pass one or more hidden layers such that the inputs to each hidden layer may be distributed over several previous layers of the network. The mapping from \mathbf{x}^k to \mathbf{y}^k is modeled as

$$\mathbf{y}^k = W\sigma(V\mathbf{x}^k) + C\mathbf{x}^k, \quad (5.20)$$

where W , V , and C are weight matrices. In order to avoid separate bias vectors that complicate formulas, the input vectors \mathbf{x}^k are assumed to have been supplemented with an additional component that is always one.

Let us assume that σ is a tanh nonlinearity and supplement it with auxiliary scalar variables α_i and β_i for each component σ_i . We define

$$\sigma_i(\mathbf{b}_i \mathbf{x}^k) = \tanh(\mathbf{b}_i \mathbf{x}^k) + \alpha_i \mathbf{b}_i \mathbf{x}^k + \beta_i, \quad (5.21)$$

where \mathbf{b}_i is the i th row vector of matrix V .

α_i 's and β_i 's are updated during training in order to help learning the other parameters W , V , and C . By updating α_i 's and β_i 's we will ensure that

$$0 = \sum_{k=1}^K \sigma_i(\mathbf{b}_i \mathbf{x}^k), \text{ and } \quad 0 = \sum_{k=1}^K \sigma_i'(\mathbf{b}_i \mathbf{x}^k). \quad (5.22)$$

These are satisfied by setting α_i and β_i to

$$\alpha_i = -\frac{1}{K} \sum_{k=1}^K \tanh'(\mathbf{b}_i \mathbf{x}^k), \quad \beta_i = -\frac{1}{K} \sum_{k=1}^K \left[\tanh(\mathbf{b}_i \mathbf{x}^k) + \alpha_i \mathbf{b}_i \mathbf{x}^k \right].$$

We motivate these seemingly arbitrary update rules below.

The effect of changing the transformation parameters α_i and β_i are compensated exactly by updating the shortcut mapping C by

$$C_{\text{new}} = C_{\text{old}} - W(\boldsymbol{\alpha}_{\text{new}} - \boldsymbol{\alpha}_{\text{old}})V - W(\boldsymbol{\beta}_{\text{new}} - \boldsymbol{\beta}_{\text{old}}) [0 \ 0 \dots 1], \quad (5.23)$$

where $\boldsymbol{\alpha}$ is a matrix with elements α_i on the diagonal and one empty row below for the bias term, and $\boldsymbol{\beta}$ is a column vector with components β_i and one zero below for the bias term. Thus, any change in α_i and β_i does not change the overall mapping from \mathbf{x}^k to \mathbf{y}^k at all, but they do change the optimization problem instead.

One way to motivate the transformations in Eq. (5.22), is to study the expected output \mathbf{y}_i and its dependency on the input \mathbf{x}_i :

$$\frac{1}{K} \sum_k \mathbf{y}^k = W \left[\frac{1}{T} \sum_k \sigma(V \mathbf{x}^k) \right] + C \left[\frac{1}{K} \sum_k \mathbf{x}^k \right], \quad (5.24)$$

$$\frac{1}{K} \sum_k \frac{\partial \mathbf{y}^k}{\partial \mathbf{x}^k} = W \left[\frac{1}{K} \sum_k \sigma'(V \mathbf{x}^k) \right] V^T + C. \quad (5.25)$$

We note that by making nonlinear activations $\sigma(\cdot)$ zero mean in Eq. (5.22) (left), we disallow the nonlinear mapping $W\sigma(V\cdot)$ to affect the expected output \mathbf{y}^k , that is, to compete with the bias term. Similarly, by making the nonlinear activations $\sigma(\cdot)$ zero slope in Eq. (5.22) (right), we disallow the nonlinear mapping $Wf(V\cdot)$ from affecting the expected dependency on the input, that is, to compete with the linear short-cut mapping C . In traditional neural networks, the linear dependencies (expected $\partial\mathbf{y}^k/\partial\mathbf{x}^k$) are modeled by many competing paths from an input to an output (via each hidden unit), whereas this architecture gathers the linear dependencies to be modeled only by C .

Raiko et al. (2012) showed experimentally that less competition between parts of the model speeds up learning significantly. It also helped to attain state-of-the-art learning results for MLP networks on three tasks (MNIST classification, CIFAR-10 classification, and MNIST deep auto-encoders). Vatanen et al. (2013) drew more careful connections to second-order optimization methods, showing that the reparameterization done using transformations make first-order optimization methods behave more like a second-order method.

Recently, Dauphin et al. (2014) argued that plateaus around saddle points in the parameter space dramatically slow down learning and giving an illusory impression of local minima. They proposed a second-order optimization method that is able to escape saddle points and showed that one can continue optimization from a seemingly converged optimization by a first-order method. They also discussed whether actual local minima are as big an issue as has long been thought. It remains an interesting open issue whether the good empirical performance of the transformations is related to this phenomenon.

5.3.3 *Where Pre-Training of One-Layer Encoders Fails*

Let us assume that we want to approximate the Boolean function $f(x_1, x_2) := x_1 \vee x_2$, where $\cdot \vee \cdot$ denotes the exclusive-or relation (XOR). For this purpose, we consider the neural network with a two-unit hidden layer shown in Fig. 5.7 (left) and perform auto-encoder pre-training for the first-layer matrix B . During the pre-

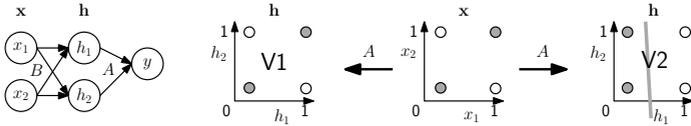


Figure 5.7: Auto-encoder pre-training can be counter-productive. The simple network on the left should learn $y = x_1 \vee x_2$. pre-training of A makes an uninformed choice between representations $V1$ and $V2$ without loss of generality, but only $V2$ is linearly separable and helps fine-tuning.

training phase, two filters have to be learned, mapping the input vector again to a two-dimensional space $\mathbf{h} = (h_1, h_2)$. Without further information, this mapping might choose a representation which is not linearly separable (denoted “ $V1$ ” in Fig. 5.7). In this case, pre-training does not aid fine-tuning, it chooses a feature representation that is not helpful for the classification task. Of course, this argument merely stresses the distinction between supervised and unsupervised learning. It does not follow that pre-training is not helpful *per se*. Our observation has, however, important consequences for *greedy* pre-training.

We can easily extend the argument of the previous paragraph to a case where greedy pre-training does not find stable non-linear features that are obvious from the data. Let us assume we have a dataset of three variables, where $\mathbf{x}^k = (x_1^k, x_2^k, x_1^k \vee x_2^k)$. The only stable feature of this dataset is $x_1 \vee x_2$, i.e. a two-layered denoising auto-encoder should be able to recover x_3^k from the first two components and any of the x_1^k, x_2^k from the other variable and $x_1^k \vee x_2^k$. If unfortunate greedy training of the first layer prevents the second layer from learning that x_1 and x_2 are XOR-related — as demonstrated above — the second layer will fail to discover this relation. Even worse, pre-training might leave the weights in a state where recovery using fine-tuning is not possible. We will empirically verify these claims in the experiments section.

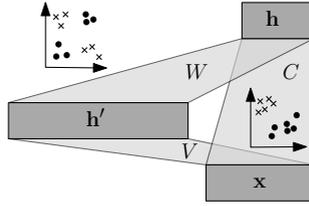


Figure 5.8: Schematic visualization of our encoder. Features \mathbf{h} of input \mathbf{x} are determined both by a one-layer encoder via C , and by a two-layer encoder via V and W . Contractive regularization (Rifai et al., 2011b) and two-layer contractive regularization (Schulz and Behnke, 2012c) are used to learn stable linear/non-linear representations in \mathbf{h} , respectively. Linear transformations in the two-layer part are moved to C using compensations (Raiko et al., 2012) (not shown).

5.3.4 Two-Layer Encoders and Contractive Regularization

Considering the failure mode of one-layer encoders discussed above, an intuitive extension is to make the encoder more powerful by adding a hidden layer $\mathbf{h}' \in \mathbb{R}^P$ to the encoder, such that $\mathbf{f}_{\text{enc}}(\mathbf{x}) = \mathbf{h} = \sigma(W\mathbf{h}') = \sigma(W\sigma(V\mathbf{x}))$, with $\mathbf{x} \in \mathbb{R}^M$, $W \in \mathbb{R}^{N \times P}$, $V \in \mathbb{R}^{P \times M}$. Extending contractive regularization (Rifai et al., 2011c) to two-layer encoders yields

$$\|J_{\mathbf{f}_{\text{enc}}}(\mathbf{x})\|_F^2 = \sum_n^N \sum_m^M (1 - \mathbf{f}_{\text{enc}}(\mathbf{x})_n^2)^2 \left(\sum_p^P w_{np} v_{pm} (1 - \mathbf{h}'(\mathbf{x})_p^2) \right)^2, \quad (5.26)$$

where $\mathbf{h}(\mathbf{x})$ is the hidden layer activation. In contrast to one-layer contractive regularization, which has the complexity of a forward pass, this regularizer is $O(MPN)$. To reduce the time required to do experiments, we only compute the regularizer for few randomly chosen instances in the mini-batch. The precise number is a tradeoff between acceptable training time and accuracy, which needs to be cross-validated. The simultaneous perturbation method (SPSA, Spall, 1998), which has not received much attention recently, could potentially be employed to reduce the regularization cost to a second forward-pass.

5.3.5 Two-Layer Encoders and Shortcut Connections

In this section, we argue that while two-layer encodings are harder to learn, we can combine their ability to detect highly non-linear features with the easy-to-learn one-layer encodings by introducing shortcuts. Shortcut weights C (see, e.g., Eq. (5.20)) from the input to the second hidden layer can be regularized as in (Rifai et al., 2011b), while the two-layer encoder is regularized as introduced above. We employ linear transformations and compensations (Section 5.3.2) to ensure that simple features continue to be learned by the shortcut weights, while the two-layer part of the encoder can focus on the difficult features. For this purpose, we extend the two-layer contractive regularizer to account for the linear transformations in Eqs. (5.20) and (5.21),

$$\|J_{\mathbf{f}_{\text{enc}}}(\mathbf{x})\|_F^2 = \sum (1 - \mathbf{f}_{\text{enc}}(\mathbf{x})^2)^{2^\top} (C + W(V^\alpha + V'))^2, \quad (5.27)$$

where $v_{pm}^\alpha = \alpha_p v_{pm}$, $v'_{pm} = v_{pm}(1 - \tanh^2(v_p \cdot \mathbf{x}))$. Fig. 5.8 illustrates the proposed encoder structure.

5.3.6 Experiments

In our experiments, we aim to establish the following claims:

1. One-layer encoders are severely restricted.
2. Two-layer encoders can be better learned with the proposed contractive regularizer.
3. Two-layer encoders can further profit from shortcut connections in the case of semi-supervised learning.

Our experiments follow a common protocol. For a fixed architecture, we repeatedly sample all free hyper-parameters from the distributions detailed in Table 5.2. For a weight matrix $W \in \mathbb{R}^{N \times M}$, weights are initialized uniformly with $w_{ij} \sim \mathcal{U}(-\sqrt{6/(N+M)}, \sqrt{6/(N+M)})$ as proposed by Glorot and Bengio (2010). The dataset is split in training, validation and testing sets. We stop each training stage before the loss on the validation set increases. The model with the best final validation error is trained again using training and validation set, for the same number of

Table 5.2: Hyper-parameter distribution used in our experiments.

Hyper-parameter	Distribution
Auto-encoder learnrate	$\log \mathcal{U}(0.01, 0.2)$
MLP learnrate	$\log \mathcal{U}(0.01, 0.2)$
Regularization strength (λ)	$\mathcal{U}(0.001, 2)$

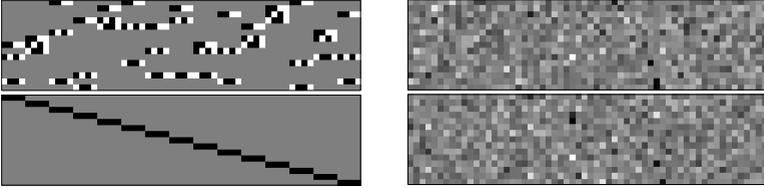


Figure 5.9: *Left*: The matrices W_{and} and W_{or}^\top used in the LDPC experiments. Weights depicted in black, gray, and white denote -1 , 0 , and 1 , respectively. *Right*: Weights of the best-performing auto-encoder after greedy pre-training on LDPC. Sections of first-layer weight matrix roughly corresponding to W_{and} and W_{or}^\top on the left. Left and right matrices should be equal up to permutation, but the greedy pre-training is unable to detect the non-linear features of the LDPC dataset.

epochs as determined in the validation phase, and is finally evaluated on the test set.

Detecting Constraint Violations in LDPC Codes

We now extend the toy example of Section 5.3.3 to a more realistic, albeit still constructed, task. A low density parity check (LDPC) code, also known as Gallager code (Gallager, 1962), is a code that allows error correction after transmission through a noisy channel. This is achieved by relating the bits in the message with a set of random constraints known to both sender and receiver. A constraint c over a set of variables \mathcal{C} is met iff $0 \equiv (\sum_{x \in \mathcal{C}} x) \pmod{2}$. Note, that the modulo operation generalizes the XOR operation to multiple binary variables.

We consider a subproblem of decoding an LDPC code, namely detecting constraint violations in the code. To this end, we construct a dataset where a code word $\mathbf{r} \in \{0,1\}^N$ is constrained by N constraints \mathcal{C}_n with three participating variables, each. Each variable participates in three random constraints. Whether a constraint is violated in \mathbf{r} can be determined by a two-layer neural network with \mathbf{r} as its input. The hidden layer has four neurons for every constraint. Each of these four neurons detects a different case where the corresponding three neurons in the input sum to an even number (i.e., for configurations 000, 011, 110, and 101). We denote the weight matrix realizing this W_{and} , as detects conjunctions in the input data. A second weight matrix, W_{or} , then detects whether any of the four neurons in the hidden layer was active and turns on the corresponding neuron in the output layer \mathbf{q} . Thus, the values in \mathbf{q} indicate which constraints in \mathcal{C} are violated in \mathbf{r} .

For our small problem size, the matrix \mathcal{C} can be generated by creating binary matrices with the correct number of ones and verifying the number of ones per row and column. The matrices W_{and} and W_{or} can then be derived directly from the constraint matrix. The matrices used in our experiments are shown on the left side of Fig. 5.9.

We now frame learning W_{and} and W_{or} as a task for the auto-encoder shown in Fig. 5.10. The auto-encoder reconstructs the code word \mathbf{r} and the constraint violation vector \mathbf{q} together:

$$(r_1, r_2, \dots, r_n, q_1, q_2, \dots, q_n) = \mathbf{x} \stackrel{!}{=} \mathbf{f}_{\text{dec}}(\mathbf{f}_{\text{enc}}(\mathbf{x})), \quad (5.28)$$

$$\text{where } \mathbf{f}_{\text{enc}}(\mathbf{x}) = \sigma(W_{\text{or}} \sigma(W_{\text{and}} \mathbf{x})) \quad (5.29)$$

$$\text{and } \mathbf{f}_{\text{dec}}(\mathbf{h}) = \sigma(W_{\text{and}}^\top \sigma(W_{\text{or}}^\top(\mathbf{h}))). \quad (5.30)$$

Since $r_i \sim \text{Binomial}(1, 0.5)$, \mathbf{r} cannot be compressed to less than N bits. Hence, a hidden layer size N creates a bottleneck where no more than the complete codeword can be represented.

The dataset is constructed from all $N = 15$ bit strings and randomly split into training (60%), validation (20%), and test set (20%). After pre-training, we fine-tune all weight matrices of the network by reconstructing (\mathbf{r}, \mathbf{q}) using the logistic reconstruction loss—again with early stopping on the validation set.

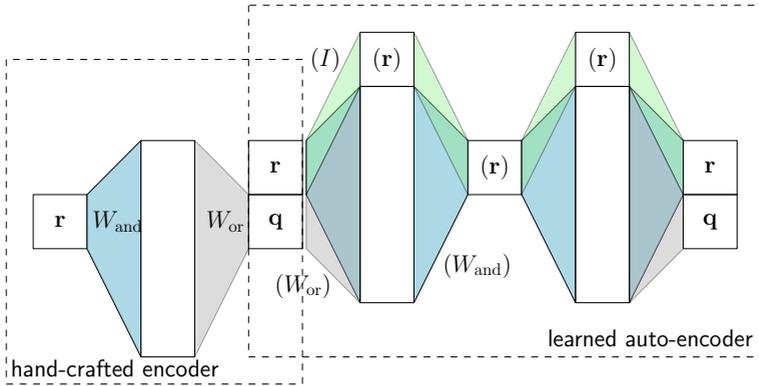


Figure 5.10: Constructed example where greedy auto-encoders fail. The matrices W_{and} and W_{or} are hand-crafted to calculate $\sum_i r_i \bmod 2$. Matrices and resulting representations in parenthesis have to be recovered, i. e. learned, by the auto-encoder to solve the reconstruction task.

An instance is reconstructed correctly if the sign of the outputs corresponds to the input. This task is clearly related to the pre-training objective, as during pre-training the hidden layer should fully represent the code.

The results are visualized in Fig. 5.11 (a). We show the fraction of draws from the hyper-parameters which performs better than a given validation error. For greedy pre-training, the chances of finding a model which performs well on the validation set are very small. This is reflected in the test errors displayed in Table 5.3 (LDPC). Only conditions where both layers are trained *simultaneously* are able to solve the task. The problem is also apparent in the learned weights after pre-training, shown on the right side of Fig. 5.9. Since the relations in the dataset can only be detected in the second hidden layer, greedy pre-training cannot learn anything useful. Finally, we compared the models to a model where we removed the first hidden layer. As expected from the construction of the dataset, this network architecture is insufficient to solve the task and yields the highest error.

The model with first and second layer size of 75 and 15, respectively, is the smallest possible model to solve the task. Since pre-

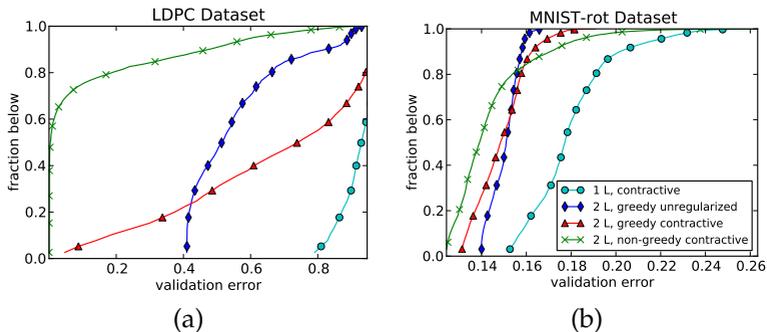


Figure 5.11: Comparison of pre-training effects for fixed architecture ($30 \times 75 \times 15$ for LDPC, $784 \times 1000 \times 500$ for MNIST-rot). Graphs show fraction of draws from hyper-parameter distribution which performs better than given validation error.

training relies on chance, its performance should improve with the layer size. To analyze the difficulty of the task, we increased the number of neurons in the first hidden layer by 400% and found only marginal improvements in the error after pre-training, as shown in Fig. 5.12. Thus, relying on chance to find good non-linear features does not work well for the LDPC dataset. Increasing the size of the second layer, on the other hand, almost solves the dataset at 160% of the minimal size. This is also expected, since the second layer size mainly influences the total compression ratio of the auto-encoder. In the case of $M = 25$, the ratio is reduced from a factor of $30/15 = 2$ to $30/25 = 1.2$.

Our experiments demonstrate that greedy layer-wise pre-training drives an auto-encoder to learn mainly “linear” features. Non-linear relations contained in the data cannot always be recovered by higher layers, confirming our first claim.

Benchmark Datasets

We also compare our approach on two benchmark datasets, MNIST (LeCun et al., 1998a) and the rotated MNIST dataset MNIST-rot (Larochelle et al., 2007). Here, we fix the architecture of the network to input size $N = 784$, first hidden layer size $P = 1000$, second hidden layer size $M = 500$, and choose a batch size of

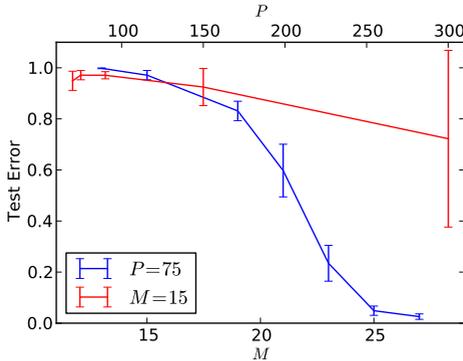


Figure 5.12: Performance of models on LDPC dataset after greedy pre-training as a function of layer size. While increasing the second layer size M with fixed $P = 75$ quickly solves the task, increasing the first layer size P while fixing $M = 15$ only helps marginally. Error-bars show standard deviations over five cross-validation runs of the best selected hyper-parameter configuration.

16. Qualitatively, we get the same results as in the constructed LDPC example for both datasets. The two-layer regularized encoder is more robust with respect to choice of hyper-parameters (Fig. 5.11 (b)) and finds better minima (Table 5.3, MNIST-rot and MNIST). Additionally, we analyzed the reconstruction error after pre-training models with best classification performance. On the MNIST-rot validation set, the one-layer case achieves an error of 126.3, greedy contractive pre-training yields 98.3, and the regularized two-layer encoder reaches 88.9. The reconstruction results for MNIST have the same ranking. This demonstrates that the features learned in the two-layer encoder are not only better for classification, they are also better representations of the input, which strongly supports our second claim.

Two-layer Encoders and Shortcut Connections

We evaluate the shortcut connections in a semi-supervised setting on MNIST. We assume that only 1200 training samples have their labels available, while all the other training samples are un-

Table 5.3: Comparison of pre-training methods for fixed architecture.

Layers	Condition		Test Error (%)		
	Pre-Training	Regularizer	LDPC	MNIST-rot	MNIST
1	no	none	88.6	13.8	1.8
1	yes	none	81.8	15.7	1.6
1	yes	contractive	71.1	14.1	1.6
2	greedy	contractive	6.8	13.2	1.6
2	greedy	none	5.3	14.6	1.7
2	no	none	0.0	12.5	1.7
2	non-greedy	none	0.0	12.5	1.7
2	non-greedy	contractive	0.0	11.4	1.4

labeled. The task is to use an MLP trained on the training samples to classify 10 000 test samples.

Our base model is a multi-layer perceptron (MLP) with two hidden layers having tanh hidden neurons. The output of the MLP is

$$\mathbf{y} = F(\sigma(W\sigma(V\mathbf{x}))), \quad (5.31)$$

where F , W and V are weight matrices. We have omitted biases for simplicity of notation. As baseline, we trained this MLP both with and without pre-training. For the pre-trained MLP, we consider the bottom two layers as an auto-encoder with two hidden layers and trained them using both labeled and unlabeled samples.

When the hidden neurons, or perceptrons, in the MLP were linearly transformed, we added shortcut connections from the input to the second hidden layer to maintain the equivalence after the transformation. In that case, the output of the MLP is

$$\mathbf{y} = F\mathbf{h} = F\sigma(W\mathbf{h}' + C\mathbf{x}), \quad (5.32)$$

Table 5.4: Classification accuracies depending on training strategy on MNIST using 1200 labeled examples. Standard deviations are over five trials with different draws of the training set.

Condition	Test Error
S	10.0 ± 1.4
U+S	7.4 ± 2.2
C+U+S	7.8 ± 1.3
T+U+S	8.0 ± 1.7
2C+U+S	7.2 ± 2.6
C+T+U+S	6.2 ± 1.2

where $\mathbf{h}' = \sigma(V\mathbf{x}) + V^a\mathbf{x} + \beta$, and C is the weight matrix of the shortcuts.²

As a comparison, we tried both using either one of the two-layer contractive encoding and the linear transformation and using both of them together. In this way, we can easily see the effectiveness of the proposed way of using both approaches together. Specifically, we used six different training strategies:

1. S: MLP trained with labeled samples (S) only,
2. U+S: MLP pre-trained with unlabeled samples (U) and fine-tuned (S),
3. C+U+S: MLP pre-trained (U) and fine-tuned (S) with two-layer contractive encoding (C),
4. T+U+S: MLP with shortcuts pre-trained (U) and fine-tuned (S) with linear transformation (T),
5. 2C+U+S: MLP pre-trained (U) with stacked contractive auto-encoders (2C) and fine-tuned (S), and
6. C+T+U+S: MLP with shortcuts pre-trained (U) and fine-tuned (S) using both the two-layer contractive encoding (C) and linear transformation (T)

² When we pre-trained the MLP as a two-layer contractive encoding, we tied the weights W and V between the encoder and decoder. However, we did not share C , α_i 's and β_i 's.

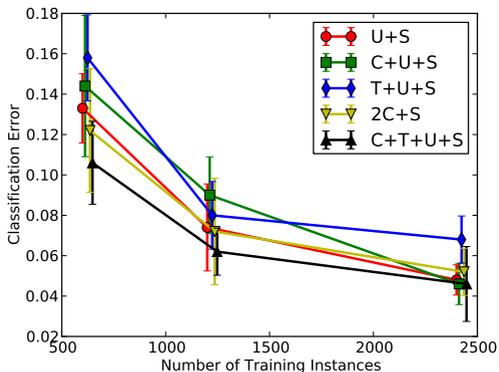


Figure 5.13: Test error for varying fine-tuning dataset size (600, 1200, 2400). Standard deviations are over five trials with different draws of the dataset. All fine-tuning hyperparameters were cross-validated for every point.

We estimated hyper-parameters such as learnrates, weight decay constant, contractive regularization strength and the sizes of the hidden layers using hyperopt (Bergstra et al., 2011). For any unregularized models, we used cross-validated L2 weight decay.

We used five-fold cross-validation, with 48 000 and 12 000 examples for training and validation, respectively, in pre-training. Pre-training was stopped when the loss on the validation set failed to improve. For fine-tuning we used 1000 examples for training and 200 for validation. Due to the small validation set size, we took great care in selecting the best model. We first determine the expected number of training epochs until the optimum is reached by averaging the early-stopping epoch over all folds. The model performance is then given by the average validation set classification error over folds at this epoch during training.

The weight matrices W and V were initialized randomly according to the normalized scale (Glorot and Bengio, 2010), while C was initialized with zeroes.

In Table 5.4, the resulting classification accuracies for all six strategies are presented. As expected, any approach with pre-

training significantly outperforms the case where only labeled samples were used for supervised training (S). The best performing strategy was the one which pre-trained the MLP as the two-layer contractive encoding using the linear transformation (C+T+U+S). This strategy was able to outperform the strategies U+S, C+U+S as well as T+U+S. Our proposed method also has an advantage over the stacked contractive auto-encoder (2C+U+S). This trend also holds when the fine-tuning dataset size is decreased, with the ratio between test and validation set size held constant, as shown in Fig. 5.13. At larger fine-tuning dataset sizes, the difference between the regularized methods vanishes.

Interestingly, using either the two-layer contractive encoding or the linear transformation only turned out to be only as good as the naïve pre-training strategy (U+S). This suggests that it is not easy to train the two-layer contractive encoding well without a good training algorithm. Only when training became easier by linearly transforming perceptrons to have zero-mean and zero-slope on average, we were able to see the improvement (C+T+U+S), which confirms our third claim.

5.3.7 Discussion

Two-layer encoders are a natural extension of greedy pre-training that enables learning of non-linear features. As stressed in Section 5.3.6, an important class of features are disjunctions of conjunctions. These functions have many intermediate results compared to the number of outputs, which requires many features in the hidden layers. Another important class of functions are subspace projections, for instance in independent subspace analysis (ISA), where the projection of features in the subspace is stable, but the computed features are not.

Such overcomplete features, where $M > N$, are often successfully used in deep learning, last but not least by Rifai et al. (2011c) and Rifai et al. (2011a). Intuitively, the overcomplete representation in the first hidden layer provides the second hidden layer more combinations to choose from. A two-layer auto-encoder, on the other hand, does not need to guess which features might help in the higher layer, since they are jointly determined. There-

fore, two-layer auto-encoders might be able to achieve higher performance more reliably with the same number of hidden units, which is supported in our experiments by greater robustness to the choice of hyper-parameters and lower reconstruction error.

Especially the LDPC example dataset shows that overcomplete intermediate representations are crucial for some tasks. With one-layer encoders, strong regularization is required to avoid learning the identity function. A two-layer encoder, on the other hand, can have an arbitrarily large hidden layer which only captures the intermediate results of the feature calculation. Regularization—here, stability of the features—is only applied to the second hidden layer, which does not need to be overcomplete.

Along with its advantage, the two-layer encoder comes with a difficulty in training. We observed this difficulty in the case of semi-supervised learning where only a fraction of samples were allowed to have labels. The method of linearly transforming neurons in a deep neural network was used to overcome this difficulty. Interestingly, the experiments showed that it is important to use the two-layer contractive encoding as well as the linear transformation to achieve good performance.

5.3.8 *Conclusions*

Common pre-training of deep architectures by RBM and AE simplifies one hard deep problem to multiple less difficult single-layer ones. In this section, we argued that this simplification goes one step too far, to the extent where the class of features which can be learned by the pre-training procedure is restricted severely.

Guided by the observation that one-layer neural networks cannot learn functions in the exclusive-or class, we constructed a task to detect constraint violations in low density parity check codes, which relies heavily on modulo computations. For this dataset, layer-wise pre-training was counterproductive for fine-tuning and only two-layer methods could solve the task.

To obtain unrestricted representational power, we employed two-layer encoders, which can be regularized using an adaptation of the contractive regularizer (Rifai et al., 2011c) and combined with one-layer encoders by using linear transformations

and the powerful learning algorithm developed by Raiko et al. (2012) and Vatanen et al. (2013).

We empirically demonstrated the validity of our claim by showing that on a task of classifying handwritten digits, pre-training with two-layer encoders resulted both in better average performance under the same hyper-parameter prior and better absolute performance. For semi-supervised learning, when only few training samples out of training samples are assumed to have annotated labels, we showed that pre-training indeed helps significantly. Furthermore, we were able to see that generalization performance could be improved by pre-training an MLP with a two-layer contractive encoding using the linear transformation, further confirming the validity of our claim.

Neural networks have a long history in the task of image classification, e.g. on the MNIST, NORB, and Caltech 101 datasets. For these datasets, neural networks rank among the top competitors (Cireşan et al., 2011). Despite the success, we should note that these image classification tasks are quite artificial. Typically, it is assumed that the object of interest is centered and at a fixed scale, i.e. that the localization problem has been solved. Natural scenes rarely contain a single object or object class. Besides object detection (Chapter 7), object class segmentation is thus an important step towards general image understanding.

The task of object class segmentation is quickly summarized in Fig. 6.1. Formally, we want to assign a label $\hat{y}_i = f_i(\mathbf{x}, \theta)$ to every pixel x_i in the image \mathbf{x} using the estimator \mathbf{f} , where the label y_i corresponds to the class of the object $c(x_i)$ that the pixel x_i belongs to. This implies that objects are not distinguished if they are of the same class. In contrast to object detection (Chapter 7), this task maps well to traditional CNN architecture, since the output of the estimator has a finite, fixed size. Due to the strong class imbalance produced by classes with many pixels (e.g. wall, sky, ...), the performance of object class segmentation methods is measured in two ways: The average pixel classification error

$$E_{\text{pix}}(\mathcal{D}, \theta) = \langle \langle [y_i = f_i(\mathbf{x}, \theta)] \rangle_i \rangle_{\mathbf{x} \in \mathcal{D}} \quad (6.1)$$

and the average class error

$$E_{\text{cls}}(\mathcal{D}, \theta) = \left\langle \left\langle [k = f_i(\mathbf{x}, \theta)]_{i:c(y_i)=k} \right\rangle_{\mathbf{x} \in \mathcal{D}} \right\rangle_{k \in \mathcal{C}}, \quad (6.2)$$

which puts more emphasis on less-represented classes.

In this chapter, we propose variations of the convolutional neural network (CNN) for object class segmentation. Specifically, we show that CNN trained from scratch can compete with state-of-the-art object class segmentation methods on multiple RGB and RGB-D datasets, at competitive speed. To achieve this, we



Figure 6.1: Object class segmentation example from the NYUD. Left: input image. Right: input overlaid with ground truth pixel-wise labels (walls, furniture, objects). Black regions in the ground truth are unlabeled and do not contribute to the loss.

- propose new models and training methods in Section 6.1
- propose novel pre-processing for the color and depth input of CNN (Sections 6.1 and 6.2),
- show that depth normalization employed by random forest (RF) in Chapter 4 can also improve the CNN performance, outperforming the RF baseline in Section 6.3, and
- recurrent convolutional neural network (RCNN) can further improve the performance by processing video streams instead of single images (Section 6.4).

6.1 LEARNING OBJECT CLASS SEGMENTATION WITH CNN

In this section, we introduce a novel DNN model for object-class segmentation, as well as multiple training and pre-processing techniques. Combining these, we demonstrate that DNN can compete with state-of-the-art object class segmentation methods on three common datasets.

6.1.1 *Methods*

NETWORK ARCHITECTURE Our network architecture (Fig. 6.2) extends the standard CNN architecture (LeCun et al., 1998a). We

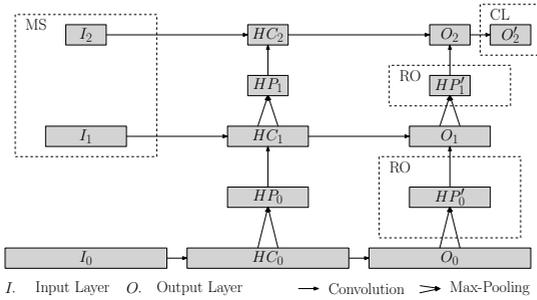


Figure 6.2: Network with three convolutional layers. Dashed parts are optional: MS multiscale inputs, RO reused outputs, CL class location filter.

use interleaved convolution and pooling layers, with filters of size 7×7 and non-overlapping 2×2 max-pooling throughout. All hidden layers have 32 maps. In addition to the standard architecture, we introduce several notable differences. Firstly, we use multiple maps for output, since we are dealing with image-like outputs and a multi-class classification problem. The cost function of the output map is either the pixel-wise cross entropy loss

$$\ell_{ce}(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_i y_i \ln \left(\frac{\exp(\hat{y}_i)}{\sum_j \exp(\hat{y}_j)} \right) \quad (6.3)$$

(i.e. softmax) or the pixel-wise squared ε -insensitive loss after a sigmoid non-linearity

$$\ell_{ei}(\mathbf{y}, \hat{\mathbf{y}}) = \max \left(0, |y_i - (1 + \exp(\hat{y}_i))^{-1}| - \varepsilon \right)^2. \quad (6.4)$$

Here, \mathbf{y} and $\hat{\mathbf{y}}$ denote teacher and net output, respectively. The choice of the loss function depends on the task: we choose ℓ_{ce} when we are interested in the best prediction per pixel and ℓ_{ei} when classes are evaluated separately. Secondly, we use a pre-training approach with intermediate output layers. In contrast to Jain and Seung (2008) and Grangier et al. (2009), however, we do not discard pretraining results, but reuse them as input to higher layers (RO in Fig. 6.2). To achieve this, we max-pool output layers and use an identity-initialized convolution to the next output

layer. With this, the next layer can focus on learning the difference to the output of the previous layer. Thirdly, we provide inputs at multiple, coarser scales to the network (MS in Fig. 6.2). In contrast to spatial pyramids, this method does not run the same algorithm on all scales, but allows the network to access low-resolution inputs directly, e.g, when pre-trained filters of lower scales (see below) are insufficient for the current scale. Both for RO and MS, multiple paths coincide at the same layer, their results are added before applying the non-linearity. Finally, Gould et al. (2008) showed that in multi-class settings, long-range dependencies between classes can be used to improve the prediction of a CRF. For example, detecting grass with high certainty in an image location x_p implies that it is unlikely that there is sky below x_p . We extend their idea, adding a class location filter (CL in Fig. 6.2) to the output, a convolution with a wide (11×11) filter learned on top of the output layer. In contrast to the Gould et al. (ibid.), we learn to correct remaining errors using long-range information.

“Valid” convolutions (Section 2.6.4) shrink the the representation by a margin proportional to filter size. To produce predictions for the whole image in one step, we need to introduce padding. Padding also simplifies the correspondence between positions in teacher and output map. Therefore, before convolution, we pad each intermediate map with the mean of that map, keeping the map size constant and largely avoiding border effects as a result. Unless stated otherwise, our input resolution is 176×176 and hidden layers have 32 maps. Filters of convolutions are initialized randomly with the method proposed by Erhan et al. (2009). Hidden layers HC_i apply the point-wise non-linearity $\tanh(\cdot)$ followed by a 2×2 non-overlapping maximum pooling HP_i .

PREPROCESSING To facilitate batch processing of images with varying aspect ratios on GPU, we first scale the image and place it in the center of a fixed-size square image. From the squared RGB images, we extract two kinds of feature, zero phase whitening (ZCA) and histogram of oriented gradients (HOG). A convolutional $5 \times 5 \times 3$ ZCA whitening transform removes first-order correlations between neighboring pixels and between color channels. A CNN-

compatible version of HOG (Dalal and Triggs, 2005) is computed for a single scale with five non-oriented bins as follows.

For every pixel \mathbf{x}_i , we determine its gradient direction \mathbf{v}_i and magnitude $|\mathbf{v}_i|$. The direction is then quantized by linear interpolation into two of B neighboring bins at every image location, and weighted by $|\mathbf{v}_i|$, resulting in quantisation bin maps $V \in \mathbb{R}^{D \times D \times B}$, where $D \times D$ is the input dimensionality and B is the number of bins. To produce histograms of the orientation strengths present at all locations, we apply a Gaussian blur filter for each bin separately. Finally, the histograms are normalized with the L_2 -hys norm (Lowe, 2004).

The main difference to the standard HOG descriptor is that no image *cells* are combined into a single descriptor. This leaves it to the network to incorporate long-range dependencies and saves space, since our descriptor contains only B values per image position.

The resulting eight maps are standardized to zero mean and unit variance separately. Ground truth, given as a map of class indices, is transformed into one map per class, which has value one iff the pixel is associated with the class. In addition, we create a ignore mask, which is used to eliminate gradients from parts of the square image which do not belong to the original image or are not labeled in ground truth. Teacher are then scaled to the size of the output map.

LEARNING We use batch stochastic gradient descent with a learnrate of $\eta_T = 10^{-3} \cdot 0.97^T$, where T denotes the epoch number. In pooling layers, the error is propagated only to the location of the maximum (Scherer et al., 2010).

Similar to previous work (Jain and Seung, 2008; Mnih and Hinton, 2010), we use supervised pre-training. Starting with O_0 , each output layer O_i is trained for 50 epochs. Afterwards, O_i becomes a regular hidden layer, and O_{i+1} is trained. Note that the derivative of output layers with softmax $S(\mathbf{x})$ simplifies considerably when used in combination with cross-entropy loss. Here, we use

softmax nonlinearities in hidden layers and therefore have to use the more involved

$$\frac{\partial S_i(\mathbf{x})}{\partial x_j} = S_i(\mathbf{x})([i = j] - S_j(\mathbf{x})). \quad (6.5)$$

All operations are performed on GPU using the CUV library.

6.1.2 Results

To measure final outcome, we crop the region of the original image from the (quadratic) output maps and scale up to the original image size. Finally, we determine the predictions $\hat{y}_i = \arg \max_{c \in \mathcal{C}} o_{c,i}$ over the output maps \mathbf{o} and determine the classification error (Eq. (6.1)).

We evaluate our architecture on three datasets. MSRC-9 (Shotton et al., 2006) is a 9 class, 240 images dataset with about 70% of the pixels labeled. We split the dataset into a stratified training and test set containing 50% of the images each. MSRC-21 is an extended version of MSRC-9 containing 591 images with 21 labeled classes. We again use the standard split into training, validation and test set (ibid.). The common evaluation criterion for both is pixel-wise accuracy in labeled regions, we consequently use $\ell_{ce}(\cdot, \cdot)$ (Eq. (6.3)). Finally, we evaluate on INRIA Graz-02 object class segmentation database (IG02) (Marszatek and Schmid, 2007). The dataset contains three classes in 479 training and 479 test images. The evaluation criterion here is per-class precision/recall at equal error rate (PR-EER), therefore we use $\ell_{ei}(\cdot, \cdot)$ (Eq. (6.4)). We augment all training sets with horizontally flipped versions of the originals.

Table 6.1 summarizes our results for MSRC-9. We find that we perform best when using RO+MS+CL, with 90.2% accuracy, improving on the previous result of Grangier et al. (2009) by 1.7%. Section 6.1.2 shows the learned class location filter for MSRC-9. The network learned e.g., that “aeroplanes” are horizontally extended objects and that to the left and right of cows, the probability for “building” is low. As demonstrated in Section 6.1.2, this filter helps to remove spurious detections where the classifier was unsure. Further example segmentations are displayed in Fig. 6.5.

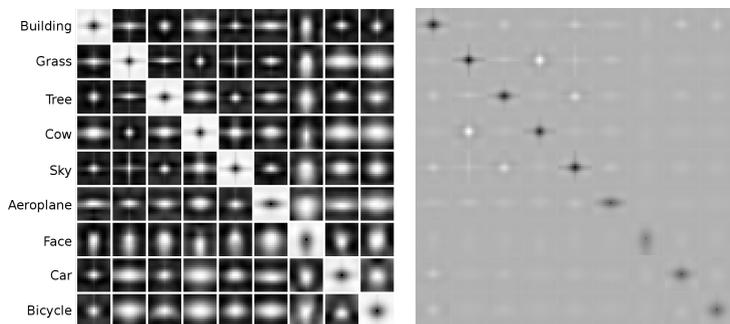


Figure 6.3: Class location filters (CL) learned for MSRC-9. Black represents more positive weights. Each prior is normalized separately (right: all normalized together).

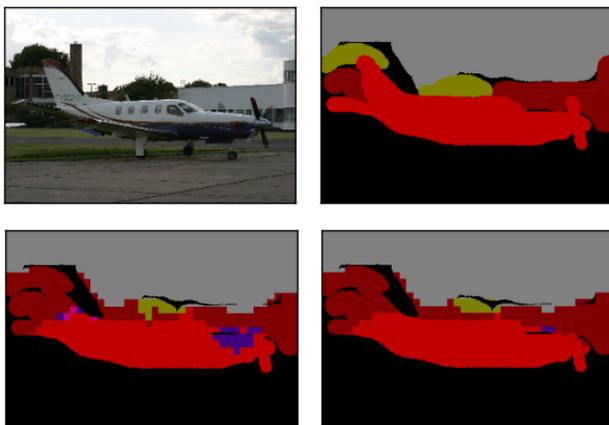


Figure 6.4: CL repairs spurious observations. *Top row*: original and ground truth; *bottom row*: before and after CL. Filters are normalized separately for every pair of classes.

Table 6.1: MSRC-9 object class segmentation database results

Condition	Accuracy (%)
RO+CL+MS	90.2
RO+CL	89.8
regular	89.1
no pretraining	87.9
Grangier et al. (2009)	88.5

For MSRC-21, we analyze the advantages of the class location filter as we vary the number of layers (Table 6.2). We find that CL improves the result of the lower layers. The effect diminishes, however, when resolution becomes too small (22×22 for 4 Layers) relative to the size of the class location filter. While our best result (80.2%) does not achieve state of the art, it improves upon Gould et al. (2008) who proposed the relative location prior which inspired the CL extension. Without pretraining, we measure only 73.9% accuracy, which emphasizes its importance.

On IG02, we find that CL does not help as much (Table 6.3). We attribute this to the structure of the dataset (only one class per image, few classes in total). Using RO only, we can already improve upon the previously best result (Fulkerson et al., 2009) in two out of three classes.

SPEED COMPARISON On a NVidia GTX 580, we process 16 images at once. A batch forward pass through the best-performing network on IG02 takes 0.23 s, which amounts to approximately 70 images per second. When preprocessing is performed on-line—e.g. as in Section 6.2.1—it can be executed concurrently to the forward pass on CPU. Our naïve HOG implementation takes 0.04 s/image at input resolution, while the ZCA whitening transform amounts to a convolution and therefore takes less time than the forward pass. Consequently, we expect framerates of more than 10 fps, which is an order of magnitude faster than e.g. speed-

Table 6.2: MSRC-21 object class segmentation database results

Condition	Accuracy (%)	
	RO	RO+CL
1 Layer	45.4	49.4
2 Layer	71.0	76.9
3 Layer	76.0	77.7
3 Layer+MS	77.9	80.2
4 Layer	77.4	77.8
Gould et al. (2008)	70.1	77.8
Ladicky et al. (2009)	86.0	

optimized work by Aldavert et al. (2010) (cf. their results in Table 6.3).

6.1.3 Related Work

While a large body of research focuses on object class segmentation, most notably associated with the Pascal VOC challenge¹, only few works have attempted to use a neural network architecture before the contents of this section were published.

Jain and Seung (2008) use supervised pre-training for a denoising task and show that their approach has relations to the optimization of a Markov random field. This work differs from ours in the task (regression vs. classification), and in the model architecture. The denoising task is learned from small (6×6) patches, which provide sufficient context. Our convolutional architecture with pooling creates a much larger receptive field for the output units. A large context is essential for non-trivial semantic segmentation tasks.

Mnih and Hinton (2010) also argue for larger context. The authors use unsupervised pre-training for a network that classifies

¹ <http://pascallin.ecs.soton.ac.uk/challenges/VOC/>

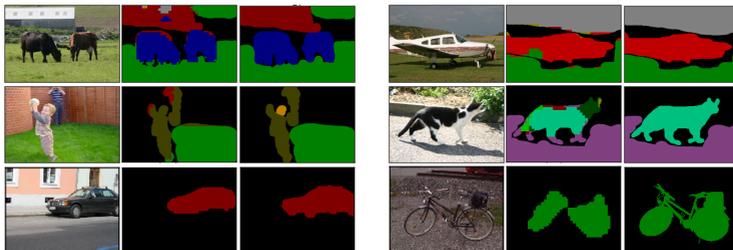


Figure 6.5: Example segmentations for MSRC-9 (top row), MSRC-21 (center row), IG02 (bottom row). Left column shows original image, center column our output, right column ground truth. MSRC-9 and MSRC-21 have ground truth void mask superimposed.

Table 6.3: INRIA Graz-02 object class segmentation database results

Condition	Precision/recall at equal error rate		
	Car	Bike	Person
RO	75.6	74.7	61.1
RO+CL	75.6	74.8	61.1
regular	74.4	74.2	61.9
Fulkerson et al. (2009)	72.2	72.2	66.3
Aldavert et al. (2010)	62.9	71.9	58.6

roads in aerial images. They then combine multiple single-pixel predictions in a post-processing step. A similar two-step procedure was used by Cireřan et al. (2012c) for segmenting cell membranes in medical images. In contrast to Mnih and Hinton (2010), we use the same supervised pretraining algorithm throughout our architecture with increasing context sizes, while continuing to train all pre-trained parameters. The road classification dataset used by Mnih and Hinton (ibid.) is not publicly available.

Gould et al. (2008) introduced the idea of a relative location prior in conditional random fields, addressing the problem that certain classes have different probabilities of co-occurrence de-

pending on their relative spatial location. We draw on their idea by explicitly learning filters that reduce errors of previous class predictions.

Finally, Grangier et al. (2009) used an architecture similar to ours. From their brief description, many details of the algorithm remain unclear. In contrast to their approach, we propose to use multi-scale inputs, reuse the pretraining predictions. We show that these extensions consistently improve the results, and directly compare our method with theirs on the MSRC-9 dataset.

In contrast to most neural networks for image processing, we use densely extracted image features (HOG) to complement ZCA-transformed color channels. Recently, HOG was shown to yield superior results over deformable parts models when used as input to a CNN (Ren et al., 2015).

Recently, Gupta et al. (2014) and Eigen and Fergus (2015) used transfer learning to address object class segmentation. They initialized the weights of their network with a network trained on a supervised large-scale image classification task (ILSVRC) and proceeded using a similar step-wise pre-training technique as presented here. In this chapter, we focus on networks learned only on the dataset that they are evaluated on.

6.1.4 Conclusion

We presented a convolutional neural network architecture for object class segmentation, which achieves state-of-the-art performance on common vision datasets. Crucial factors for good performance are supervised pretraining and class location filters. The network can be parallelized well on GPU and exhibits very good recall times.

6.2 ENCODING DEPTH INFORMATION FOR CNN

In the previous chapter, we have seen that CNN achieve good results on object class segmentation. To further improve on this task, it is essential to correctly identify object borders. Modern depth cameras can make the task easier, since changes in depth

are often co-occur with object boundaries. The depth gradient can provide useful cues to the object type. However, the depth information needs to be incorporated into existing techniques. In this section, we demonstrate how depth images can be used in a convolutional neural network for scene labeling by employing a simplified version of the HOG descriptor to the depth channel (HOD). We train and evaluate our model on the challenging NYUD dataset and compare its classification performance and execution time to state of the art methods.

6.2.1 *Methods*

NETWORK ARCHITECTURE We train the four-stage convolutional neural network, illustrated in Fig. 6.2 for object class segmentation. In contrast to Section 6.1, we use rectifying nonlinearities $\sigma(x) = \max(x, 0)$ after convolutions. This non-linearity improves convergence (Krizhevsky et al., 2012) and results in more defined boundaries in the output maps than sigmoid nonlinearities.

PRE-PROCESSING The main difference to previous work lies in the pre-processing. To increase generalization, we generate variations of the training set. This is performed online on the CPU while the GPU evaluates the loss and the gradient, at no cost in speed. We randomly flip the image horizontally, scale it by up to $\pm 10\%$, shift it by up to seven pixels in horizontal and vertical direction and rotate by up to $\pm 5^\circ$. The depth channel is processed in the same way. We then generate three types of input maps.

From random patches in the training set, we determine a whitening filter that decorrelates RGB channels as well as neighboring pixels. Subtracting the mean and applying the filter to an image yields three zero phase (ZCA) whitened image channels.

On the RGB-image, we again compute a computationally inexpensive version of HOG introduced in Section 6.1.1. Additionally, we perform the same operation on the depth channel, resulting in a CNN compatible version of the histogram of oriented depths (HOD) descriptor. (Spinello and Arras, 2011).

All maps are normalized to have zero mean and unit variance over the training set. The process is repeated for every scale, where the size is reduced by a factor of two. For the first scale, we use a size of 196×196 . The teacher maps are generated from ground truth by downsampling, rotating, scaling, and shifting to match the network output. We use an additional ignore map, which sets the loss to zero for pixels which were not annotated or where we added a margin to the image by mirroring. Sample maps and segmentations are shown in Fig. 6.7.

6.2.2 Experiments

The NYUD dataset (Silberman and Fergus, 2011) is comprised of video sequences taken from 464 indoor scenes, annotated with a total of 894 categories. We use the popular relabeling into four high-level semantic categories: small objects that can be easily carried (prop), large objects that cannot be easily carried (furniture), non-floor parts of the room: walls, ceiling, columns (structure), and the floor of the room (floor).

We split the training data set into 796 training and 73 validation images. In a stage s , we use the RMSProp learning algorithm with an initial learnrate 10^{-4} , to train the weights of all stages below or equal to s . The active stage is automatically switched once the validation error increases or fails to improve. The pixel mean of the classification error over training is shown in Fig. 6.6. During the first two stages, training and validation error behave similarly, while in the final stages the network capacity is large enough to overfit.

CLASSIFICATION PERFORMANCE To evaluate performance on the 580 image test set, we crop the introduced margins, determine the pixel-wise maximum over output maps and scale the prediction to match the size of the original image. There are two common error metrics in the literature, the average pixel accuracy and the average accuracy over classes, both of which are shown in Table 6.4. Our network benefits greatly from the introduction of depth maps, as apparent in the class accuracy increase from 56.1 to 62.0. We compare our results with the architecture of Couprie

et al. (2013), which is similar but computationally more expensive. While we do not reach their overall accuracy, we outperform their model in two of the four classes, furniture and, interestingly, the rather small props—despite our coarser output resolution.

We also compare HOD processing to the simpler depth input used by Couprie et al. (2013). Couprie et al. use local mean and contrast normalization, resulting in a single depth map input. This way of using the depth results in a performance drop of three percentage points.

Reevaluating our choice of using ZCA instead of RGB inputs, we find that RGB-only input results in an error of $E_{\text{pix}} = 54.0\%$ and $E_{\text{cls}} = 54.8\%$, while providing only ZCA inputs result in significantly better $E_{\text{pix}} = 55.7\%$ and $E_{\text{cls}} = 56.1\%$.

PREDICTION SPEED We can also attempt to compare the time it takes to process an image by the network. Couprie et al. (ibid.) report 0.7s per image on a laptop. We process multiple images in parallel on a GPU. In contrast to Section 6.1, we implement pre-processing asynchronously in CUVNET. With this asynchronous pre-processing, our performance saturates at a batch size of 64, where we are able to process 52 frames per second on a 12 core Intel Xeon at 2.6 GHz and a NVidia GeForce GTX TITAN GPU. Note that this is faster than the frame rate of the sensor collecting the dataset (30 Hz). While the implementation of Couprie et al. (ibid.) could certainly also profit from a GPU implementation, it requires more convolutions as well as expensive superpixel averaging and upscaling operations. Our network is also faster than random forests on the same task (30.3 fps (Müller and Behnke, 2014), hardware similar to ours).

6.2.3 *Related Work*

Our work builds on the architecture proposed in Schulz and Behnke (2012c), which (in the same year as Farabet et al. (2012)) introduced neural networks for RGB scene segmentation. We improve on their model by employing rectifying non-linearities, recent learning algorithms, online pre-processing, and a novel method to provide depth modality inputs.

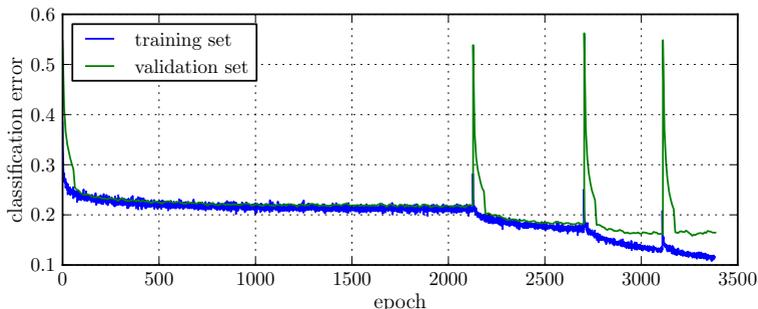


Figure 6.6: Classification error on NYUD during training, measured as the mean over output pixels. The peaks and subsequent drops occur when one stage is finished and learning proceeds to the next—randomly initialized—stage.

Scene labeling using RGB-D data was introduced with the NYU Depth V1 dataset by Silberman and Fergus (2011). They present a CRF-based approach and provide handcrafted unary and pairwise potentials encoding spatial location and relative depth, respectively. These features improve significantly over the depth-free approach. In contrast to their work, we use learned filters to combine predictions. Furthermore, our pipeline is less complex and achieves a high framerate. Later work (Silberman et al., 2012) extends the dataset to version two, which we use here. Here, the authors also incorporate additional domain knowledge into their approach, which further adds to the complexity.

Coupric et al. (2013) present a neural network for scene labeling which is very close to ours. Their network processes the input at three different resolutions using the same network structure for each scale. The results are then upsampled to the original size and merged within superpixels. Our model is only applied once to the whole image, but uses inputs from multiple scales, which involves less convolutions and is therefore faster. Outputs are also produced at all scales, but instead of a heuristic combination method, our network learns how to use them to improve the final segmentation results. Finally, the authors use raw depth as input to the network, which cannot be exploited easily by a con-

Table 6.4: Classification Results on the NYUD

Method	Accuracy (%)					
	Per-class				Average	
	floor	struct	furnit	props	pixel	class
Ours without depth	69.1	57.8	55.7	41.7	56.2	56.1
Ours with depth	77.9	65.4	55.9	49.9	61.1	62.0
Couprie et al. (2013) without depth	68.1	87.8	51.1	29.9	59.2	63.0
Couprie et al. (2013) with depth	87.3	86.1	45.3	35.5	63.5	64.5
RGB only	67.0	61.1	49.6	38.4	54.0	54.8
ZCA only	69.2	62.7	50.4	40.7	55.7	56.1
ZCA, HOG, SD	71.6	62.8	52.3	44.6	57.7	57.8

Controls in lower part of table compare RGB-only and ZCA-only condition. Last line is full model where HOD is substituted with depth pre-processing of Couprie et al. (2013).

volutional neural network, e.g., absolute depth is less indicative of object boundaries than are relative depth changes.

A common method for pixel-wise labeling are random forests (e.g. Sharp, 2008; Shotton et al., 2013; Schulz et al., 2016), which currently provide the best results for RGB-D data (Stückler et al., 2014; Müller and Behnke, 2014). These methods scale feature size to match the depth for every image position, an idea which we will explore for CNNs in Section 6.3.

6.2.4 Conclusion

We presented a convolutional neural network architecture for RGB-D semantic scene segmentation, where the depth channel is provided as feature maps representing components of a simplified histogram of oriented depths (HOD) operator. We evaluated the network on the challenging NYUD dataset and found that introducing depth significantly improved the performance

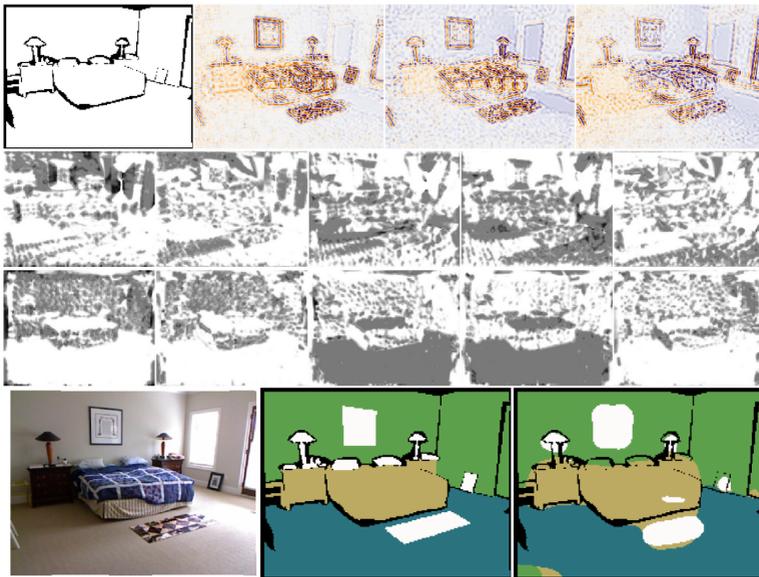


Figure 6.7: Network maps, inputs and outputs. First row, ignore mask and ZCA-whitened RGB channels. Second and third row, HOG and HOD maps, respectively. Fourth row, original image, ground truth and network prediction multiplied by ignore mask for reference. Mirrored margins are removed for presentation to save space. Note that HOG and HOD encode very different image properties with similar statistics.

of our model, resulting in competitive classification performance. We also find that HOD performs better than locally contrast-normalized depth input.

In contrast to other published results of neural network and random forest-based methods, our GPU implementation is able to process images at a high framerate of 52 fps.

6.3 DEPTH AND HEIGHT AWARE SEMANTIC PERCEPTION

An important property of convolutional neural networks (CNN) is their invariance to translations. This invariance property is created by sharing weights between all image locations, and by pool-

ing over nearby locations. For many tasks in computer vision, translation invariance is desirable, but some object classes (e. g. sea, sky) are more likely to appear in certain scene regions. Furthermore, depending on the distance to the camera, the same objects can also appear at different scales, but scale invariance is missing in the CNN architecture. There are two commonly applied solutions to the problem.

We might present the input to the network at multiple scales during training, such that the network can learn the invariance. This option requires large models and sufficient data, since we need to learn the same task independently at all scales.

If object annotations are available, we can scale objects to a uniform size before presenting them to the CNN. To produce a prediction for a novel image, the network has to be applied to every image position at multiple scales, and the results must be combined. This approach is called sliding window.

In this section, we propose a third option which relies on the availability of depth for the image pixels (RGB-D). The required dense depth estimates are produced e.g. by affordable consumer cameras such as the Microsoft Kinect, which has become popular in computer vision (Shotton et al., 2011; Bo et al., 2013) and robotics (Newcombe et al., 2011). We propose to use depth information in CNNs as follows:

1. We train the network on image patches with a size chosen proportional to the depth of the patch center. Since training is scale invariant, we can afford smaller models and make more efficient use of training data.
2. For novel images, we propose a sampling scheme which covers the image with overlapping patches of depth-adjusted size. Thus, closer image regions are processed at a large scale, while far-away regions are processed at a small scale. This automatic adjustment is more efficient than sliding window, because the scale is chosen automatically. In contrast to a multi-scale sliding window, our scale adjustments are continuous.
3. Finally, we propose to use height above ground as an additional input to the CNN. Height is an important clue and

quite distinct from the distance from the camera. E.g., floor pixels might occur at any distance from the camera, but they always have zero height.

We evaluate our method on the NYUD, which contains indoor scenes annotated according to their object class and find that both height annotation and depth normalization significantly improve CNN performance.

6.3.1 *Related Work*

Depth normalization of features has been proposed in the context of random forests (Lepetit et al., 2005; Shotton et al., 2011) by Stückler et al. (2012). The binary features in their work consist of region average differences, where both region sizes and distances to the query pixel are scaled with the depth. In this work, we scale image patches, not features. While this requires more computation during preprocessing, it allows for more expressive features.

Hermans et al. (2014) and Stückler et al. (2014) use random forests as a baseline and aggregate video information over time with self localization and mapping (SLAM). Here, we focus on single image prediction, which is comparable to random forest learning.

Using height for indoor scene object class segmentation was introduced by Müller and Behnke (2014) and Gupta et al. (2014). Müller and Behnke use the output of a random forest, merge the predictions within superpixels and learn a conditional random field (CRF) which has access to the average superpixel height. In contrast to their work, we incorporate height into the base classifier, which we then use to directly improve the unary term of their CRF.

Couprie et al. (2013) and Höft et al. (2014) train CNNs for object class segmentation using depth information, with very different approaches. Couprie et al. (2013) train three CNNs with shared weights on three scales. The upsampled results are then combined to yield output maps corresponding to object class labels. Thus, in contrast to our proposed method, the image is always

Table 6.5: Network architecture used for this section

Layer	#Parameter	Filter Size	Stride	#Maps	Map Size
Input	–	–	–	8	64×64
Conv1	12 576	7×7	1	32	64×64
Pool1	–	2×2	2	32	32×32
Conv2	50 208	7×7	1	32	32×32
Pool2	–	2×2	2	32	16×16
Conv3	6304	7×7	1	4	16×16

trained and evaluated on all three scales. The label probabilities are then averaged within superpixels of an oversegmentation. Superpixel averaging is compatible with our approach and further improves our performance.

Höft et al. (2014) also use a CNN with a multi-scale approach, but treat scales differently. Larger scales have access to predictions from smaller scales and can modify them. Treating scales differently can be justified by the fact that in indoor scenes, certain objects (dressers, beds) are typically much larger than others (vases, television sets), and need more context to be recognized. Note that while in this work, we use only one scale for every patch, it is also possible to use a multi-scale approach, where all scales are depth-adjusted simultaneously.

6.3.2 Methods

NETWORK ARCHITECTURE We use the simple feed forward convolutional architecture shown in Table 6.5, with interleaved convolutional max-pooling layers, and rectification (ReLU) nonlinearities. With less than 70 000 parameters in total, it is a very small network (cf. Couprie et al., 2013; Höft et al., 2014). While performance might improve with size and better regularization (i. e. dropout), we would like to emphasize that depth and height

awareness allows to reduce the number of parameters significantly.

COVERING WINDOWS We choose patch sizes s in the original image inversely proportional to the depth $d(\mathbf{x}_c)$ of a selected patch center \mathbf{x}_c , with $s = \gamma/d(\mathbf{x}_c)$. The parameter γ is set to 300 pxm throughout this paper, such that enough context is provided to the network, and receptive field sizes are similar to the scale of the random forest features by Stücker et al. (2014). The patch is then scaled to the input dimension of the CNN with bilinear interpolation. If parts of the patch are outside the original image, we extend it by reflection on the border. Due to irregular patch sizes, a sliding window approach with fixed strides would sample too densely in regions with shallow depth or too coarsely in far-away regions. Instead, we simply ensure that the patches cover the image. We sequentially sample patch centers \mathbf{x}_c from a multinomial distribution, with probabilities

$$p(\mathbf{x}_c) \propto \begin{cases} 0 & \text{if } \mathbf{x}_c \in \bigcup_{w \in W} w \\ d(\mathbf{x}_c) & \text{else,} \end{cases} \quad (6.6)$$

where W is the set of patches sampled so far. Depth-proportional probabilities encourage that far regions are covered by their corresponding small patches before near regions are covered by large patches. The greedy approach — selecting the patch with the largest distance which is not covered — results in a significantly larger number of patches due to the increased overlap (115.4 vs. 75.5 patches per image on average), while classification precision does not change significantly.

When predicting, we use bilinear interpolation to upsample the network output to the original patch size and accumulate the predictions for all image patches. We use radially decreasing weights $r(\|\mathbf{x} - \mathbf{x}_c\|)$ in the accumulation, since the depth normalization is strictly valid only for the patch center.

INPUT FEATURES We use eight input maps: The raw RGB channels, four containing a simplified histogram of oriented depth (Höft et al., 2014), and one map for the height. The height

map is computed by extracting normals in the depth images, finding ten clusters with k -means and determining the cluster that is most vertical. All points are projected to this normal and the height of the lowest point is subtracted. From all input maps, we subtract the dataset mean and ensure that maps have similar variances.

TRAINING PROCEDURE During training, we select patch centers \mathbf{x}_c randomly, determine their size, and add small distortions (rotations of up to 5° , scalings of up to 5%, and horizontal flipping). CNN weights are initialized randomly from $\mathcal{U}(-0.01, 0.01)$. We use a batch size of 128 and an initial learnrate of 0.001, with a momentum of 0.9 and exponentially decreasing learnrate schedule. We optimize pixel-wise weighted multinomial logistic loss over the output maps, with weights

$$w(\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{x} \text{ is not annotated} \\ 0 & \text{if } \mathbf{x} \text{ is outside the original image} \\ \frac{r(\|\mathbf{x}-\mathbf{x}_c\|)}{p(c(\mathbf{x}))} & \text{else,} \end{cases} \quad (6.7)$$

where $p(c(\mathbf{x}))$ is the prior probability of the class \mathbf{x} is annotated with.

6.3.3 Experiments

As in the previous section, we use the NYUD (Silberman et al., 2012) for our evaluation. We train our network for object class segmentation on the indoor scenes using the default split of 795 training and 654 testing images. We focus on the four semantic structural classes floor, structure, furniture, and prop. An additional void class resembles regions not annotated and is excluded from the evaluation.

Our results are summarized in Table 6.6. We compared our method with other state-of-the-art neural networks as well as methods which, for comparability, do not use transfer learning, extensive post-processing through CRFs or aggregation over time. We trained four models: a baseline method only using cover-

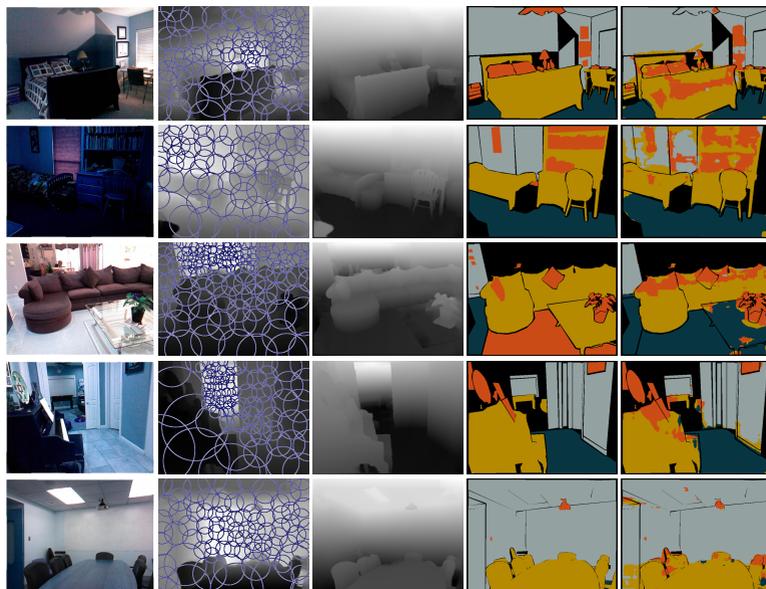


Figure 6.8: Sample segmentations from test set using the CW+DN+H model. Left to right: Original image, depth with patches denoted by circles, height above ground, ground truth and prediction.

Table 6.6: Results on NYUD

Method	Per-class accuracies (%)				Average acc. (%)	
	floor	struct	furnit	props	Pixel	Class
CW	84.6	70.3	58.7	52.9	66.6	65.4
CW+DN	87.7	70.8	57.0	53.6	67.3	65.5
CW+H	78.4	74.5	55.6	62.7	68.1	66.5
CW+DN+H	93.7	72.5	61.7	55.5	70.9	70.5
CW+DN+H+SP	91.8	74.1	59.4	63.4	72.2	71.9
CW+DN+H+CRF	93.5	80.2	66.4	54.9	73.7	73.4
RF+CRF *	94.9	78.9	71.1	42.7	71.9	72.3
RF+H*	91.2	81.0	71.6	22.7	69.6	66.5
RF *	90.8	81.6	67.9	19.9	65.0	68.3
Couprie et al. (2013)	87.3	86.1	45.3	35.5	63.5	64.5
Höft et al. (2014)	77.9	65.4	55.9	49.9	61.1	62.0
Silberman et al. (2012)	68.0	59.0	70.0	42.0	59.6	58.6

CW is covering windows, H is height above ground, DN is depth normalized patch sizes. SP is averaged within superpixels and SVM-reweighted. RF is random forest, CRF is a conditional random field over superpixels (Müller and Behnke, 2014). Structure class numbers are optimized for class accuracy. * Müller and Behnke (2014)

ing windows (CW), two with added depth normalization and height (CW+DN and CW+H, respectively), and a combined model (CW+DN+H). When training without depth normalization, we use the average patch size found by the depth normalization (135 px). We find that our combined model improves significantly over the other methods in terms of class average and pixel accuracies. The height feature contributes more to the overall improvement than depth normalization, but both ideas seem to complement each other.

Finally, our predictions can be used as input to high-level methods, such as super-pixel averaging (CW+DN+H+SP) and conditional random fields (CW+DN+H+CRF). We use method and implementation of Müller and Behnke (2014), and find that class and

pixel average accuracies improve by more than one percentage point when using our CNN predictions in place of their globally optimized random-forest predictions.

Sample segmentations, as well as patch size and height visualizations, are shown in Fig. 6.8. Note that the network sometimes overgeneralizes (carpet is labeled as floor, not prop in row 3), but generally identifies floor/not-floor well even in images where no or little floor is visible and our simple height extraction algorithm fails (row 5).

6.3.4 Conclusion

We proposed two extensions for convolutional neural networks which exploit depth information: (i) covering windows which are scaled by the depth of their center and (ii) height-above-ground input maps. Our evaluation on the NYUD shows that the proposed approach can outperform other neural network and random forest methods. In future work, we plan to extend our method with multi-scale depth-normalized processing.

6.4 RECURRENT NETWORKS FOR DEPTH PERCEPTION

Deep neural networks compute increasingly abstract features, which simultaneously become more and more semantically meaningful, and incorporate larger contexts. A real-world vision system will have to deal with the time dimension as well. Content is increasingly generated in the form of videos by Internet users, surveillance cameras, cars, or mobile robots. Video information can be helpful, as looking at a whole sequence instead of single frames may enable the interpretation of ambiguous measurements.

Similar to increasingly abstract features on images, we are interested in neural networks which produce high-level features on sequences. In a recursive computation, these high-level features should help to interpret the next frame in a sequence. In addition to a semantically meaningful localized content description, such

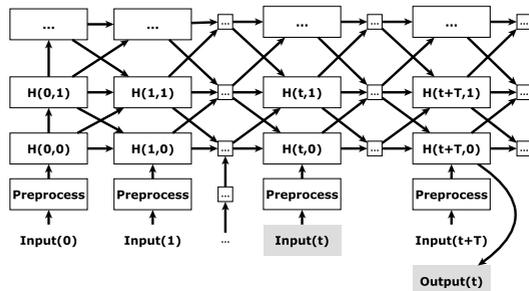


Figure 6.9: Architecture of our RNN. The layers are connected to each other with valid convolutions. Upward (*forward*) connections additionally include spatial max-pooling operations, while downward (*backward*) connections include a spatial upsampling. Delay D is number of time frames between an input and corresponding output frame.

features should form high-level descriptions of motions with increasing temporal context.

Recurrent neural network (RNN) lend themselves naturally to sequence processing, especially the architecture proposed by Behnke (2003b), which produces pixel level outputs and can be used for object class segmentation.

In this section, we introduce a recurrent convolutional neural network architecture which produces high-level localized sequence features for object class segmentation. In contrast to previous sections—where we used single frames only—we train our approach on sequences from the NYUD.

In short, our contributions are as follows:

- We introduce a recurrent convolutional neural network model for processing image sequences.
- On toy datasets, we show that our recurrent models are able to keep an abstract state over time, track and interpret motion, and retain uncertainty.
- We show that our model improves RGB-D object class segmentation classification accuracy on the challenging NYUD dataset when compared to other CNN models. When combined with a CRF, RNN performance is close to carefully

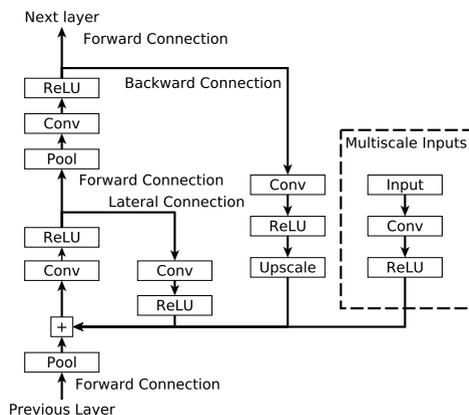


Figure 6.10: Recurrent connections as viewed from a single hidden layer. Activations of a hidden layer are the result of forward connections from below, backward connections from above, and lateral connections from the same layer at previous time steps. Inputs may be provided at all scales.

tuned transfer-learning approaches initialized on much larger datasets.

- We analyze the obtained networks and show that the improved performance of our network stems from the recurrent processing combined with the exploitation of temporal, i.e. video, information.

The remainder of this section is organized as follows. Section 6.4.1 introduces our architecture and learning methods. In Section 6.4.2, we discuss related work. We evaluate our model in Section 6.4.3, and discuss the results in Section 6.4.4.

6.4.1 Model Description

In this section, we present our network architecture. It is inspired by the Neural Abstraction Pyramid of Behnke (ibid.), a hierarchical recurrent neural network architecture for image processing. A schematic overview is shown in Figs. 6.9 and 6.10. We use con-

volutional neural networks (CNN (LeCun et al., 1998a)), which retain the topological image structure and ensure that features are localized. Our base configuration (without the ability to process sequences) is a fairly standard small CNN with $L = 3$ convolutional layers, ReLU non-linearities, and interleaved spatial max-pooling.

In contrast to the Neural Abstraction Pyramid of Behnke (2003b), which doubles the number of filters as the resolution of the representation is halved, we keep the number of filters to a constant 32 on all layer of the network. Initial experiments on our dataset have shown that doubling the number of filters leads to a too complex model, which takes longer to learn and overfits early.

Connection Types

To process sequences, we replicate our model for T time steps and introduce connections between the temporal copies. Three types of connections exist: forward, lateral, and backward. Computationally, all connections are valid convolution operations followed by a half-rectifying point-wise non-linearity $f(z) = \max(0, z)$.

A hidden layer $H(t, l)$, at time step t and abstraction level l , is connected to layer $H(t + 1, l + 1)$ using a forward connection. These connections allow the vertical flow of information from the bottom of the network to the top and thus the construction of high level features based on the low level features. The non-linearity of the forward connections are followed by a spatial 2×2 maximum pooling operation with stride 2.

Lateral connections connect layers $H(t, l)$ and $H(t + 1, l)$. These horizontal connections can incorporate immediate spatial context from the same activation level. The intermediate context is limited by the receptive field size of the convolution filters.

Backward connections connect layer $H(t, l)$ to layer $H(t + 1, l - 1)$, and can be interpreted as providing a high-level prior for the lower, more detailed layers. Since higher layers have a coarser spatial resolution, they also provide a convenient shortcut for infor-

mation that needs to travel long distances. Backward connections are immediately followed by a spatial upsampling operation.

Due to padded convolutions and the opposing pooling and upsampling operations, all connections coinciding on a given hidden layer have the same size, and are simply summed element-wise.

We use a convolutional feature extraction layer between the inputs and the RNN, whose weights are initialized using symmetry k means (Konda et al., 2013). Since this unsupervised initialization prevents us from controlling the gain, we ensured that these weights are not part of a feedback loop. For similar reasons, the weights of the first forward pass are excluded from weight sharing as well.

All other hidden-to-hidden connections use temporal weight sharing, i.e. for all t and all $k \in \{-1, 0, 1\}$, the weights used in the convolution from $H(t, l)$ to $H(t + 1, l + k)$ are identical across time steps.

Output in the Bottom Layer

In contrast to common CNN models, the output of our network is always obtained in the lowest layer of the network. This structural property allows us to produce detailed outputs at input resolution. We add a time delay between input and output, ensuring that the inputs from the last presented frame were able to reach the topmost layer and return to the bottom layer before evaluating the loss.

A final convolution without temporal weight sharing is used to extract one map per target class from the bottom layer. The cross-entropy loss is computed perpixel over all output maps.

Recurrent Processing

We process the images of videos sequentially, one image per time step. The state (i.e., the activations) at time t containing information about its past is combined with the image at time t , producing an output and a new state. Since the last output benefits from learning from the whole sequence, it is natural to place the frame that we want to evaluate at the end.

The first temporal copy is special, since it contains regular feed forward connections. This allows us to produce activations in each layer such that all connection types can be used in the transition from t to $t + 1$.

NETWORK DEPTH When processing input at time t , we allow $L - 1$ time steps for the information to reach the top level of the network and the same amount for propagating back to the bottom layer, where the output corresponding to time t is evaluated. Note that the last temporal steps do not need all the hidden layers, since their activation would no longer propagate to the output.

Our RNN is trained with backpropagation through time (BPTT), and can be interpreted as a very deep non-recurrent net after unfolding. In this non-recurrent network, multiple paths lead to the output, with the shortest path—from input $t = T$ to the final output—having only length $2L - 1$, and the longest $2L + t$, which amounts to a depth of 14 layers for our $L = 3, T = 8$ network.

WEIGHT INITIALIZATION AND OPTIMIZATION We initialize the weights biases from a Gaussian distribution. It is important to ensure that the activations do not explode or vanish early during training. Ideally, activations in the first forward pass should have similar magnitudes. This is difficult to control, however. Instead, we choose the standard deviation of the weights for each layer l according to the scheme proposed by He et al. (2015):

$$\sigma = \sqrt{\frac{2}{k_l^2 \cdot d_{l-1}}}, \quad (6.8)$$

which takes into account the filter size k_l and the number of filters of the last layer d_{l-1} . We determine the mean of the bias such that the average of the activations in every point of our network is positive and slightly decreasing over time. Liang and Hu (2015) use local contrast normalization at all layers to the same effect, which requires more GPU memory for the hidden layer activations. Due to our larger inputs and outputs and the increased number of time steps, current GPU memory restrictions prevent us from doing the same.

We learn the parameters of our network with backpropagation through time (BPTT) using RMSP_{Prop}, which is a variant of resilient backpropagation (RPROP, Riedmiller and Braun 1993) suitable for mini-batch learning (Dauphin et al., 2015). RPROP and RMSP_{Prop} to a large degree consider only the sign of the gradient, thus being robust against vanishing and exploding gradients, both common phenomena in RNN training.

During learning, we apply dropout (Srivastava et al., 2014). Combining dropout with RNN is delicate, however. If it affects recurrent connections, their ability to learn long-range dependencies suffers (Pham et al., 2014). Thus, we add a convolution with non-shared weights to extract the output from the state and apply dropout here, and find that it consistently improves our results.

6.4.2 *Related Work*

Several groups have used DNNs to process image sequences. Most works use stacks of frames to provide time context to the neural network. Le et al. (2011) and Taylor et al. (2010) learn hierarchical spatio-temporal features on video sequences using Gated Convolutional Restricted Boltzmann Machines (convGRBM) and independent subspace analysis (ISA), respectively. Their image features are not learned discriminatively and the models do not allow localized predictions.

Simonyan and Zisserman (2014) use a two-stream architecture for action recognition, which separately creates high-level features from single-frame content and motion. Motion is provided through a stack of optical flow images, so that the modeled complexity is limited by the stack size. In our experiments, we found that increasing temporal context by providing frames consecutively yields improved performance.

More recently, Michalski et al. (2014) introduced a model designed to explicitly represent and predict the movement of entities in images. The architecture is chosen in a way that higher layers represent invariances of derivatives of positions (motions, accelerations, jerk). Our models do not explicitly model motion. However, our models can make use of deep layers even in the case no high-level position invariances exist, since in addition

to motion, they also encode static content. Furthermore, in our model, deep layers have a lower resolution and facilitate transport of information across longer distances.

Jung et al. (2014) introduce a multiple timescale recurrent neural network for action recognition, which uses neurons with fixed time constants. The model uses leaky integrator neurons, which limits the rate at which higher layer activations can change. It is trained and evaluated on a simplified version of the Weizmann Human Action Recognition dataset.

Various architectures for processing video data are explored by Karpathy et al. (2014). The architecture most similar to our model, *slow fusion*, uses weight sharing between time steps and merges them in higher layers. In their study, *slow fusion* yields best results. In contrast to classifying video sequences with a single label, we produce label output at pixel level.

RNN were successfully used for object class segmentation by Graves (2012) and Pinheiro and Collobert (2014). Both works use recurrence only to increase spatial context, whereas we extend processing to the temporal domain.

Object recognition is another task where RNN achieved state of the art results. In a recent work by Liang and Hu (2015), use Convolutional Layer unfolded in time similar to ours. Their architecture consists of a stack of five such layers interleaved with pooling and dropout layers. Similarly to previous works, Liang and Hu (ibid.) only use static information. Nevertheless, their model obtained superior results classifying images of multiple datasets.

long short-term memory (LSTM) units are capable of carrying information, at the original resolution, over long temporal distances. LSTM is especially is often used in speech recognition (Graves et al., 2013) or language understanding (Sundermeyer et al., 2012), where e.g. a specific property of a distant word or sound might influence the semantics of the current context. In this paper, we opt for simple neural units instead. While we are also interested in learning long-range dependencies, we do not provide spatial or temporal context at the original resolution. Instead, our architecture maintains expressive low resolution context information in higher layers. This is more realistic for

natural images, where correlations are stronger between nearby pixels than those between distant ones. It also allows for sparser connectivity between units, since temporally distant units do not need to be wired.

Recent work of Bogun et al. (2015) uses LSTM units for object recognition in video sequences. They obtained state of the art results on the Washington Dataset (Lai et al., 2011a) by incorporating information from several frames. Their most successful strategy was to train the network unidirectionally and to use a bidirectional model, based on the same set of weights, during prediction.

Pascanu et al. (2013) suggest that LSTM also addresses the problem of vanishing gradients. Here, we use RMSProp as gradient method, which—in addition to preventing vanishing gradients—also counteracts gradient explosion.

Our architecture choices are motivated by the neural abstraction pyramid (NAP) of Behnke (2003b), which performs pixel-wise classification tasks at input resolutions as well. In contrast to our work, Behnke did not train on video sequences, but only on stationary patterns, which in some cases were corrupted by temporally changing noise. We also include modern architectural features, such as max-pooling and ReLU non-linearities, dropout, and use RMSProp to increase learning speed.

6.4.3 *Experiments*

We first conduct experiments on handcrafted datasets, which allow us to demonstrate important characteristics of our model. In a second step, we use our architecture for object class segmentation on the already introduced NYUD dataset.

Toy Experiments

We present three toy experiments, showing that our network is able to learn 1) filtering noisy information over time, 2) tracking and interpreting motion, and 3) retaining an internal state including uncertainty.

Table 6.7: Denoising Results for different models

Method	Accuracy
RNN	93.3
CNN	84.7
Thresholded Average	81.6

DENOISING In this experiment, we feed different degraded versions of the same binary image to the network. We use salt and pepper noise, uniformly distributed over the whole image. We also draw random black or white lines, to make the task more difficult. The task is to obtain the original image without noise. One way the network could solve this task would be to learn to average the image over time (which is our baseline). In addition, denoising filters learned by the neural network can remove high frequency noise.

To ensure that the network is able to generalize instead of learning an input by heart, we use different objects for training, validation and testing. Every split contains 100 independently generated sequences.

Since the task has a low complexity, we opt for a simple convolutional model of only one hidden layer with 32 maps. A small filter size of 5×5 provides sufficient spacial context. There is no specific order in such a sequence of noised images, thus we only test the unidirectional architecture on this task.

We use $T=6$ temporal copies. During training, we optimize a weighted sum of the losses at all time steps, with a ten times larger weight placed on the final output. In all toy examples, we train for 12 000 iterations with minibatches of size 16.

Fig. 6.11 shows an example from the test set. Our model is able to improve its prediction step by step, accumulating over time information even from the areas which are more affected by noise. After only two steps, the network is able to remove most of the false positives and to assemble together almost all features of the object. Table 6.7 and Fig. 6.11 (bottom) show that the RNN

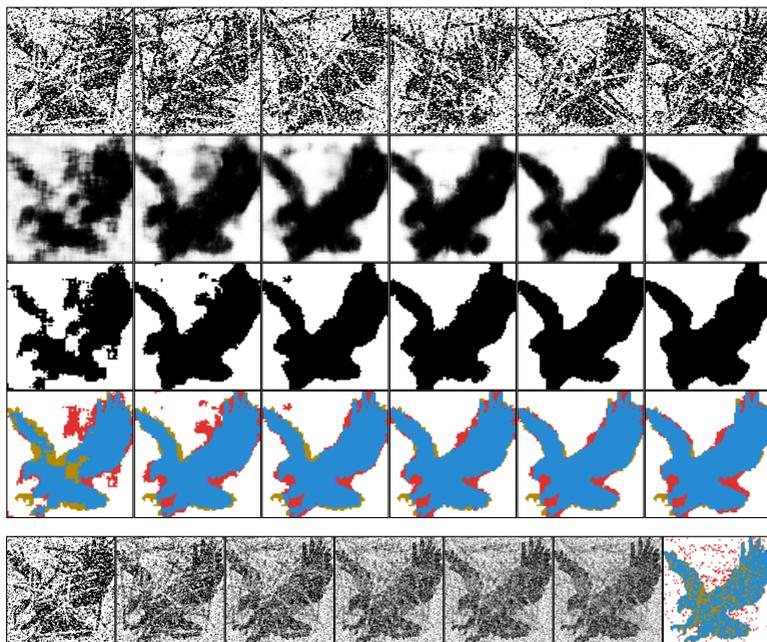


Figure 6.11: Toy experiment: Denoising. Top: Rows represent, in order: the RGB input of the network for each time-step, the output of the softmax layer, the final outputs of the network, the evaluation (● True Positives ○ True Negatives ● False Positives ● False Negatives). The last output is used for evaluation of pixelwise classification error. Bottom: Average of the inputs presented to the network.

performance compares favorably to a naïve thresholding of the average image.

We also train the single-timestep base model on the same dataset. The images (not shown) produced by this model have less articulate contours, resulting in a classification accuracy of 84.7% network compared to 93.2% of the recurrent model, outperforming the naïve averaging approach only by a small margin.

DETECTING MOVEMENT In this experiment, we test the capabilities of the network to track a foreground object while the object is moving with constant speed through a noisy image. To

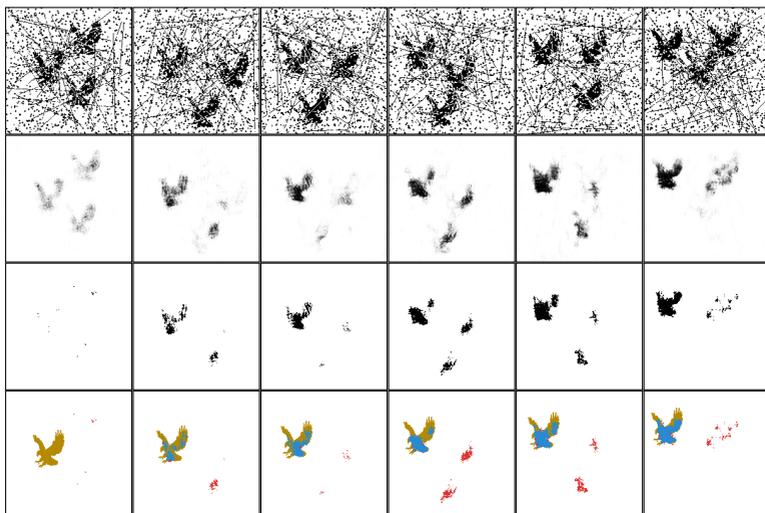


Figure 6.12: Toy Experiment: Detecting movement. Rows represent, in order: the RGB input of the network for each time-step, the output of the softmax layer, the final outputs of the network and the evaluation (● True Positives ○ True Negatives ● False Positives ● False Negatives). Rightmost output used to determine classification error.

ensure that motion is the cue for tracking, we add two randomly placed distractor objects of the same shape and size in a random position at every time step. These distractors should be classified as background. To prevent the network from overfitting on motion direction and speed, we generate several sequences, each moving the object from a random position to another, with varying speed.

Fig. 6.12 shows the results obtained on this task using the unidirectional network. In the first time step, the network cannot decide which object is moving continuously. Already at $t=2$, however, the network detects a slight positional change from one of the objects, while the others are further away from their initial position. The softmax layer activations (second row) show that the certainty of the hypothesis increases step by step. Also, one can notice that more details are added to the representation. Some

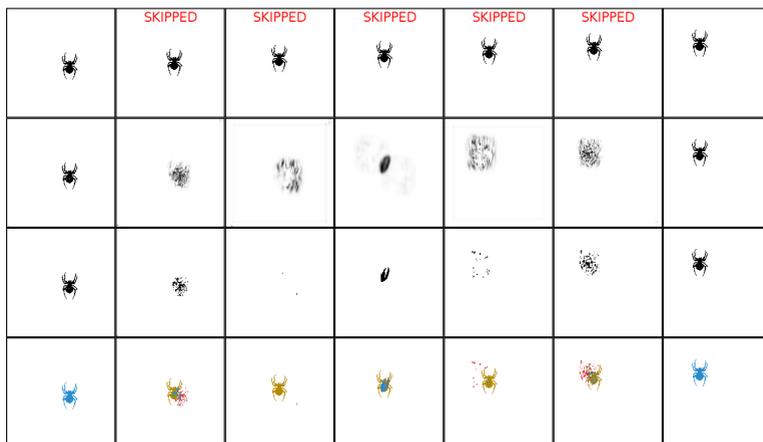


Figure 6.13: Toy experiment: Retaining uncertainty. Rows represent, in order: the RGB input of the network for each time-step, the output of the softmax layer, the final outputs of the network and the evaluation (● True Positives ○ True Negatives ● False Positives ● False Negatives). Center output of the bidirectional network is used to determine classification error.

false positives still exist when the new random position of a distractor object is close to its former position.

RETAINING UNCERTAINTY While in the previous experiments, we showed that the network is able to track a moving object, we now consider if a regular movement can be inferred from temporally distant information. We use a bi-directional version of our model (with weights shared between the past and future network parts) and provide only the first and the last input, so that the initial positions have to be remembered until information from the past and future converges at the center time step. Since denoising is not an essential component of this task, we do not add noise.

Fig. 6.13 depicts a sample sequence from the test set. As no motion information is provided, the best strategy of the network is to create a circular expanding hypothesis from the seen location, which then collapses as both timelines are combined. This

is what we observe in the output maps of Fig. 6.13. While the position is correctly identified, the shape of the object is largely lost.

Experiments on NYUD

As in earlier sections, we concentrate here on experiments on NYUD (Silberman et al., 2012). In contrast to these sections, we use the full video dataset, not just the few labeled frames. This change entails extensive preparations, and a closer understanding how the dataset was created.

NYUD is comprised of video sequences taken from 464 indoor scenes, annotated with a total of 894 categories. We use the popular relabeling into four high-level semantic categories as before: small objects that can be easily carried (prop), large objects that cannot be easily carried (furniture), non-floor parts of the room: walls, ceiling, columns (structure), and the floor of the room (floor).

Although NYUD was recorded as a video sequence, a subset of 1449 frames which were preprocessed and manually labeled is more prevalent in the literature. The remainder—407 024 frames—consists of raw RGB-D camera images.

To transform the dataset into an image sequence dataset, but at the same time use the labeled frames for evaluation, we extract the past and future context of each labeled frame from the video stream and preprocess it similar to the labeled frames. For evaluation, we compare outputs corresponding to the labeled frame with the ground truth, retaining the same training/testing split as in the literature and in the previous sections.

To preprocess the RGB and depth images, we follow standard procedure, sequentially applying lens correction², projecting the depth readings into the RGB sensor frame, and filling-in missing depth readings (Levin et al., 2004).

To obtain ground truth information for unlabeled frames in our sequences, we propagate labels along optical flow directions. Due to manual labeling of the dataset, classes are often separated by small unlabeled regions especially at edges, which are crucial

² computed by OpenCV's "Camera Calibration and 3D Reconstruction" package

to the optical flow estimation. To address this problem, we use a colorization method (ibid.) to determine missing labels close to labeled regions. Only the training set is modified, which allows direct comparison with other methods on the test set. After label fill-in, we compute optical flow (Farnebäck, 2003) on RGB image pairs. When label information is unavailable or ambiguous due to limitations in the optical flow computation, we write ignore labels, which are excluded from the computation of the loss and its gradient.

We also use optical flow to select which images are included in a sequence. Fast displacements are tricky to detect (Brox and Malik, 2011), while too slow motion results in quasi-static images, foiling our attempt to exploit the temporal context. Since the dataset was recorded at 30 Hz, taking every frame would result in no visible motion at input resolution. We thus decide to add a new image to the sequence once the optical flow shows, on average, a motion larger than half of our filter size.

Not all sequences in the dataset contain sufficient temporal context. In such cases, we complete them by replicating the first or last available frame.

Learning

We train the network with depth $L = 3$, an input resolution of 160×160 pixels, using minibatches of size 16, and a temporal context of 8 frames. As input, we use histogram of oriented gradients (HOG) and histogram of oriented depths (HOD) channels, ZCA filtered images, estimated height above ground (c.f. Section 6.3), and the optical flow. We use randomly chosen 10% of the training set for validation (early stopping and model selection). While images are randomly transformed during training, we use a fixed, randomly picked set of transformations for validation to ensure stable estimates. Ground truth is provided at times $t=3, 6$, and 8 (c.f. Fig. 6.15). Training continues for 12 000 iterations, with an initial learnrate of $3 \cdot 10^{-4}$. The learnrate was automatically decreased three times when the validation error failed to improve. Once learning stopped, we retained the model which obtained the best performance on the validation set.

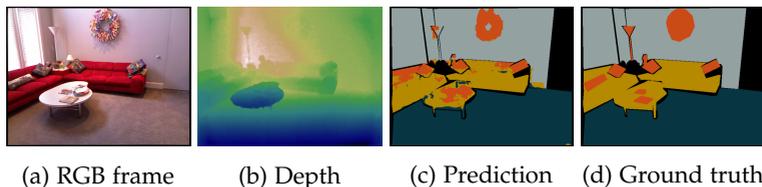


Figure 6.14: Prediction for one of the NYUD dataset frames. Images (a) and (b) show RGB and depth, respectively, after being preprocessed. (c) and (d) represent the prediction and ground truth, respectively, where color codes floor (●), prop (●), furniture (●), structure (●) and unknown (●). The network detects most of the pixels correctly, even some wrongly labeled ones (e.g. the third object on the table and the center of the wall-mounted piece).

DEPTH-NORMALIZATION We integrate depth-normalized covering windows (Section 6.3) into our learning algorithm. This approach evaluates the model on image patches at a spatial resolution which is dependent on the distance of the center pixel to the observer, effectively building scale invariance and adaptive output resolution into the model. Similar to Section 6.3, this patch-based approach allows us to use smaller maps, which speeds up the training dramatically (days vs. weeks). As input, we used 80×80 instead of 160×160 pixels, and scaled hidden layers accordingly. At prediction time, the network has to process patches that fully cover the original image and combine their prediction afterwards. Thus speed depends on the number of patches, which in turn depends on the depth distribution of the RGB-D image.

Comparison with state of the art

Table 6.8 shows our result with and without depth-normalized covering windows (CW) together with state-of-the-art results on the same dataset. We compare against other methods which do not use transfer learning (e.g. by initializing filters using ImageNet-based transfer learning). Our RNN (W+CW) outperforms all other published approaches. Overall, depth-normalization (CW) turns out to be worse for RNN, maybe indi-

Table 6.8: Comparison of NYUD classification performance with other state-of-the-art approaches without transfer learning. Our model with enabled covering windows (CW), unsupervised weight initialization (WI), and conditional random fields (CRF). Input consists of color/depth information, optionally with height (H). Baselines use convolutional neural networks (CNN), random forests (RF), and self-localization and mapping (SLAM).

Method	Class Accuracies (%)				Average (%)	
	ground	struct	furnit	prop	Class	Pixel
ours (CW)	95.8	74.6	54.2	64.0	72.1	68.6
ours (WI+CW)	94.9	76.8	65.5	60.8	74.5	73.1
ours (WI)	94.3	83.7	72.0	54.9	76.2	76.4
ours (WI+CW+CRF)	95.4	78.9	67.3	60.8	75.6	74.6
ours (WI+CRF)	94.2	83.9	72.0	56.3	76.6	77.2
Schulz et al. (2015a) (CNN+CRF)	93.6	80.2	66.4	54.9	73.7	73.4
Müller et al. (2014) (RF+CRF)	94.9	78.9	71.1	42.7	71.9	72.3
Stückler et al. (2014) (RF+SLAM)	90.8	81.6	67.9	19.9	65.0	68.3
Coupric et al. (2013) (CNN)	87.3	86.1	45.3	35.5	63.5	64.5
Silberman et al. (2012) (RF)	68.0	59.0	70.0	42.0	59.6	58.6

cating that there is not enough context to be interpreted in the chosen window sizes. However, CW is much better at identifying small objects (prop), at up to 64% vs. 54.9% for full frames.

The outputs of our RNN can still be improved through other means. Instead of determining a pixel-wise maximum, we supply the probabilities as data term to the conditional random field (CRF) of Müller and Behnke (2014). The result (WI+CW+CRF and WI+CRF) improves performance even further. This indicates that our predictions are better suited for CRF postprocessing than CNN (Coupric et al., 2013; Schulz et al., 2015a) and random forests

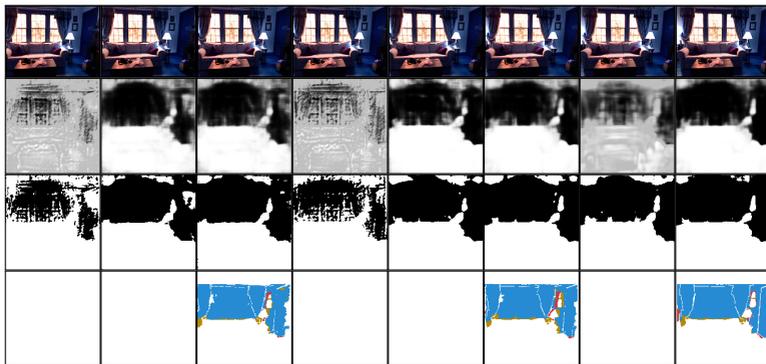


Figure 6.15: Prediction for one sample of NYUD test dataset. Rows represent, from top to bottom: the RGB input, the softmax layer output, the output of the network; and the evaluation (● True Positives ○ True Negatives ● False Positives ● False Negatives) for the class structure.

(Silberman et al., 2012; Müller and Behnke, 2014; Schulz et al., 2016).

COMPARISON TO TRANSFER LEARNING Similar to non-recurrent neural networks (e.g. Erhan et al., 2010b), our RNN profits from a weight initialization (CW vs. WI+CW). Supervised weight initialization using transfer learning on the large ImageNet database led to the (to the best of our knowledge) current top result on NYUD by Eigen and Fergus (2015), with a class accuracy of 79.1%. Our RNN only narrows the gap to the transfer learning approach. Weight initialization for recurrent connections is, therefore, an open but promising research direction.

DOES TEMPORAL CONTEXT HELP? Except for Stückler et al. (2014), none of the publications in Table 6.8 made use of temporal context to determine class labels. We would like to know whether our improvement is due to the fact that we use additional training data (albeit without labels), or through recurrent processing.

To check whether our network takes advantage of temporal context, we perform a static frame experiment. We use the same

frame as input at all time steps during training and prediction. In this setting, the recurrent architecture is still able to learn long-range spatial dependencies, but cannot exploit temporal context, which results in accuracy reduction in both class accuracy and pixel-wise accuracy (2.7 and 5.5 percentage points, respectively) relative to the model which has access to temporal context. This model still outperforms non-recurrent models, showing that spatial recurrent processing is the essential improvement.

When using static frames as above, we remove both, temporal context and additional ground truth generated through optical flow. Removing only additional ground truth during training results in a reduction of the class accuracy by 2%, while the pixel-wise accuracy decreases by 3.1%. These controls suggest that the superior accuracy of the RNN is mainly due to recurrent spatial and temporal processing, and intermediate ground truth is important to help this very deep—and mostly uninitialized—network learn.

6.4.4 *Conclusion*

In this work, we introduced a recurrent convolutional neural network architecture, which in addition to learning spatial relations is also able to exploit temporal relations from video. We started with a series of toy examples that showed that our networks are able to solve tasks that require denoising, detecting movement, and retaining uncertainty.

We further carried out experiments on sequences of indoor RGB-D video sequences from the NYUD dataset. Combined with dropout, unsupervised weight initialization, covering windows and conditional random fields, our proposed model improves performance when compared to other non-recurrent baseline models and random forests, obtaining close to the state of the art RGB-D segmentation results.

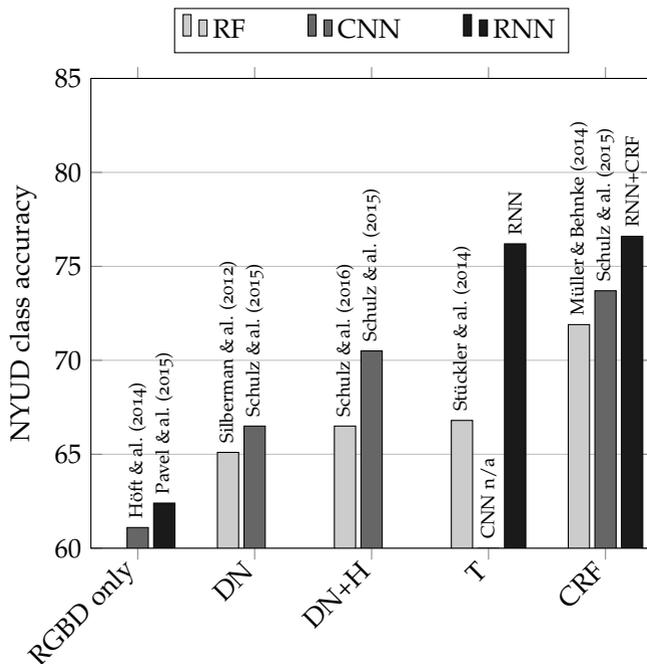


Figure 6.16: Summary of object class segmentation results on NYUD. DN is depth normalization, H is height above ground input, T is time. CRF is with post-processing using conditional random fields.

6.5 CHAPTER SUMMARY

In this chapter, we introduced a number of models, training techniques and pre-processing methods for object class segmentation. The presented techniques successively improve performance over competing methods. Figure 6.16 summarizes the overall progress on the NYUD dataset.

STRUCTURED PREDICTION FOR OBJECT DETECTION

After great success in image classification, neural network research has recently turned to detecting instances of object categories in images. Here, the task is to predict a bounding box $y \in \mathbb{R}^4$ for every object of a given class. Object detection differs from image classification in one main aspect—the solution space is huge. In fact, it contains any number of bounding boxes with position, size, and aspect ratio. Correctness is typically defined by measuring overlap with the ground truth, such that more than one correct solution exists. Under these conditions, simply scaling the multinomial logistic loss, which worked for classification of the 1000-class ImageNet dataset, is not an option. Instead, a number of competing approaches have been proposed.

In this section, we shortly review recently published methods for object detection with deep neural networks and emphasize that they, successfully, optimize heuristic surrogate loss functions. These surrogate loss functions are related to the overlap criterion via an accompanying post-processing step, which is not part of the training algorithm.

We then apply an alternative method based on structured prediction (Lampert, 2011; Lampert et al., 2009), which optimizes the overlap criterion directly. Similar to Chapter 6, we create a model that predicts values for all image pixels, reflecting whether they are part of a bounding box or not. In contrast to Chapter 6, the output is immediately interpreted by an additional operator that efficiently produces bounding box predictions. These predictions are then used in a structured loss function to determine gradients. We evaluate our proposed method on two selected classes of the VOC 2007 dataset.

7.1 STRUCTURED PREDICTION FOR OBJECT DETECTION

We start with a deep convolutional neural network, without fully-connected layers. The number of maps in the output layer is equal to the number of detectable object-classes in the dataset. Our goal will be to produce values in the output map from which the bounding boxes in the image can be inferred. For inference, which is also part of our learning procedure, we make use of the fact that the space of bounding boxes is *structured*, i.e. can be searched efficiently. In the following description, we draw heavily on Lampert (2011) and Lampert et al. (2009), with the main difference that instead of hand-crafted support vector machine (SVM) features, our predictions are based on features learned by the CNN.

We are given a training set of tuples

$$\mathcal{D} = (\mathbf{x}^i, Y^i)_{i=1, \dots, n} \subset \mathcal{X} \times \mathbb{P}(\mathcal{Y}), \quad (7.1)$$

where \mathbf{x} is an image as usual, Y is a set of bounding boxes, \mathcal{Y} contains all possible bounding boxes in \mathbf{x} , and $\mathbb{P}(\mathcal{Y})$ is the powerset of \mathcal{Y} . The prediction function $g(\mathbf{x}) : \mathcal{X} \mapsto \mathbb{P}(\mathcal{Y})$ should minimize the empirical loss over the training set

$$\langle \Delta(g(\mathbf{x}), Y) \rangle_{(\mathbf{x}, Y)}, \quad (7.2)$$

where $g(\mathbf{x}) =: \hat{Y}$ is the inference function. The evaluation criterion for detection is typically given as a 50% threshold on the Jaccard index of the prediction and a ground truth object,

$$A(\hat{y}, y) = \begin{cases} 0 & \text{if } \frac{\text{area}(\hat{y} \cap y)}{\text{area}(\hat{y} \cup y)} > \frac{1}{2} \\ 1 & \text{otherwise.} \end{cases} \quad (7.3)$$

We also penalize the case where not all objects in an image have been identified, or more objects were returned than present in the ground truth Y . This is formalized in the set loss $\Delta : \mathbb{P}(\mathcal{Y}) \times \mathbb{P}(\mathcal{Y}) \mapsto \mathbb{R}$. Lampert (2011) suggests to use the max loss,

$$\Delta(Y, \hat{Y}) = \max_{y \in Y \ominus \hat{Y}} \lambda(Y, y), \quad (7.4)$$

where $Y \ominus \hat{Y}$ is the symmetric set difference and

$$\lambda(Y, y) = \begin{cases} 1 & \text{if } y \in Y \\ \min_{\bar{y} \in Y} A(\bar{y}, y) & \text{else.} \end{cases} \quad (7.5)$$

Expanding the symmetric set difference and simplifying slightly, we get

$$\Delta(Y, \hat{Y}) = \max \left(\min(1, |Y \setminus \hat{Y}|), \max_{y \in \hat{Y} \setminus Y} \min_{\bar{y} \in Y} A(\bar{y}, y) \right). \quad (7.6)$$

The first term in the outer maximum accounts for objects in the ground truth for which no corresponding detection $\hat{y} \in \hat{Y}$ exists. The second term penalizes predictions which are not corresponding to ground truth objects. One possible problem here is that we could find one object as many times as there are objects in the image, using slightly different \hat{y} that all overlap with one y . To prevent multiple detection, we require that elements of \hat{Y} have a maximum Jaccard index of 0.2, which can be enforced by greedily rejecting non-matching bounding boxes in the sequential inference procedure.

Next, we define a compatibility function f between a neural network output map $N(\mathbf{x})$ and the bounding boxes $y \in Y$ over pixels i, j . A bounding box is a mask y on $N(\mathbf{x})$, where the rectangular part corresponding to the bounding box was set to one, and all other values to zero:

$$f(\mathbf{x}, Y, \theta) = \sum_{y \in Y} \sum_{ij} N_{ij}(\mathbf{x}, \theta) \cdot y_{ij}. \quad (7.7)$$

For learning, we would like to find parameters θ s.t.

$$g(\mathbf{x}) = \arg \max_{\hat{Y} \in \mathbb{P}(\mathcal{Y})} f(\mathbf{x}, \hat{Y}, \theta) \approx y. \quad (7.8)$$

For a given training tuple (\mathbf{x}^i, Y^i) , we can bound the loss using a hinge loss upper bound, as in Taskar et al. (2005), obtaining

$$\Delta(g(\mathbf{x}^i), Y^i) = \Delta \left(\arg \max_{Y \in \mathcal{P}(\mathcal{Y})} f(\mathbf{x}^i, Y, \theta), Y^i \right) \quad (7.9)$$

$$\leq \max_{\hat{Y} \in \mathcal{P}(\mathcal{Y})} \left(\Delta(\hat{Y}, Y^i) - f(\mathbf{x}, Y^i, \theta) + f(\mathbf{x}, \hat{Y}, \theta) \right) \quad (7.10)$$

$$= \max_{\hat{Y} \in \mathcal{P}(\mathcal{Y})} \underbrace{\left(\Delta(\hat{Y}, Y^i) + f(\mathbf{x}, \hat{Y}, \theta) \right)}_{H(N(\mathbf{x}^i), \hat{Y}, Y^i)} - f(\mathbf{x}, Y^i, \theta). \quad (7.11)$$

The role of the loss term is to ensure that the bounding boxes selected tend to have a bad detection measure, thereby forcing the network in $f(\cdot, \cdot, \cdot)$ to increase its margin over them.

The maximization in Eq. (7.11) can be performed as described in Lampert et al. (2009), using branch-and-bound on \mathcal{Y} . Branch-and-bound recursively splits \mathcal{Y} into subsets, which are described by two rectangles; o_{\cup} describes the maximum extents of all bounding boxes in the set, whereas o_{\cap} describes the minimum extents. For a given tuple (o_{\cup}, o_{\cap}) , we construct an upper bound $\overline{\Delta H}$ on the change in Δ caused by adding $\hat{y} \in (o_{\cup}, o_{\cap})$ to \hat{Y} ,

$$\max_{\hat{y} \in (o_{\cup}, o_{\cap}) \subset \mathcal{Y}} \left(H(N(\mathbf{x}), \hat{Y} \cup \{\hat{y}\}, Y) - H(N(\mathbf{x}), \hat{Y}, Y) \right) \quad (7.12)$$

$$\leq \overline{\Delta H}(o_{\cup}, o_{\cap}, N(\mathbf{x}), \hat{Y}, Y) \quad (7.13)$$

$$= F^+(N(\mathbf{x}), o_{\cup}) - F^-(N(\mathbf{x}), o_{\cap}) \quad (7.14)$$

$$+ \max \left(\min(1, \max_{\hat{y} \in (o_{\cup}, o_{\cap})} |Y \setminus (\hat{Y} \cup \{\hat{y}\})|), \right. \\ \left. \min_{\hat{y} \in Y} \bar{A}(\hat{y}, o_{\cap}, o_{\cup}) \right)$$

where

$$\bar{A}(y, o_{\cap}, o_{\cup}) = \begin{cases} 0 & \text{if } \frac{y \cap o_{\cup}}{y \cup o_{\cap}} > \frac{1}{2} \\ 1 & \text{else,} \end{cases} \quad (7.15)$$

$$F^+(N(\mathbf{x}), o_{\cup}) = \sum_{ij} \max(0, N_{ij}(\mathbf{x})) o_{\cup ij}, \quad \text{and} \quad (7.16)$$

$$F^-(N(\mathbf{x}), o_{\cap}) = \sum_{ij} \min(0, N_{ij}(\mathbf{x})) o_{\cap ij}. \quad (7.17)$$

The rectangle sums $F^{\pm}(\cdot, \cdot)$ can be efficiently evaluated by pre-computing two integral image for each network output map. A queue datastructure ensures that the search in \mathcal{Y} is efficient.

Intuitively, learning proceeds by minimizing sums in the rectangles which are found by branch-and-bound on an output map $N(\cdot)$ and maximizing the sum of ground truth bounding boxes. Since bounding boxes overlapping with ground truth are given a disadvantage in the loss-augmented inference of Eq. (7.11), training focuses on likely false detections—hard negatives—during optimization. Due to non-convexity of the objective, the CNN cannot maximize the margin directly. However, we found that the combination of weight decay and loss-augmented prediction had a positive effect on training. For non-loss-augmented inference—e.g., during prediction on the test set—the loss term is dropped from Eq. (7.11).

Note that in contrast to Szegedy et al. (2013), we do not need to specify on a per-pixel basis which value the output map should have, we only require the sums of regions to be higher or lower. The gradient consists of (differences of) rendered bounding boxes.

MULTI-CLASS TRAINING WITH WEAK LABELINGS In principle, all classes can be handled separately. Labels are typically weak, however, since not all objects are annotated. As commonly done, we only treat images I as negative for class c when no object of class c is annotated in I . Alternatively, it would be possible to use modified ground truth bounding boxes adjusted to have low overlap (Zhang et al., 2015, e.g.). If a ground truth bounding box y is matched by \hat{y} with a Jaccard index greater 0.5, we render the “negative” bounding box for \hat{y} to refine the position found and eliminate the gradient in the overlap region. Since we intend to increase the margin of ground truth bounding boxes over

Table 7.1: Network Architecture for learning two classes

Layer	Output Size	Filter Size	Channels/ Stride Groups	Pad	Pool Size	Pool Stride
Input	224×224	–	3 / –	–	–	–
Conv1	108×108	7	96 / 1	2	0	3
Conv2	49×49	5	256 / 6	1	0	3
Conv3	24×24	3	512 / 8	1	1	–
Conv4	24×24	3	512 / 16	1	1	–
Conv5	24×24	3	64 / 16	1	1	–
Conv6	22×22	3	2 / 1	1	0	–



Figure 7.1: Sample first-layer features of the classification network trained to discriminate between cow and horse images.

others in \mathcal{Y} , we train only on images or parts of images which contain at least one annotated object.

7.2 EXPERIMENTS

We present experiments on two difficult and easily confusable object classes, cow and horse. Our network architecture is shown in Table 7.1. It is roughly inspired by Krizhevsky et al. (2012), but optimized for larger output maps. It also lacks the fully connected layers, which significantly reduces its capacity.

We pre-train the model with binary classification between the two classes we want to detect, using 8193 images from the PASCAL-10X database (Zhu et al., 2012). For this purpose, we add two fully connected layers with 1024 hidden neurons and MaxOut non-linearity (Goodfellow et al., 2013b), each. The network is trained using ADAGRAD (Duchi et al., 2011) to adapt the learnrates. The

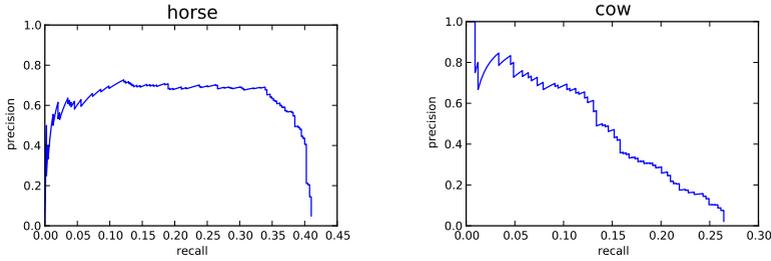


Figure 7.2: Precision and recall curves for the two detected object classes horse and cow on the VOC 2007 test set, with average precision of 0.295 and 0.149, respectively.

input images are scaled such that their shortest dimension is 256, then we extract a random crop of size 224×224 . The images are flipped horizontally with a probability 0.5. We perform this pre-processing in parallel on CPU, while the network runs on GPU. The network does not overfit on a held-out validation set (4466 images), but reduces the classification error on the training set to zero. ILSVRC pretraining is likely to improve the results even further. Figure 7.1 shows a subset of our first-layer features after pre-training.

In a second step, we train the network to detect objects on the same split of PASCAL-10X, using the methods described in Section 7.1. Again, we use AdaGrad for learnrate adaptation, and a learning rate of 0.01. Bounding boxes in the dataset are scaled down by a factor of two before supplying them to the network to aid discrimination between neighboring objects. During loss-augmented inference, we find up to four bounding boxes with a Jaccard index of at most 0.2 between any pair. Considering that only one object is annotated per image in PASCAL-10X, this is sufficient.

The time required for loss-augmented inference amounts to approximately $1/7$ of the network evaluation time (forward and backward pass) when performed on CPU without any parallelization. As input we use (possibly flipped) images from three different scales in steps of $1/2$ octaves. The largest scale is chosen such that the original image corners are in the center of the receptive

field of the corner output neurons. For this purpose, we add margin by mirroring parts of the image. Objects have to be at least two pixels wide when transformed to the output map, otherwise we do not use them as training examples on a given scale.

After convergence, we evaluate the network on the test set of the 2007 version of the Pascal Visual Object Classes Challenge (Everingham et al., 2010) (402 images containing either horses or cows). Here, we use a sliding window on all three scales and combine the outputs within one scale by a $\max(\cdot, \cdot)$ operation. We scale all maps to the same size. In contrast to training, sliding window and multiple scales results in many potentially overlapping bounding box candidates, so that a post-processing step becomes necessary. Following Szegedy et al. (2013), we use k -means clustering on the bounding boxes in the training set and determine 10 reference bounding boxes, which we then scale by factors of $\{0.1, 0.2, \dots, 0.9\}$. We slide all 90 bounding boxes over every scale and determine local maxima. Finally, we reduce the predicted set by removing bounding boxes which overlap with higher scoring ones by more than 20%. Figure 7.2 shows precision/recall curves for the two learned classes, and Figure 7.3 shows sample detections. The average precision for the horse class is comparable to Szegedy et al. (ibid.) and Erhan et al. (2014), however, both of these works trained on a complete VOC 2012 dataset.

7.3 RELATED WORK

Deep neural networks are increasingly applied to computer vision tasks such as image classification (Krizhevsky et al., 2012) or object-class segmentation (Schulz and Behnke, 2012c). Recent advances leading to success in the ImageNet challenge stem from *dropout* to prevent overfitting (Hinton et al., 2012), rectifying linear units for improved convergence, backpropagation through max-pooling (Scherer et al., 2010) and GPU implementations for speed, all of which are also used in this work.

Neural networks trained for an easy task can be adapted to more complex, but related tasks (Hinton and Salakhutdinov, 2006; Bengio et al., 2006). Consequently, neural net-based object detection methods start with networks trained for classification

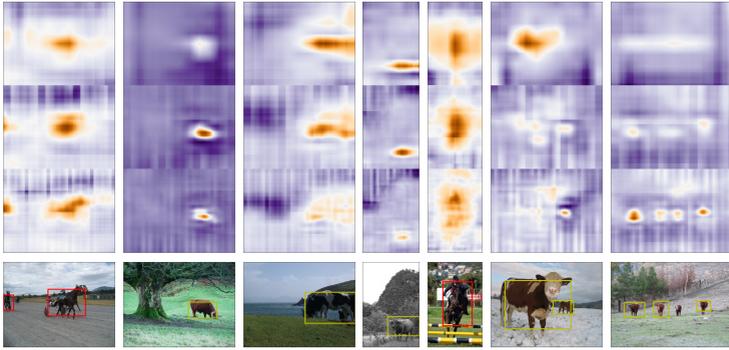


Figure 7.3: Sample object detections. The figure shows the output map activations for the respective class. There are three scales, the lower two are processed with sliding window and merged using $\max(\cdot, \cdot)$. The bottom row shows the input image with detected bounding boxes.

(Szegedy et al., 2013; Girshick et al., 2014), or learn classification at the same time as detection (Sermanet et al., 2013). Here, we also make use of the supervised pre-training technique, using only few images of the classes which also contain the objects we want to detect.

To detect Mitosis in medical images, Cireşan et al. (2013) used convolutional neural networks with per-pixel targets. This method only produces point locations, not bounding boxes of arbitrary size and aspect ratio.

Girshick et al. (2014) use a pipeline for detection with selective search region proposals (Uijlings et al., 2013), which are warped and classified by a deep neural network. The learned features are then used as input to a support vector machine and outputs are ranked. In contrast to our method, this practice relies heavily on pre-segmentation to find object candidates.

Sermanet et al. (2013) regress on bounding box coordinates and classification confidence from a low-resolution output map. The network is run on six scales and produces many bounding box candidates, which are then merged by a heuristic merging operation for prediction. While we also use a post-processing step,

we directly optimize bounding box overlap and the association of predicted bounding boxes with ground truth objects during learning.

Erhan et al. (2014) produce a fixed set of bounding boxes during training, which are associated to the ground truth bounding boxes to determine the gradient. This method is closest to ours. In contrast to them, we do not explicitly regress on bounding box coordinates and instead rely on output maps to represent the direct correspondence between image and bounding box (Long et al., 2014). This enables us to find objects even in positions which did not occur in the training set (see e.g. Zhang et al., 2015, for a discussion of this problem). Our method also does not output a fixed set of bounding box candidates per image, since we infer their number from the network output.

Finally, Szegedy et al. (2013) construct targets for low-resolution output-maps, which encode the bounding box and its object quadrants. A heuristic function combines the outputs to produce the final bounding boxes. This approach requires one network per class and five times as many output maps as classes, which poses a potential scaling problem. Also, it is not clear whether the network can or should spend effort on refining a regression, when the desired output is a bounding box with maximal overlap. Our proposed loss vanishes when the correct bounding box output is produced.

7.4 CONCLUSION

Following success on ImageNet classification, there is much attention on adopting deep convolutional neural networks to perform more complex computer vision tasks, especially object detection. Multiple formulations have been published in the time the presented work was submitted and evaluated. In contrast to these works, we produce output at pixel level, which is interpreted by a detector that is part of the loss function. We infer bounding boxes during training and minimize the overlap criterion directly, drawing heavily on previous work on structured prediction for support vector machines.

We evaluate our model on two difficult classes of the VOC 2007 dataset, cow and horse, pretrained by classification, and show that with our formulation, a deep neural network can learn to localize instances of the two classes well. While results on two classes are not conclusive, this proof-of-concept shows that learning bounding boxes with structured prediction is feasible in deep neural networks.

After the content in this section were published, Zhang et al. (2015) used a similar approach in a larger pipeline to fine-tune localizations by R-CNN networks (Girshick et al., 2014, and derivatives), improving VOC 2012 localization results.

TRANSFER LEARNING FOR RGB-D OBJECT RECOGNITION

The recent success of deep convolutional neural networks (CNN) in computer vision can largely be attributed to massive amounts of data and immense processing speed for training these non-linear models. However, the amount of data available varies considerably depending on the task. Especially robotics applications typically rely on very little data, since generating and annotating data is highly specific to the robot and the task (e.g. grasping) and thus prohibitively expensive. This section addresses the problem of small datasets in robotic vision by reusing features learned by a CNN on a large-scale task and applying them to different tasks on a comparably small household object dataset. Works in other domains (Girshick et al., 2014; Razavian et al., 2014; Donahue et al., 2014) demonstrated that this transfer learning is a promising alternative to feature design.

Figure 8.1 gives an overview of our approach. To employ a CNN, the image data needs to be carefully prepared. Our algorithm segments objects, removes confounding background information and adjusts them to the input distribution of the CNN. Features computed by the CNN are then fed to a support vector machine (SVM) to determine object class, instance, and pose. While already on-par with other state-of-the-art methods, this approach does not make use of depth information.

Depth sensors are prevalent in today's robotics, but large datasets for CNN training are not available. Here, we propose to transform depth data into a representation which is easily interpretable by a CNN trained on color images. After detecting the ground plane and segmenting the object, we render it from a canonical pose and color it according to distance from the object center. Combined with the image color features, this method outperforms other recent approaches on the WRGBDO dataset on a number of subtasks. This dataset requires categorization of

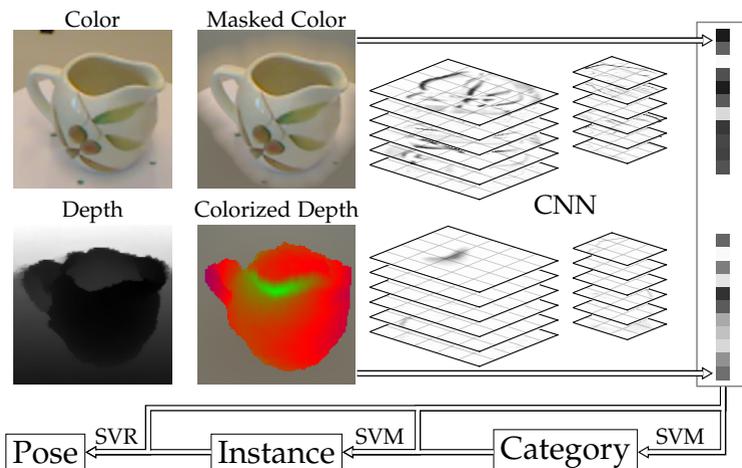


Figure 8.1: Overview of our approach. Images are pre-processed by extracting foreground, reprojecting to canonical pose, and coloring depth. The two images are processed by independently by a convolutional neural network. CNN features are then used to successively determine category, instance and pose.

household objects, recognizing category instances, and estimating their pose.

In short, our contributions are as follows:

1. We introduce a novel pre-processing pipeline for RGB-D images facilitating CNN use for object categorization, instance recognition, and pose regression.
2. We analyze features produced by our pipeline and a pre-trained CNN. We show that they naturally separate common household object categories, as well as their instances, and produce low-dimensional pose manifolds.
3. We demonstrate that with discriminative training, our features improve the state of the art on the WRGBDO classification dataset.
4. We show that in contrast to previous work, only few instances are required to attain good results.

5. Finally, we demonstrate that with our method even category level pose estimation is possible without sacrificing accuracy.

After discussing related work, we describe our feature extraction pipeline and supervised learning setup in Sections 8.2 and 8.3, respectively. We analyze the features and their performance in Section 8.4.

8.1 RELATED WORK

Deep convolutional neural networks (CNN, LeCun et al. 1998a; Riesenhuber and Poggio 1999; Behnke 2003b) became the dominant method in the ImageNet large scale image classification challenge (Russakovsky et al., 2014) since the seminal work of Krizhevsky et al. (2012). Their success can largely be attributed to a large amount of available data and fast GPU implementations, enabling the use of large non-linear models. To solve the task of distinguishing 1000, sometimes very similar object categories, these networks compute new representations of the image with repeated convolutions, spatial max-pooling (Scherer et al., 2010), and non-linearities (Krizhevsky et al., 2012). The higher-level representations are of special interest, as they provide a generic description of the image in an increasingly semantic feature space (Zeiler and Fergus, 2014). This observation is supported by the impressive performance of CNN features learned purely on classification tasks applied to novel tasks in computer vision such as object detection (Girshick et al., 2014; Razavian et al., 2014), sub-categorization, domain adaptation, scene recognition (Donahue et al., 2014), attribute detection, and instance retrieval (Razavian et al., 2014). In many cases, the results are on par with or surpass the state of the art on the respective datasets.

While Girshick et al. (2014) report improvements when fine-tuning the CNN features on the new task, this approach is prone to overfitting in the case of very few training instances. We instead follow Donahue et al. (2014) and only re-interpret features computed by the CNN. In contrast to the investigations of Don-

ahue et al. (ibid.) and Razavian et al. (2014), we focus on a dataset in a robotic setting with few labeled instances.

We use pre-trained CNN in conjunction with preprocessed depth images, which is not addressed by the works discussed so far. Very recently, Gupta et al. (2014) proposed a similar technique, where “color” channels are given by horizontal disparity, height above ground, and angle with vertical. In contrast to their method, we propose an object-centered colorization scheme, which is tailored to the classification and pose estimation task.

Previous work on the investigated RGB-D Objects dataset begins with the dataset publication by Lai et al. (2011a), who use a combination of several hand-crafted features (SIFT, Texton, color histogram, spin images, 3D bounding boxes) and compares the performance of several classifiers (linear SVM, Gaussian kernel SVM, random forests). These baseline results have been improved significantly in later publications.

Lai et al. (2011b) propose a very efficient hierarchical classification method, which minimizes classification and pose estimation jointly on all hierarchy levels. The method uses stochastic gradient descent (SGD) for training and is able to warm-start training when adding new objects to the hierarchy. While the training method is very interesting and could possibly be applied to this work, the reported results stay significantly behind the state of art.

Finally, Bo et al. (2013) show a very significant improvement in classification accuracy and reduction in pose estimation error. Their method learns hierarchical feature representations from RGB-D Objects data without supervision using hierarchical matching pursuit (HMP). This work shows the promise of feature learning from raw data and is the current state-of-the-art approach on the RGB-D Objects dataset. However, the number of training examples must be suitably large to allow for robust feature learning—in contrast to our work, which uses pre-learned features and is thus able to learn from few examples. Of course, the feature learning process is not needed in our case which leads to a significant runtime advantage for our method. Finally, our method generates less feature dimensions (10 192 vs. 188 300) and thus is also faster in the classifier training and recall steps.

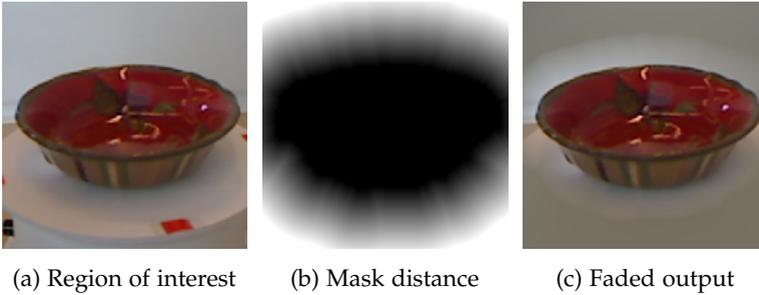


Figure 8.2: Overview of the RGB preprocessing pipeline, with ROI from object segmentation, distance transform from the object mask and faded output used for CNN feature extraction.

8.2 CNN FEATURE EXTRACTION PIPELINE

In order to use a pre-trained CNN for feature extraction, it is necessary to preprocess the input data into the format that matches the training set of the neural network. CaffeNet (Jia et al., 2014), which we use here, expects square 227×227 RGB images depicting one dominant object as in the ImageNet Large Scale Visual Recognition Challenge (Russakovsky et al., 2014).

8.2.1 RGB Image Preprocessing

Since the investigated CNN was trained on RGB images, not much pre-processing is needed to extract robust features from color images. Example images from the preprocessing pipeline can be seen in Fig. 8.2.

We first crop the image to a square region of interest (see Fig. 8.2a). In a live situation, this region of interest is simply the bounding box of all object points determined by the tabletop segmentation (Section 8.2.2). During evaluation on the WRGBDO dataset (Section 8.4), we use the provided object segmentation mask to determine the bounding box.

The extracted image region is then scaled to fit the CNN input size, in our case 227×227 pixels. To reduce the CNN's response to the background, we apply a fading operation to the image

(Fig. 8.2c). Each pixel is interpolated between its RGB color $\mathbf{c}_0 = (r_0, g_0, b_0)$ and the corresponding ILSVRC 2011 mean image pixel $\mathbf{c}_m = (r_m, g_m, b_m)$ based on its pixel distance r to the nearest object pixel:

$$\mathbf{c} := \alpha \cdot \mathbf{c}_0 + (1 - \alpha) \cdot \mathbf{c}_m, \quad (8.1)$$

where

$$\alpha := \begin{cases} 1 & \text{if } r = 0, \\ 0 & \text{if } r > R, \\ (R - r)^\beta & \text{else.} \end{cases} \quad (8.2)$$

The fade radius $R = 30$ was manually tuned to exclude as much background as possible while keeping objects with non-optimal segmentation intact. The exponent $\beta = 0.75$ was later roughly tuned for best cross-validation score in the category level classification.

8.2.2 Depth Image Preprocessing

Feeding depth images to a CNN poses a harder problem. The investigated CNN was trained on RGB images and is thus not expected to perform well on raw depth images. To address this, we render image-like views of the object from the depth data using a five-step pipeline, which will be detailed below. Figure 8.3 illustrates all steps for an example.

In the first step, we perform a basic segmentation to extract the horizontal surface the object is resting on. Following Holz et al. (2012), we estimate surface normals from the depth image, discard points from non-horizontal surfaces and register a planar model using Random Sample Consensus (RANSAC). The main objective here is to gain a local reference frame which is fixed in all dimensions, except for rotation around the surface normal. The planar model also allows us to find points on the plane and extract object clusters with Euclidean Clustering (see Fig. 8.3b).

In a second step, we fill-in holes in the depth map. We employ a common scheme based on the work of Levin et al. (2004), who

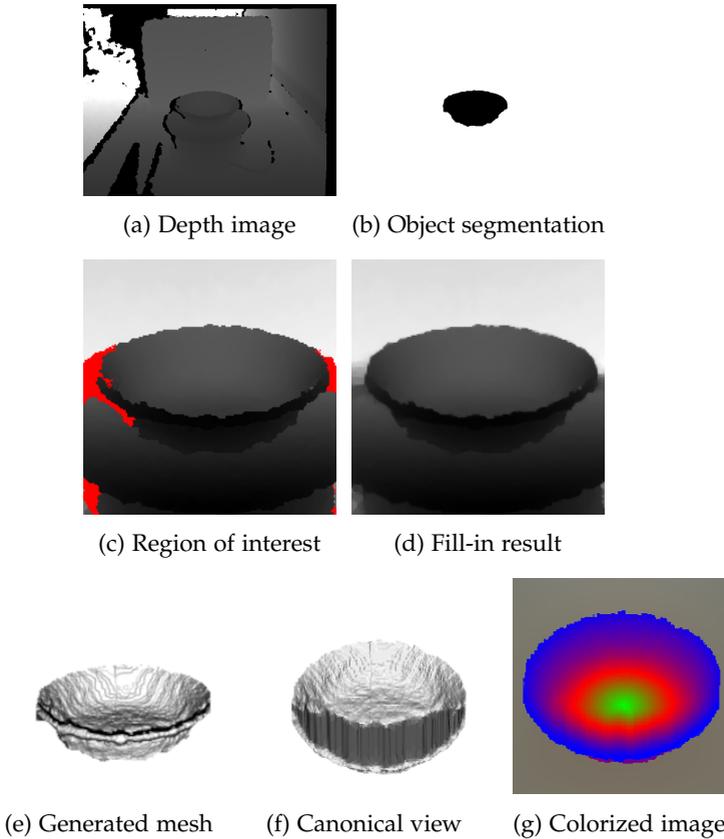


Figure 8.3: Overview of the depth preprocessing pipeline. (a) input depth map, (b) extracted segmentation mask after tabletop segmentation (Holz et al., 2012), (c) region of interest containing only the object with unavailable depth pixels shown in red, (d) missing depth values filled in, (e) mesh extracted from point cloud, (f) reprojection of mesh to a canonical camera pose, (g) final image used for CNN feature extraction.

investigated the colorization of grayscale images from few given color pixels. The colorization is guided by the grayscale image in such a way that regions with similar intensity are colored the same. We fill-in depth values using the same technique guided by a grayscale version of the RGB image. This has the advantage of using the RGB information for disambiguation between different depth layers, whereas the standard approach of median filtering the depth image cannot include color information.

In detail, the objective of the fill-in step is to minimize the squared difference between the depth value $d_{\mathbf{p}}$ with $\mathbf{p} = (u, v)$ and the weighted average of the depth at neighboring pixels,

$$J(\mathbf{d}) = \sum_{\mathbf{p}} \left(d_{\mathbf{p}} - \sum_{\mathbf{s} \in \text{neigh}(\mathbf{p})} w_{\mathbf{p}\mathbf{s}} d_{\mathbf{s}} \right)^2, \quad (8.3)$$

with

$$w_{\mathbf{p}\mathbf{s}} = \exp \left[-(x_{\mathbf{p}} - x_{\mathbf{s}})^2 / 2\sigma_{\mathbf{p}}^2 \right], \quad (8.4)$$

where $x_{\mathbf{p}}$ is the grayscale intensity at position \mathbf{p} and $\sigma_{\mathbf{p}}$ is the variance of intensity in a window around \mathbf{p} defined by $\text{neigh}(\cdot)$. Minimizing $J(\mathbf{d})$ leads to a sparse linear equation system, which we solve with the BiCGSTAB solver of the Eigen linear algebra package. A result of the fill-in operation can be seen in Fig. 8.3d.

After the fill-in operation, we filter the depth map using a shadow filter, where points whose normals are perpendicular to the view ray get discarded. This operation is only executed on the boundaries of the object to keep the object depth map continuous.

To increase invariance of the generated images against camera pitch angle changes, we normalize the viewing angle by an optional reprojection step. The goal is to create a view of the object from a canonical camera pitch angle. To enable reprojection, we first create a mesh using straight-forward triangulation from the filled depth map (Fig. 8.3e). We then render the mesh from the canonical perspective to create a new depth image. Our naïve meshing approach creates a linear interpolation for previously hidden surfaces of the object (see Fig. 8.3f). We believe this to

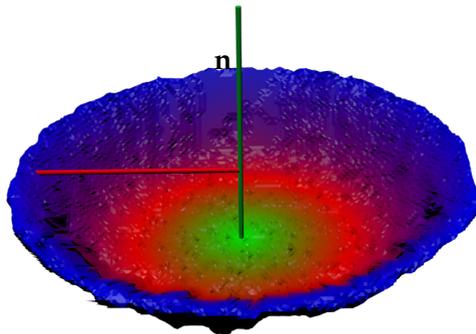


Figure 8.4: Coordinate system used for colorization. The detected plane normal \mathbf{n} through the object center \mathbf{p} is depicted as a green bar, while an exemplary distance to a colored point is shown as a red bar.

be a plausible guess of the unknown geometry without making further assumptions about the object, such as symmetries.

The final preprocessing step is to calculate a color $c_{\mathbf{p}}$ for each depth image pixel $\mathbf{p} = (u, v)$ with the 3D reprojection $\mathbf{p}_3 = (x, y, z)$. We first estimate a 3D object center \mathbf{q} from the bounding box of the object point cloud. The points are then colored according to their distance r from a line g through \mathbf{q} in the direction of the plane normal \mathbf{n} from tabletop segmentation (Fig. 8.4):

$$c_{\mathbf{p}} = \text{pal}(\text{dist}_g(\mathbf{p})). \quad (8.5)$$

As palette function $\text{pal}(\cdot)$, we chose a fixed RGB interpolation from green over red and blue to yellow. Since the coloring is not normalized, this allows the network to discriminate between scaled versions of the same shape. If scale-invariant shape recognition is desired, the coloring can easily be normalized.

Note that depth likely carries less information than color and could be processed at a coarser resolution. We keep resolution constant, however, since the input size of the learned CNN cannot be changed.



Figure 8.5: CNN activations for sample RGB-D frames. The first column shows the CNN input image (color and depth of a pitcher and a banana), all other columns show corresponding selected responses from the first convolutional layer. Note that each column is the result of the same filter applied to color and pre-processed depth.

8.2.3 Image Feature Extraction

We investigated the winning CNN from ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 2011 by Krizhevsky et al. (2012). The open-source Caffe framework (Jia et al., 2014) provides a pre-trained version of this network.

We extract features from the previous-to-last and the last fully connected layer in the network (named $fc7$ and $fc8$ in Caffe). This gives us $4096 + 1000 = 5096$ features per RGB and depth image each, resulting in 10 192 features per RGB-D frame.

Reprojection and coloring are only used for instance-level classification and pose regression, since object categorization cannot benefit from a canonical perspective given the evaluation regime of the WRGBDO dataset.

Figure 8.5 shows responses of the first convolutional layer to RGB and depth stimuli. The same filters show different behavior in RGB and depth channels. As intended by our preprocessing, the activation images exhibit little activity in faded-out background regions.

8.3 LEARNING METHOD

8.3.1 *Object Classification*

For classification, we use linear Support Vector Machines (SVMs). We follow a hierarchical approach as in Lai et al. (2011b): In a first level, a linear multiclass SVM predicts the object category. The next level contains SVMs predicting the instance in each particular category.

8.3.2 *Object Pose Estimation*

The RGB-D object dataset makes the assumption that object orientation is defined by a single angle α around the normal vector of the planar surface. This angle is consistently annotated for instances of each object category. However, annotations are not guaranteed to be consistent across categories.

Instead of regressing α directly, we construct a hierarchy for pose estimation to avoid the discontinuity at $\alpha = 0^\circ = 360^\circ$, which is hard for a regressor to match. We first predict a rough angle interval for α using a linear SVM. In our experiments, four angle intervals of 90° gave best results. For each interval, we then train one RBF-kernel support vector regressor to predict α . During training, we include samples from the neighboring angle intervals to increase robustness against misclassifications on the interval level.

This two-step regressor is trained for each instance. We further train the regressor for each category to provide pose estimation without instance identification, which is supported by the dataset but is not reported by other works, albeit being required in any real-world household robotics application.

8.4 EVALUATION

8.4.1 *Evaluation Protocol*

We evaluate our approach on the WRGBDO dataset (Lai et al., 2011a). It contains 300 objects organized in 51 categories. For each object, there are three turntable sequences captured from different camera elevation angles (30° , 45° , and 60°). The sequences were captured with an ASUS Xtion Pro Live camera with 640×480 resolution in both RGB and depth channels. The dataset also contains approximate ground truth labels for the turntable rotation angle.

Furthermore, the dataset provides an object segmentation based on depth and color. We use this segmentation mask in our pre-processing pipeline. However, since our RGB pre-processing needs background pixels for smooth background fading (Section 8.2.1), we could not use the provided pre-masked evaluation dataset but instead had to use the corresponding frames from the full dataset. Since our method fades out most of the background, only features close to the object remain. This includes the turntable surface, which is not interesting for classification or pose regression and the turntable markers, which do not simplify the regression problem since the objects are placed randomly on the turntable in each view pose. Thus, we believe that our results are still comparable to other results on the same dataset.

For evaluation, we follow the protocol established by Lai et al. (2011a) and Bo et al. (2013). We use every fifth frame for training and evaluation. For category recognition, we report the cross-validation accuracy for ten predefined folds over the objects, i.e. in each fold the test instances are completely unknown to the system.

For instance recognition and pose estimation, we employ the *Leave-Sequence-Out* scheme of Bo et al. (2013), where the system is trained on the 30° and 60° sequences, while evaluation is on the 45° sequence of every instance.

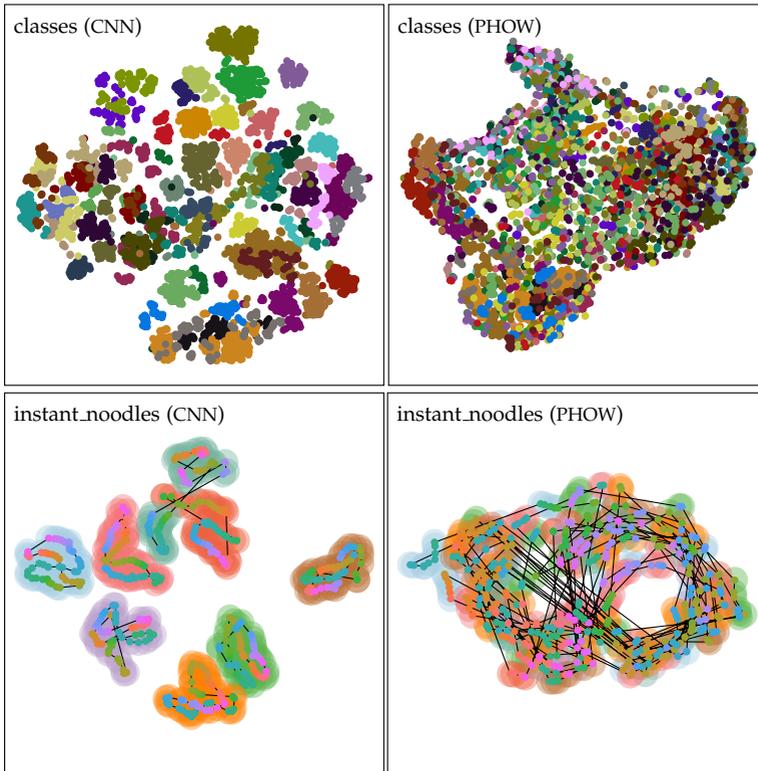


Figure 8.6: Visualization of our CNN-based features. *Top row* shows t-SNE embedding of $1/10$ of the WRGBDO dataset using CNN and PHOW features, colored by class. CNN separates classes better than PHOW. *Bottom row* shows a separate t-SNE embedding of the `instant_noodles` class (45° sequence), colored and connected by pose. The CNN separates instances and creates pose manifolds.

8.4.2 Results

In addition to the work of Bo et al. (ibid.), we compare our proposed method to a baseline of dense SIFT features (PHOW, Bosch et al. 2007), which are extracted at multiple scales, quantized using k -means and histogrammed in a 2×2 and a 4×4 grid over

Table 8.1: Comparison of category and instance level classification accuracies on the WRGBDO dataset.

Method	Category Acc. (%)		Instance Acc. (%)	
	RGB	RGB-D	RGB	RGB-D
Lai et al. (2011a)	74.3 \pm 3.3	81.9 \pm 2.8	59.3	73.9
Bo et al. (2013)	82.4 \pm 3.1	87.5 \pm 2.9	92.1	92.8
PHOW	80.2 \pm 1.8	—	62.8	—
Ours	83.1 \pm 2.0	89.4 \pm 1.3	92.0	94.1

the image. We used vlfeat¹ with standard settings, which are optimized for the Caltech 101 dataset. We then apply SVM training for classification and pose estimation as described in Section 8.3.

Without any supervised learning, we can embed the features produced by the CNN and PHOW in \mathbb{R}^2 using a t-SNE embedding (Van der Maaten and Hinton, 2008). The result is shown in Fig. 8.6. While the upper row shows that CNN object classes are well-separated in the input space, the lower row demonstrates that object instances of a single class also become well-separated. Similar poses of the same object remain close in the feature-space, expressing a low-dimensional manifold. These are highly desirable properties for an unsupervised feature mapping which facilitate learning with very few instances. In contrast, PHOW features only exhibit these properties to a very limited extent: Classes and instances are less well-separated, although pose similarities are largely retained.

Table 8.1 summarizes our recognition results and compares them with other works. We improve on the state of the art in category and instance recognition accuracy for RGB and RGB-D data. The exception is RGB-based instance recognition, where the HMP approach by Bo et al. (2013) wins by 0.1%.

Analyzing the confusion matrix (Fig. 8.7), the category level classification exhibits few systematic errors. Some object cate-

¹ <http://www.vlfeat.org>

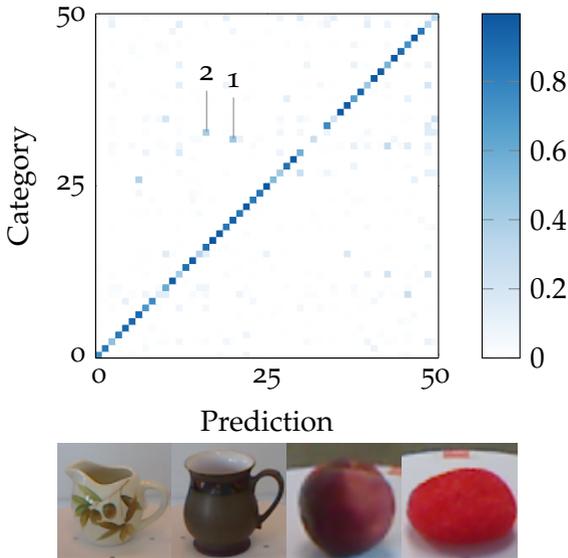


Figure 8.7: *Top*: Confusion matrix for category recognition, normalized by number of samples for each ground truth label. Selected outliers: 1) pitcher recognized as coffee_mug, 2) peach as sponge. *Bottom*: Sample images for pitcher, coffee_mug, peach, and sponge.

gories prove to be very difficult, since they contain instances with widely varying shape but only few examples (e.g. mushroom), or instances which are very similar in color and shape to instances of other classes (e.g. pitcher and coffee_mug). Telling apart the peaches from similarly rounded but brightly colored sponges would likely profit from more examples and detailed texture analysis.

Classification performance degrades gracefully when the dataset size is reduced, which is shown in Fig. 8.8. We reduce the dataset for category and instance recognition by uniform stratified sampling on category and instance level, respectively. With only 30% of the training set available, category classification accuracy decreases by 0.65 percentage points only (PHOW: 2.2%), while instance classification decreases by roughly 2% (PHOW: 25.2% from 62.6%, not shown). This supports our observation that the CNN

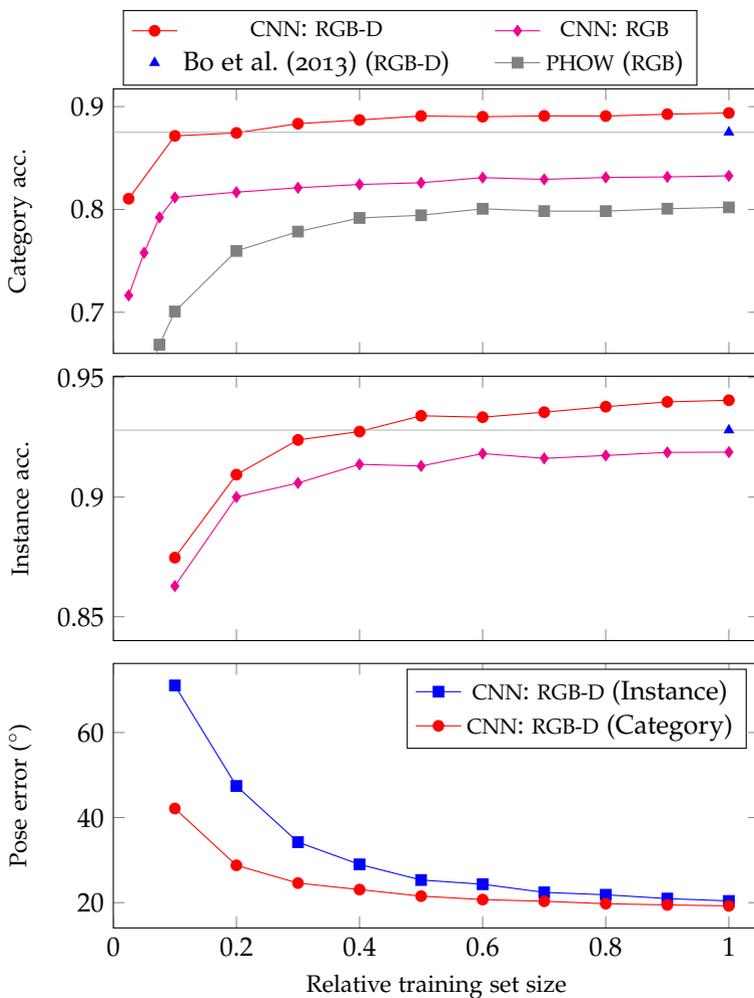


Figure 8.8: Learning curves for classification accuracy (top and center) and median pose estimation error (bottom). We report cross validation accuracy for category recognition and accuracy on the 45° sequence for instance recognition.

Table 8.2: Median and average pose estimation error on the WRGBDO dataset. Wrong classifications are penalized with 180° error. *Cat* and *Inst* describe subsets with correct category/instance classification, respectively.

Method	Median Pose Err. ($^\circ$)			Average Pose Err. ($^\circ$)		
	All	Cat	Inst	All	Cat	Inst
Lai et al. (2011b)	62.6	51.5	30.2	83.7	77.7	57.1
Bo et al. (2013)	20.0	18.7	18.0	53.6	47.5	44.8
Ours—instance level	20.4	20.4	18.7	51.0	50.4	42.8
Ours—category level	19.2	19.1	18.9	45.0	44.5	43.7

feature space already separates the categories of the RGB-D objects in a semantically meaningful way.

When using the evaluation regime for instance classification and pose estimation, i.e. when all instances are known, our category classification achieves near perfect accuracy (99.6%).

We also improve the state of the art in pose estimation by a small margin. Table 8.2) reports the pose estimation error of the instance-level estimation and the category-level estimation. The Median Pose Error (All) is computed with a 180° penalty if the class or instance of the object was not recognized. The Median Pose Error (Cat) is only computed for objects where the correct category was predicted, with a 180° penalty for wrongly predicted instances. Finally, Median Pose Error (Inst) only counts the samples where class and instance were identified correctly. Average Pose (All/Cat/Inst) describe the average pose error in each case, respectively. Notably, our average pose error is significantly lower than the pose error of the other methods. We were not able to produce reasonable accuracies for pose based on the PHOW features, since the large instance classification error strongly affects all pose estimation metrics.

Surprisingly, our category-level pose regression achieves even lower median pose error, surpassing the state-of-the-art result of Bo et al. (2013). The category-level estimation is less precise only

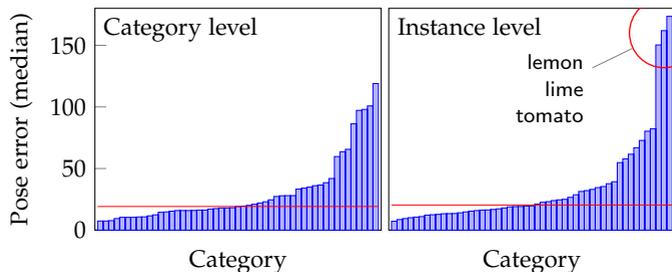


Figure 8.9: Distribution of median pose error over categories. Left plot shows median pose error over categories, right plot over instances. Median over all categories is shown in red. Some objects of type lemon, lime, and tomato exhibit high rotation symmetry and do not support pose estimation.

Table 8.3: Runtimes of various algorithm steps per input frame in seconds. We measured runtime on an Intel Core i7-4800MQ @ 2.7 GHz and a standard mobile graphics card (Nvidia GeForce GT 730M) for CUDA computations. Timings include all preprocessing steps.

Step	Our work	Bo et al. (2013)
Feature extraction (RGB)	0.013	0.294
Feature extraction (depth)	0.173	0.859
Total	0.186	1.153

in the Median Pose Error (Inst) and Average Pose Error (Inst) categories, where its broader knowledge is not as useful as precise fitting to the specific instance. Figure 8.9 shows the distribution of pose errors over categories. We note that the dataset contains objects in least three categories which exhibit rotation symmetries and do not support estimating pose. This effect is mitigated by category level pose estimation. This shows that pose estimation can greatly benefit from the generalization across instances provided by category-level training.

The color palette choice for our depth colorization is a crucial parameter. We compare the four-color palette introduced in Sec-

Table 8.4: Color palettes for depth colorization with corresponding instance recognition accuracy.

Palette		Accuracy (%)	
		Depth only	RGB-D
Gray		41.8	93.1
Green		38.8	93.3
Green-red-blue-yellow		45.5	94.1

tion 8.2 to two simpler colorization schemes (black and green with brightness gradients) shown in Table 8.4 and compared them by instance recognition accuracy. Especially when considering purely depth-based prediction, the four-color palette wins by a large margin. We conclude that more colors result in more discriminative depth features.

Since computing power is usually very constrained in robotic applications, we benchmarked runtime for feature extraction and prediction on a lightweight mobile computer with an Intel Core i7-4800MQ CPU @ 2.7 GHz and a common mobile graphics card (Nvidia GeForce GT 730M) for CUDA computations. As can be seen in Table 8.3, the runtime of our approach is dominated by the depth pre-processing pipeline, which is at the time of writing not yet optimized for speed. Still, our runtimes are low enough to allow frame rates of up to 5 Hz in a future real-time application.

8.5 CONCLUSION

We presented an approach which allows object categorization, instance recognition and pose estimation of objects on planar surfaces. Instead of learning or handcrafting features, we relied on a convolutional neural network (CNN) which was trained on a large image categorization dataset. We made use of depth features by rendering objects from canonical views and proposed a CNN-compatible coloring scheme which codes metric distance from the object center. We evaluated our approach on the chal-

lenging WRGBDO dataset and find that in feature space, categories and instances are well separated. Supervised learning on the CNN features improves state of the art in classification as well as average pose accuracy. Our performance degrades gracefully when the dataset size is reduced.

After the publication of the contents of this chapter, Eitel et al. (2015) used a simple depth colorization approach on the same dataset. The authors fine-tune the features, which improves the category classification when compared to our results. Bogun et al. (2015) further increased categorization accuracy by including temporal context into the decision using convolutional LSTM networks. It remains unclear, however, whether our good performance on reduced dataset sizes can be met by fine-tuned neural networks due to the danger of overfitting small training sets. Due to the labeling work required, we consider robustness to small datasets essential for any robotics application.

CONCLUSION

In this thesis, we developed deep neural network methods that help computers to understand natural images. We focused on two tasks, object recognition and object class segmentation. For our studies, we first developed a software framework, which allows fast execution on GPU while allowing high-level model specification and debugging. This open-source framework has been used in numerous theses, publications, and as a teaching device. We also built and evaluated a fast GPU library for random forests, which greatly reduces training time and allows real-time image segmentation even on mobile GPUs. Our optimized random forest segmentations provide the baseline for later object class segmentations using deep neural networks.

We provided an introduction to deep learning, which is a set of tools to build hierarchical models from a limited set of operations. One possibility is to learn the hierarchical models step by step greedily, adding one block of operations at a time, and only later fine-tune on e.g. object recognition. In multiple studies, we analyzed two types of blocks, restricted Boltzmann machines (RBMs) and auto-encoders. Our study has shown that RBMs can be further restricted to use local connectivity, which reflects statistical properties of natural images and resulted in better models of the data distribution compared to fully connected models of the same size. Lateral connections even allowed the (now semi-restricted) RBMs to model long-range interactions between hidden variables. In a second study, we empirically analyzed methods for the evaluation of RBM learning. We ran hundreds of experiments on medium-size RBMs on multiple datasets, and repeatedly compared two commonly used approximate quality measures—reconstruction error and AIS—to ground truth. We found that while the former is not reliable at all, the latter systematically fails to detect degradation, especially in rather desirable situations of fast learning progress. These crucial results for RBM model com-

parison were novel, as these effects do not occur in small models. We then turned to auto-encoders, another building block of deep learning models. We showed that in common models, encoders are too simplistic and cannot detect certain connections between input variables. We demonstrated this problem on a constructed example and showed that our proposed solution—a contractive two-layer encoder with shortcuts—improves over the more common one-layer encoder.

The third main topic of the thesis is object class segmentation of images. Object class segmentation was not a task typically attempted with DNNs. Our proposed DNN model makes use of HOG inputs, supervised pre-training, and prediction refinement. Evaluating on the IG02 and MSRC-9 datasets, we could demonstrate that DNN can excel at this task as well. We then add methods to deal with depth inputs from RGB-D cameras and with the time domain for indoor object class segmentation on the NYUD dataset. For depth, we found that using depth-normalization, covering windows, and height-above ground results in models which outperform previously superior random forest-based methods. We created a recurrent neural network (RNN) for object class segmentation inspired by the neural abstraction pyramid. Compared to the previously introduced non-recurrent models, larger spatial contexts can be processed and sequential processing of video frames becomes possible. Both properties contribute to a performance which surpasses all other methods training on NYUD.

Mainly by substituting the loss function, we used pixel-wise predictions for object detection by combining DNNs with established structured learning methods. We demonstrated that it is feasible to compute DNN gradients through a structured loss function and evaluated on a subset of the VOC 2007 dataset classes.

In a final study, we proposed to use a transfer learning approach to deal with small-scale datasets in robotics. We found that our object-centered depth colorization scheme allows us to use RGB-D inputs for DNNs that have been trained only on RGB images. The resulting model outperformed competing approaches on the Washington RGB-D objects dataset tasks of object recognition, instance recognition and pose estimation, even when dataset size was greatly reduced.

9.1 FUTURE DIRECTIONS

Many questions remain open for future research. To extend the methods presented in this thesis, we think the following directions would be interesting to follow.

The computer vision community has largely moved away from unsupervised learning, considering that many supervised learning datasets are available which produce good features. Nevertheless, even supervised learning methods performing well on large datasets can assign high probability to low-probability regions (Szegedy et al., 2014; Goodfellow et al., 2014). Developing methods for detection and tracking of these modes becomes more important as DNN usage in our devices grows.

We have shown that certain variable relationships are hard to detect with single-layered auto-encoders. Our experimental results show that—in contrast to our constructed example—images are less prone to be affected. We hypothesize that datasets with more complicated variable relationships will show a stronger effect.

Our object class segmentation methods are focused on purely supervised learning, but transfer learning approaches (e.g. Gupta et al., 2014; Eigen and Fergus, 2015) currently provide best performance. Combining our proposed HOG/HOD inputs—recently again shown to be advantageous by Ren et al. (2015)—with supervised learning methods could further increase performance. Considering that our best model is a recurrent one, moving to unsupervised pre-training (e.g. via Liang and Hu, 2015) or even transfer learning of RNNs might also be a productive research direction.

Object detection is making tremendous progress, mostly based on region proposals using image over-segmentations (e.g. Girshick et al., 2014). Structured methods such as introduced in this thesis are used for fine-tuning (Gidaris and Komodakis, 2015). Ideally, we would like to invert the order, and let neural networks propose regions. For this, object class segmentation methods might be a good starting point (German et al., 2016; Gidaris and Komodakis, 2015), especially if both approaches can be combined into one loss function. This combination would also repre-

sent a step in the direction of general scene understanding, since object class segmentation is more helpful for surface categorization (e.g. floor, walls), while object detection can identify e.g. objects which can be manipulated. With similar reasoning, our proposed normalization schemes— for depth, which we applied to segmentation, and for viewing angle, which we applied to centered objects— are likely to improve object detection results as well.

ACRONYMS AND SYMBOLS

DATASETS

Caltech 101	Caltech 101 object recognition dataset, Fei-Fei et al. (2007), a database containing 101 categories, with 40 to 800 images per category
CIFAR-10	CIFAR-10 object recognition dataset, Krizhevsky (2009), a database containing 60 000 color images of size 32×32 with ten classes
IG02	INRIA Graz-02 object class segmentation database, Marszatek and Schmid (2007), containing 958 RGB images with pixels annotated by one of 4 classes
ILSVRC	ImageNet large scale visual recognition challenge, Russakovsky et al. (2014)
MNIST	MNIST database of handwritten digits, LeCun et al. (1998a), containing 70 000 grayscale images of size 28×28 labeled with ten classes
MNIST-rot	database of rotated handwritten digits derived from MNIST, Larochelle et al. (2007), containing 12 000 training and 50 000 test grayscale images of size 28×28 labeled with ten classes
MSRC-21	MSRC-21 object class segmentation database, Shotton et al. (2006), containing 591 RGB images with pixels annotated by one of 21 classes

MSRC-9	MSRC-9 object class segmentation database, Shotton et al. (2006), containing 240 RGB images with pixels annotated by one of 9 classes
NORB	NORB 3D object recognition database, LeCun et al. (2004), containing stereo images of 50 toys belonging to 5 categories with 5 instances each
NYUD	New York University depth dataset, version 2, Silberman and Fergus (2011), a database containing 1440 RGB-D VGA images of indoor scenes
PASCAL-10X	a database containing roughly 10 times as many images as VOC for 10 of its classes, with bounding box annotations for exactly one object per image, Zhu et al. (2012)
VOC	PASCAL visual object classes recognition challenge, Everingham et al. (2010)
WRGBDO	Washingont RGB-D objects dataset, Lai et al. (2011a), a database containing RGB-D images of 300 household objects from 51 categories, with category and pose annotations

ALGORITHMS

AIS	annealed importance sampling, Neal (2001) and Salakhutdinov (2009a)
AE	auto-encoder
BPTT	backpropagation through time
CD	contrastive divergence, Hinton et al. (2006)
CD1	contrastive divergence a single Markov chain step
CNN	convolutional neural network, LeCun et al. (1998a)

CRF	conditional random field
DBM	deep Boltzmann machine, Salakhutdinov and Hinton (2009)
DBN	deep belief network, Hinton et al. (2006)
DNN	deep neural network
FPCD	fast persistent CD, Tieleman and Hinton (2009)
HMP	hierarchical matching pursuit, Bo et al. (2011)
HOD	histogram of oriented depths, Spinello and Arras (2011)
HOG	histogram of oriented gradients, Dalal and Triggs (2005)
ISA	independent subspace analysis
LDPC	low density parity check, Gallager (1962)
LIRBM	local impact RBM, Schulz et al. (2010a)
LSTM	long short-term memory, Hochreiter and Schmidhuber (1997)
MCMC	Markov chain Monte Carlo
MLP	multi-layer perceptron
NAP	neural abstraction pyramid, Behnke (2003b)
PCD	persistent CD, Tieleman (2008)
PHOW	descriptor generated by computing dense SIFT at multiple resolutions, Bosch et al. (2007)
RBM	restricted Boltzmann machine, Smolensky (1986)
RCNN	recurrent convolutional neural network
ReLU	rectifying linear unit
RF	random forest, Breiman (2001)
RMSProp	resilient root mean square backpropagation, analyzed in Dauphin et al. (2015)
RNN	recurrent neural network

RPROP	resilient backpropagation, Riedmiller and Braun (1993)
SGD	stochastic gradient descent
SIFT	scale invariant feature transform, Lowe (2004)
SLAM	self-localization and mapping,
SRBM	semi-restricted Boltzmann machine, Osindero and Hinton (2008)
SVM	support vector machine, Cortes and Vapnik (1995)
ZCA	zero phase whitening, Hyvärinen and Oja (2000)

OTHER ACRONYMS

API	application programming interface
CPU	central processing unit
CUDA	compute unified device architecture
CURFIL	CUDA random forests for image labeling (library name), Schulz et al. (2016)
CUV	GPU library created for thesis
CUVNET	gradient descent optimization library created for this thesis
GPU	graphics processing unit
GUI	graphical user interface
IDE	integrated development environment
NaN	not a number
PR-EER	precision/recall at equal error rate
RGB	red, green, and blue (image)
RGB-D	red, green, blue, and depth (image)
SIMD	single instruction, multiple data
XML	extended markup language

XOR exclusive OR

NOTATION

Throughout the thesis, unless noted otherwise, we adhere to the following conventions:

- Bold lowercase symbols (\mathbf{x} , \mathbf{h} , etc.) denote vectors.
- V , W , X , etc. denote matrices or multi-dimensional arrays.
- To avoid excessive subscripts, we use capital A , B , C , ... for the maximum value in loop counters. For example, in $\sum_{d=1}^D x_d$, the symbol D denotes the maximum value of d . In these contexts, we use bold capital letters for matrices.
- Lowercase symbols (x , h , etc.) denote scalars. Thus, the first value in the vector $\mathbf{x} \in \mathbb{R}^N$ is x_0 , and the top left value in matrix $W \in \mathbb{R}^{N \times M}$ is $w_{0,0}$.
- Estimated values carry a caret, e.g. $\hat{\mathbf{y}}$ is an estimator for \mathbf{y}
- The notation $\langle x \rangle_{x \in X}$ refers to the average of all elements in X .
- We make use of the Iverson bracket notation, i.e.,

$$[P] = \begin{cases} 1 & \text{if } P \text{ is true,} \\ 0 & \text{else.} \end{cases}$$

- Sans-serif fonts are used for experimental conditions and category names.

BIBLIOGRAPHY

- Aldavert, D., R. De Mantaras, A. Ramisa, and R. Toledo (2010). "Fast and robust object segmentation with the Integral Linear Classifier." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*, pp. 1046–1053.
- Amit, Y. and D. Geman (1997). "Shape quantization and recognition with randomized trees." In: *Neural computation* 9.7, pp. 1545–1588.
- Baldi, P. and K. Hornik (1989). "Neural networks and principal component analysis: learning from examples without local minima." In: *Neural Networks* 2.1, pp. 53–58.
- Behnke, S. (1999). "Hebbian learning and competition in the Neural Abstraction Pyramid." In: *International Joint Conference on Neural Networks (IJCNN)*. Vol. 2. Washington, DC, USA, pp. 1356–1361.
- Behnke, S. (2003a). "Discovering hierarchical speech features using convolutional non-negative matrix factorization." In: *International Joint Conference on Neural Networks (IJCNN)*. Vol. 4. Portland, Oregon, USA, pp. 2758–2763.
- Behnke, S. (2003b). *Hierarchical neural networks for image interpretation*. Vol. 2766. Lecture Notes in Computer Science (LNCS). Springer.
- Bengio, Y. and O. Delalleau (2011). "On the expressive power of deep architectures." In: *Algorithmic Learning Theory*. Springer, pp. 18–36.
- Bengio, Y. (2009). "Learning deep architectures for AI." In: *Foundations and Trends in Machine Learning* 2.1, pp. 1–127.
- Bengio, Y., A. Courville, and P. Vincent (2013). "Representation Learning: A Review and New Perspectives." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 8. Special Issue on Learning Deep Architectures, pp. 1798–1828.
- Bengio, Y., P. Lamblin, D. Popovici, and H. Larochelle (2006). "Greedy layer-wise training of deep networks." In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 153–160.
- Bergstra, J., B. Frédríc, J. Turian, R. Pascanu, O. Delalleau, O. Breuleux, P. Lamblin, G. Desjardins, D. Erhan, and Y. Bengio (2010). "Deep Learning on GPUs with Theano." In: *The Learning Workshop*. Abstract only.
- Bergstra, J., R. Bardenet, Y. Bengio, B. Kégl, et al. (2011). "Algorithms for hyper-parameter optimization." In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 2546–2554.

- Bo, L., X. Ren, and D. Fox (2011). "Hierarchical matching pursuit for image classification: Architecture and fast algorithms." In: *Advances in neural information processing systems*, pp. 2115–2123.
- Bo, L., X. Ren, and D. Fox (2013). "Unsupervised feature learning for RGB-D based object recognition." In: *International Symposium Experimental Robotics (ISER)*, pp. 387–402.
- Bogun, I., A. Angelova, and N. Jaitly (2015). "Object Recognition from Short Videos for Robotic Perception." In: arXiv: 1509.01602 [cs.CV].
- Bosch, A., A. Zisserman, and X. Munoz (2007). "Image classification using random forests and ferns." In: *International Conference on Computer Vision (ICCV)*.
- Bottou, L. (2014). "From machine learning to machine reasoning." In: *Machine Learning* 94 (2), pp. 133–149. arXiv: 1102.1808 [cs.AI].
- Boureau, Y., F. Bach, Y. LeCun, and J. Ponce (2010). "Learning mid-level features for recognition." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*. San Francisco, CA, USA, pp. 2559–2566.
- Bourlard, H. and Y. Kamp (1988). "Auto-association by multilayer perceptrons and singular value decomposition." In: *Biological Cybernetics* 59, pp. 291–294.
- Breiman, L. (2001). "Random forests." In: *Machine learning* 45.1, pp. 5–32.
- Brox, T. and J. Malik (2011). "Large displacement optical flow: descriptor matching in variational motion estimation." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 33.3, pp. 500–513.
- Buchaca, D., E. Romero, F. Mazzanti, and J. Delgado (2013). "Stopping criteria in contrastive divergence: Alternatives to the reconstruction error." In: arXiv: 1312.6062 [cs.NE].
- Cho, K., T. Raiko, and A. Ilin (2010). "Parallel Tempering is Efficient for Learning Restricted Boltzmann Machines." In: *International Joint Conference on Neural Networks (IJCNN)*.
- Cho, K., T. Raiko, and A. Ilin (2011a). "Enhanced gradient and adaptive learning rate for training restricted Boltzmann machines." In: *International Conference on Machine Learning (ICML)*, pp. 105–112.
- Cho, K., A. Ilin, and T. Raiko (2011b). "Improved learning of Gaussian-Bernoulli restricted Boltzmann machines." In: *International Conference on Artificial Neural Networks (ICANN)*. Springer.
- Cho, K., T. Raiko, and A. Ilin (2013). "Enhanced gradient for training restricted Boltzmann machines." In: *Neural Computation* 25.3, pp. 805–831.
- Cho, K., T. Raiko, A. Ilin, and J. Karhunen (2012). "A Two-Stage Pretraining Algorithm for Deep Boltzmann Machines." In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*. Lake Tahoe.

- Cireřan, D., U. Meier, J. Masci, L. Gambardella, and J. Schmidhuber (2011). "High-Performance Neural Networks for Visual Object Classification." In: arXiv: 1102.0183 [cs.AI].
- Cireřan, D., U. Meier, and J. Schmidhuber (2012a). "Multi-column deep neural networks for image classification." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.
- Cireřan, D. C., U. Meier, and J. Schmidhuber (2012b). "Multi-column Deep Neural Networks for Image Classification." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*, pp. 3642–3649.
- Cireřan, D., A. Giusti, L. Gambardella, and J. Schmidhuber (2012c). "Deep Neural Networks Segment Neuronal Membranes in Electron Microscopy Images." In: *Advances in Neural Information Processing Systems (NIPS)*.
- Cireřan, D., A. Giusti, L. Gambardella, and J. Schmidhuber (2013). "Mitosis Detection in Breast Cancer Histology Images with Deep Neural Networks." In: *Medical Image Computing and Computer Assisted Intervention (MICCAI), International Conference on*.
- Coates, A., H. Lee, and A. Y. Ng (2010). "An Analysis of Single-Layer Networks in Unsupervised Feature Learning." In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. Chia Laguna, Italy.
- Cortes, C. and V. Vapnik (1995). "Support-Vector Networks." In: *Machine Learning* 20.3, pp. 273–297.
- Coupric, C., C. Farabet, L. Najman, and Y. LeCun (2013). "Indoor Semantic Segmentation using depth information." In: arXiv: 1301.3572 [cs.CV].
- Cybenko, G. (1989). "Approximation by superpositions of a sigmoidal function." In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4, pp. 303–314.
- Dalal, N. and B. Triggs (2005). "Histograms of oriented gradients for human detection." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.
- Dauphin, Y. N., H. de Vries, J. Chung, and Y. Bengio (2015). "RMSProp and equilibrated adaptive learning rates for non-convex optimization." In: arXiv: 1502.04390 [cs.LG].
- Dauphin, Y., R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio (2014). "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization." In: arXiv:1406.2572 [cs.LG].
- Dean, J., G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. (2012). "Large scale distributed

- deep networks." In: *Advances in Neural Information Processing Systems (NIPS)*.
- Desjardins, G., A. Courville, Y. Bengio, P. Vincent, and O. Dellaleau (2010). "Parallel tempering for training of restricted Boltzmann machines." In: *Journal of Machine Learning Research Workshop and Conference Proceedings*. Vol. 9, pp. 145–152.
- Donahue, J., L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell (2015). "Long-term recurrent convolutional networks for visual recognition and description." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.
- Donahue, J., Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell (2014). "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition." In: *International Conference on Machine Learning (ICML)*, pp. 647–655.
- Duchi, J., E. Hazan, and Y. Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of Machine Learning Research (JMLR)* 12, pp. 2121–2159.
- Eigen, D. and R. Fergus (2015). "Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture." In: *International Conference on Computer Vision (ICCV)*.
- Eitel, A., J. T. Springenberg, L. Spinello, M. Riedmiller, and W. Burgard (2015). "Multimodal Deep Learning for Robust RGB-D Object Recognition." In: *Intelligent Robots and Systems (IROS), International Conference on*.
- Erhan, D., P. Manzagol, Y. Bengio, S. Bengio, and P. Vincent (2009). "The difficulty of training deep architectures and the effect of unsupervised pre-training." In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 153–160.
- Erhan, D., Y. Bengio, A. C. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio (2010a). "Why does unsupervised pre-training help deep learning?" In: *Journal of Machine Learning Research (JMLR)* 11, pp. 625–660.
- Erhan, D., Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio (2010b). "Why Does Unsupervised Pre-training Help Deep Learning?" In: *Journal of Machine Learning Research (JMLR)* 11, pp. 625–660.
- Erhan, D., C. Szegedy, A. Toshev, and D. Anguelov (2014). "Scalable Object Detection using Deep Neural Networks." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.

- Everingham, M., L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman (2010). "The pascal visual object classes (VOC) challenge." In: *International Journal of Computer Vision (IJCV)* 88.2.
- Farabet, C., C. Couprie, L. Najman, and Y. LeCun (2012). "Scene parsing with multiscale feature learning, purity trees, and optimal covers." In: *International Conference on Machine Learning (ICML)*.
- Farneback, G. (2003). "Two-frame motion estimation based on polynomial expansion." In: *Image Analysis*. Springer, pp. 363–370.
- Fei-Fei, L., R. Fergus, and P. Perona (2007). "Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories." In: *Computer Vision and Image Understanding* 106.1, pp. 59–70.
- Fidler, S. and A. Leonardis (2007). "Towards scalable representations of object categories: Learning a hierarchy of parts." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*. Minneapolis, MN, USA.
- Fischer, A. and C. Igel (2010). "Empirical Analysis of the Divergence of Gibbs Sampling Based Learning Algorithms for Restricted Boltzmann Machines." In: *International Conference on Artificial Neural Networks (ICANN)*, pp. 208–217.
- Flynn, M. J. (1972). "Some computer organizations and their effectiveness." In: *Computers, IEEE Transactions on* 100.9, pp. 948–960.
- Fukushima, K. (1980). "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position." In: *Biological Cybernetics* 36.4, pp. 193–202.
- Fulkerson, B., A. Vedaldi, and S. Soatto (2009). "Class segmentation and object localization with superpixel neighborhoods." In: *International Conference on Computer Vision (ICCV)*.
- Gallager, R. (1962). "Low-density parity-check codes." In: *IRE Transactions on Information Theory* 8.1, pp. 21–28.
- German, M., F. Husain, H. Schulz, S. Frintrop, and S. Behnke (2016). In: *International Conference on Robotics and Automation (ICRA)*. Submitted.
- Gidaris, S. and N. Komodakis (2015). "Object detection via a multi-region & semantic segmentation-aware CNN model." In: arXiv: 1505.01749 [cs.CV].
- Girshick, R., J. Donahue, T. Darrell, and J. Malik (2014). "Rich feature hierarchies for accurate object detection and semantic segmentation." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.

- Gkioxari, G., B. Hariharan, R. Girshick, and J. Malik (2014). "R-CNNs for Pose Estimation and Action Detection." In: arXiv: 1406.5212 [cs.CV].
- Glorot, X. and Y. Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks." In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 249–256.
- Goodfellow, I. J., J. Shlens, and C. Szegedy (2014). "Explaining and harnessing adversarial examples." In: arXiv: 1412.6572 [cs.LG].
- Goodfellow, I. J., D. Warde-Farley, P. Lamblin, V. Dumoulin, M. Mirza, R. Pascanu, J. Bergstra, F. Bastien, and Y. Bengio (2013a). "Pylearn2: a machine learning research library." In: arXiv: 1308.4214 [stat.ML].
- Goodfellow, I. J., D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio (2013b). "Maxout networks." In: *International Conference on Machine Learning (ICML)*.
- Gould, S., J. Rodgers, D. Cohen, G. Elidan, and D. Koller (2008). "Multi-class segmentation with relative location prior." In: *Computer Vision* 80.3, pp. 300–316.
- Grangier, D., L. Bottou, and R. Collobert (2009). "Deep convolutional networks for scene parsing." In: *ICML 2009 Deep Learning Workshop*.
- Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. Vol. 385. Studies in Computational Intelligence. Springer.
- Graves, A., M. Abdelrahman, and G. E. Hinton (2013). "Speech Recognition with Deep Recurrent Neural Networks." In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.
- Grosse, R. B., C. J. Maddison, and R. R. Salakhutdinov (2013). "Annealing between distributions by averaging moments." In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 2769–2777.
- Gupta, S., R. Girshick, P. Arbeláez, and J. Malik (2014). "Learning rich features from RGB-D images for object detection and segmentation." In: *European Conference on Computer Vision (ECCV)*, pp. 345–360.
- He, K., X. Zhang, S. Ren, and J. Sun (2015). "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In: arXiv: 1502.01852 [cs.CV].
- Hermans, A., G. Floros, and B. Leibe (2014). "Dense 3D Semantic Mapping of Indoor Scenes from RGB-D Images." In: *International Conference on Robotics and Automation (ICRA)*. Hong Kong.
- Hinton, G., S. Osindero, and Y. Teh (2006). "A fast learning algorithm for deep belief nets." In: *Neural computation* 18.7, pp. 1527–1554.
- Hinton, G. (2002). "Training products of experts by minimizing contrastive divergence." In: *Neural Computation* 14 (8), pp. 1771–1800.

- Hinton, G. (2012). "A Practical Guide to Training Restricted Boltzmann Machines." In: *Neural Networks: Tricks of the Trade*. Ed. by G. Montavon, G. B. Orr, and K.-R. Müller. Vol. 7700. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 599–619.
- Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov (2012). "Improving neural networks by preventing co-adaptation of feature detectors." In: arXiv: 1207.0580 [cs.NE].
- Hinton, G. and R. Salakhutdinov (2006). "Reducing the dimensionality of data with neural networks." In: *Science* 313.5786, pp. 504–507.
- Ho, T. (1995). "Random decision forests." In: *International Conference on Document Analysis and Recognition (ICDAR)*.
- Hochreiter, S., Y. Bengio, P. Frasconi, and J. Schmidhuber (2001). "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies." In: *A Field Guide to Dynamical Recurrent Neural Networks*. Ed. by S. C. Kremer and J. F. Kolen. Wiley-IEEE Press.
- Hochreiter, S. and J. Schmidhuber (1997). "Long short-term memory." In: *Neural computation* 9.8, pp. 1735–1780.
- Höft, N., H. Schulz, and S. Behnke (2014). "Fast Semantic Segmentation of RGB-D Scenes with GPU-Accelerated Deep Neural Networks." In: *German Conference on Artificial Intelligence (KI)*. Lecture Notes in Computer Science (LNCS) 8736. Springer.
- Holz, D., S. Holzer, R. B. Rusu, and S. Behnke (2012). "Real-time plane segmentation using RGB-D cameras." In: *RoboCup 2011: Robot Soccer World Cup XV*, pp. 306–317.
- Hornik, K., M. Stinchcombe, and H. White (1989). "Multilayer feedforward networks are universal approximators." In: *Neural Networks* 2.5, pp. 359–366.
- Huang, J. and D. Mumford (1999). "Statistics of natural images and models." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*. Ft. Collins, CO, USA.
- Hyvärinen, A. and E. Oja (2000). "Independent component analysis: algorithms and applications." In: *Neural Networks* 13.4, pp. 411–430.
- Jain, V. and H. Seung (2008). "Natural image denoising with convolutional networks." In: *Advances in Neural Information Processing Systems (NIPS)*.
- Jansson, K., H. Sundell, and H. Bostrom (2014). "gpuRF and gpuERT: Efficient and Scalable GPU Algorithms for Decision Tree Ensembles." In: *International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*.

- Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell (2014). "Caffe: Convolutional Architecture for Fast Feature Embedding." In: arXiv: 1408.5093 [cs.CV].
- Jung, M., J. Hwang, and J. Tani (2014). "Multiple Spatio-Temporal Scales Neural Network for Contextual Visual Recognition of Human Actions." In: *International Conference on Development and Learning and on Epigenetic Robotics (ICDL)*.
- Kahn, A. B. (1962). "Topological sorting of large networks." In: *Communications of the ACM* 5.11, pp. 558–562.
- Karlsson, B. (2002). "Smart pointers in Boost." In: *C/C++ Users Journal* 20.4, pp. 34–40.
- Karpathy, A., G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei (2014). "Large-scale video classification with convolutional neural networks." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*, pp. 1725–1732.
- Kavukcuoglu, K., M. Ranzato, and Y. LeCun (2010). "Fast Inference in Sparse Coding Algorithms with Applications to Object Recognition." In: arXiv: 1010.3467 [cs.CV].
- Konda, K. R., R. Memisevic, and V. Michalski (2013). "Learning to encode motion using spatio-temporal synchrony." In: arXiv: 1306.3162 [cs.CV].
- Krizhevsky, A. (2009). "Learning multiple layers of features from tiny images." University of Toronto.
- Krizhevsky, A. (2015). *cuda-convnet – High-performance C++/CUDA implementation of convolutional neural networks*. URL: <https://code.google.com/p/cuda-convnet/>.
- Krizhevsky, A., I. Sutskever, and G. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems (NIPS)*. Ed. by P. Bartlett, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, pp. 1106–1114.
- Ladicky, L., C. Russell, P. Kohli, and P. Torr (2009). "Associative hierarchical crfs for object class image segmentation." In: *International Conference on Computer Vision (ICCV)*, pp. 739–746.
- Lai, K., L. Bo, X. Ren, and D. Fox (2011a). "A large-scale hierarchical multi-view RGB-D object dataset." In: *International Conference on Robotics and Automation (ICRA)*, pp. 1817–1824.
- Lai, K., L. Bo, X. Ren, and D. Fox (2011b). "A Scalable Tree-Based Approach for Joint Object and Pose Recognition." In: *Conference on Artificial Intelligence (AAAI)*.

- Lampert, C. H. (2011). "Maximum Margin Multi-Label Structured Prediction." In: *Advances in Neural Information Processing Systems (NIPS)*. Vol. 11.
- Lampert, C. H., M. B. Blaschko, and T. Hofmann (2009). "Efficient sub-window search: A branch and bound framework for object localization." In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 31.12.
- Larochelle, H., D. Erhan, A. Courville, J. Bergstra, and Y. Bengio (2007). "An empirical evaluation of deep architectures on problems with many factors of variation." In: *International Conference on Machine Learning (ICML)*. ACM, pp. 473–480.
- Le Roux, N. and Y. Bengio (2008). "Representational power of restricted boltzmann machines and deep belief networks." In: *Neural Computation* 20.6, pp. 1631–1649.
- Le, Q. V., W. Y. Zou, S. Y. Yeung, and A. Y. Ng (2011). "Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*, pp. 3361–3368.
- LeCun, Y., B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel (1989). "Backpropagation applied to handwritten zip code recognition." In: *Neural computation* 1.4, pp. 541–551.
- LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998a). "Gradient-based learning applied to document recognition." In: vol. 86. 11, pp. 2278–2324.
- LeCun, Y., L. Bottou, G. Orr, and K. R. Müller (1998b). "Efficient Back-Prop." In: *Neural Networks: Tricks of the Trade*. Ed. by G. Orr and K. Müller. Vol. 1524. Lecture Notes in Computer Science. Springer Verlag, pp. 5–50.
- LeCun, Y., F. J. Huang, and L. Bottou (2004). "Learning methods for generic object recognition with invariance to pose and lighting." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*. Vol. 2.
- Lee, H., R. Grosse, R. Ranganath, and A. Ng (2009). "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations." In: *International Conference on Machine Learning (ICML)*. Montreal, Quebec, Canada: ACM, pp. 609–616.
- Lepetit, V. and P. Fua (2006). "Keypoint recognition using randomized trees." In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 28.9, pp. 1465–1479.
- Lepetit, V., P. Lagger, and P. Fua (2005). "Randomized trees for real-time keypoint recognition." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*. Vol. 2.

- Levin, A., D. Lischinski, and Y. Weiss (2004). "Colorization using optimization." In: *Transactions on Graphics (TOG)*. Vol. 23. 3, pp. 689–694.
- Liang, M. and X. Hu (2015). "Recurrent Convolutional Neural Network for Object Recognition." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*, pp. 3367–3375.
- Liao, Y., A. Rubinsteyn, R. Power, and J. Li (2013). "Learning Random Forests on the GPU." In: *NIPS Workshop on Big Learning: Advances in Algorithms and Data Management*.
- Long, J. L., N. Zhang, and T. Darrell (2014). "Do Convnets Learn Correspondence?" In: *Advances in Neural Information Processing Systems (NIPS)*.
- Long, P. and R. Servedio (2010). "Restricted Boltzmann Machines are Hard to Approximately Evaluate or Simulate." In: *Proceedings of the 27th International Conference on Machine Learning*, pp. 703–710.
- Lowe, D. G. (2004). "Distinctive image features from scale-invariant keypoints." In: *International Journal of Computer Vision (IJCV)* 60.2, pp. 91–110.
- Marszatek, M. and C. Schmid (2007). "Accurate object localization with shape masks." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*, pp. 1–8.
- Memisevic, R. (2011). "Gradient-based learning of higher-order image features." In: *International Conference on Computer Vision (ICCV)*. Barcelona, Spain, pp. 1591–1598.
- Michalski, V., R. Memisevic, and K. Konda (2014). "Modeling Deep Temporal Dependencies with Recurrent Grammar Cells." In: *Advances in Neural Information Processing Systems (NIPS)*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, pp. 1925–1933.
- Mikolov, T., K. Chen, G. Corrado, and J. Dean (2013). "Efficient estimation of word representations in vector space." In: arXiv: 1301.3781 [cs.CL].
- Minsky, M. and S. Papert (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.
- Mnih, V. and G. E. Hinton (2010). "Learning to Detect Roads in High-Resolution Aerial Images." In: *European Conference on Computer Vision (ECCV)*, pp. 210–223.
- Müller, A. C. and S. Behnke (2014). "Learning Depth-Sensitive Conditional Random Fields for Semantic Segmentation of RGB-D Images." In: *International Conference on Robotics and Automation (ICRA)*. Hong Kong.

- Müller, A., H. Schulz, and S. Behnke (2010). "Topological Features in Locally Connected RBMs." In: *International Joint Conference on Neural Networks (IJCNN)*.
- Neal, R. (2001). "Annealed importance sampling." In: *Statistics and Computing* 11.2, pp. 125–139.
- Newcombe, R. A., A. J. Davison, S. Izadi, P. Kohli, O. Hilliges, J. Shotton, D. Molyneaux, S. Hodges, D. Kim, and A. Fitzgibbon (2011). "KinectFusion: Real-time dense surface mapping and tracking." In: *International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 127–136.
- Ngiam, J., Z. Chen, P. W. Koh, and A. Y. Ng (2011). "Learning deep energy models." In: *International Conference on Machine Learning (ICML)*, pp. 1105–1112.
- Nguyen A Yosinski J, C. J. (2015). "Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.
- Norouzi, M., M. Ranjbar, and G. Mori (2009). "Stacks of Convolutional Restricted Boltzmann Machines for Shift-Invariant Feature Learning." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.
- Olshausen, B. A. and D. J. Field (1997). "Sparse coding with an overcomplete basis set: a strategy employed by V1?" In: *Vision Res* 37.23, pp. 3311–3325.
- Olshausen, B. A. et al. (1996). "Emergence of simple-cell receptive field properties by learning a sparse code for natural images." In: *Nature* 381.6583, pp. 607–609.
- Osindero, S. and G. Hinton (2008). "Modeling image patches with a directed hierarchy of Markov random fields." In: *Advances in Neural Information Processing Systems (NIPS)*. Ed. by J. Platt, D. Koller, Y. Singer, and S. Roweis. Cambridge, MA: MIT Press, pp. 1121–1128.
- Parker, D. B. (1985). *Learning Logic*. Tech. rep. TR-47. Cambridge, MA: MIT Center for Research in Computational Economics and Management Science.
- Pascanu, R., T. Mikolov, and Y. Bengio (2013). "On the difficulty of training recurrent neural networks." In: *Journal of Machine Learning Research (JMLR)* 28, pp. 1310–1318.
- Pavel, M. S., H. Schulz, and S. Behnke (2015). "Recurrent Convolutional Neural Networks for Object-Class Segmentation of RGB-D Video." In: *International Joint Conference on Neural Networks (IJCNN)*.
- Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Van-

- derplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay (2011). "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12, pp. 2825–2830.
- Pham, V., T. Bluche, C. Kermorvant, and J. Louradour (2014). "Dropout improves recurrent neural networks for handwriting recognition." In: *International Conference on Frontiers in Handwriting Recognition (ICFHR)*.
- Pinheiro, P. H. and R. Collobert (2014). "Recurrent convolutional neural networks for scene labeling." In:
- Raiko, T., H. Valpola, and Y. LeCun (2012). "Deep learning made easier by linear transformations in perceptrons." In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 22. JMLR Workshop and Conference Proceedings. La Palma, Canary Islands, Spain: JMLR W&CP, pp. 924–932.
- Ranzato, M. and G. Hinton (2010). "Modeling pixel means and covariances using factorized third-order Boltzmann machines." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*. San Francisco, CA, USA, pp. 2551–2558.
- Ranzato, M., C. Poultney, S. Chopra, and Y. LeCun (2007). "Efficient Learning of Sparse Representations with an Energy-Based Model." In: *Advances in Neural Information Processing Systems (NIPS)*. Ed. by B. Schölkopf, J. Platt, and T. Hoffman. Cambridge, MA: MIT Press, pp. 1137–1144.
- Razavian, A. S., H. Azizpour, J. Sullivan, and S. Carlsson (2014). "CNN Features off-the-shelf: an Astounding Baseline for Recognition." In: *CVPR DeepVision Workshop*.
- Ren, S., K. He, R. Girshick, X. Zhang, and J. Sun (2015). "Object Detection Networks on Convolutional Feature Maps." In: arXiv: 1504.06066 [cs.CV].
- Riedmiller, M. and H. Braun (1993). "A direct adaptive method for faster backpropagation learning: The RPROP algorithm." In: *Neural Networks*, pp. 586–591.
- Riesenhuber, M. and T. Poggio (1999). "Hierarchical models of object recognition in cortex." In: *Nature neuroscience* 2.11, pp. 1019–1025.
- Rifai, S., G. Mesnil, P. Vincent, X. Muller, Y. Bengio, Y. Dauphin, and X. Glorot (2011a). "Higher Order Contractive Auto-Encoder." In: *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pp. 645–660.
- Rifai, S., P. Vincent, X. Muller, X. Glorot, and Y. Bengio (2011b). "Contractive Auto-Encoders: Explicit Invariance During Feature Extraction." In: *International Conference on Machine Learning (ICML)*. Ed. by L.

- Getoor and T. Scheffer. Bellevue, Washington, USA: ACM, pp. 833–840.
- Rifai, S., P. Vincent, X. Muller, X. Glorot, and Y. Bengio (2011c). “Contractive auto-encoders: Explicit invariance during feature extraction.” In: *International Conference on Machine Learning (ICML)*, pp. 833–840.
- Rodrigues, J., J. Kim, M. Furukawa, J. Xavier, P. Aguiar, and T. Kanade (2012). “6D pose estimation of textureless shiny objects using random ferns for bin-picking.” In: *Intelligent Robots and Systems (IROS), International Conference on*.
- Rumelhart, D., G. Hintont, and R. Williams (1986). “Learning representations by back-propagating errors.” In: *Nature* 323.6088, pp. 533–536.
- Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. (2014). “ImageNet Large Scale Visual Recognition Challenge.” In: arXiv: 1409.0575 [cs.CV].
- Salakhutdinov, R. (2009a). “Learning Deep Generative Models.” PhD thesis. University of Toronto.
- Salakhutdinov, R. (2009b). “Learning in Markov Random Fields using Tempered Transitions.” In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 1598–1606.
- Salakhutdinov, R. and I. Murray (2008). “On the quantitative analysis of deep belief networks.” In: *International Conference on Machine Learning (ICML)*. ACM, pp. 872–879.
- Salakhutdinov, R. (2008). *Learning and Evaluating Boltzmann Machines*. Tech. rep. UTML TR 2008-002. Department of Computer Science, University of Toronto.
- Salakhutdinov, R. and G. Hinton (2009). “Deep Boltzmann machines.” In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 5. JMLR Workshop and Conference Proceedings. Clearwater Beach, FL, USA: JMLR W&CP, pp. 448–455.
- Salakhutdinov, R., A. Mnih, and G. Hinton (2007). “Restricted Boltzmann machines for collaborative filtering.” In: *International Conference on Machine Learning (ICML)*. Corvallis, Oregon: ACM, pp. 791–798.
- Schaul, T., S. Zhang, and Y. Lecun (2013). “No more pesky learning rates.” In: *International Conference on Machine Learning (ICML)*, pp. 343–351.
- Scherer, D., A. Müller, and S. Behnke (2010). “Evaluation of pooling operations in convolutional architectures for object recognition.” In: *International Conference on Artificial Neural Networks (ICANN)*. Springer, pp. 92–101.

- Schulz, H. and S. Behnke (2012a). "Deep Learning: Layer-wise Learning of Feature Hierarchies." In: *Künstliche Intelligenz* 26.4: *Neural Learning Paradigms*.
- Schulz, H. and S. Behnke (2012b). "Learning Object-Class Segmentation with Convolutional Neural Networks." In: *European Conference on Neural Networks (ESANN)*.
- Schulz, H. and S. Behnke (2012c). "Learning Two-Layer Contractive Encodings." In: *International Conference on Artificial Neural Networks (ICANN)*. Superseded by Schulz et al. (2014).
- Schulz, H. and S. Behnke (2014). "Structured Prediction for Object Detection in Deep Neural Networks." In: *International Conference on Artificial Neural Networks (ICANN)*.
- Schulz, H., K. Cho, T. Raiko, and S. Behnke (2013). "Two-Layer Contractive Encodings with Shortcuts for Semi-supervised Learning." In: *International Conference on Neural Information Processings (ICONIP)*. Daegu, Korea. Superseded by Schulz et al. (ibid.).
- Schulz, H., K. Cho, T. Raiko, and S. Behnke (2014). "Two-Layer Contractive Encodings for Learning Stable Nonlinear Features." In: *Neural Networks: Deep Learning of Representations*. Ed. by Y. Bengio and H. Lee. Supersedes Schulz and Behnke (2012c) and Schulz et al. (2013).
- Schulz, H., N. Höft, and S. Behnke (2015a). "Depth and Height Aware Semantic RGB-D Perception with Convolutional Neural Networks." In: *European Conference on Neural Networks (ESANN)*.
- Schulz, H., A. Müller, and S. Behnke (2010a). "Exploiting local structure in Stacked Boltzmann machines." In: *European Conference on Neural Networks (ESANN)*. Superseded by Schulz et al. (2011).
- Schulz, H., A. Müller, and S. Behnke (2010b). "Investigating Convergence of Restricted Boltzmann Machine Learning." In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.
- Schulz, H., A. Müller, and S. Behnke (2011). "Exploiting Local Structure in Boltzmann Machines." In: *Neurocomputing* 74.9. Supersedes Schulz et al. (2010a).
- Schulz, H., B. Waldvogel, R. Sheikh, and S. Behnke (2015b). "CURFIL: Random Forests for Image Labeling on GPU." In: *International Conference on Computer Vision Theory and Applications (VISAPP)*. Superseded by Schulz et al. (2016).
- Schulz, H., B. Waldvogel, R. Sheikh, and S. Behnke (2016). "CURFIL: A GPU library for Image Labeling with Random Forests." In: *Computer Vision, Imaging and Computer Graphics. Theory and Application*. Communications in Computer and Information Science. Springer. Supersedes Schulz et al. (2015b).

- Schwarz, M. (2014). "Objektklassifikation, Identifizierung und Posen-schätzung mit Hilfe von vortrainierten Konvolutionsnetzen. Bachelor thesis." University Bonn.
- Schwarz, M., H. Schulz, and S. Behnke (2014). "RGB-D Object Recognition and Pose Estimation based on Pre-trained Convolutional Neural Network Features." In: *International Conference on Robotics and Automation (ICRA)*.
- Sermanet, P., D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun (2013). "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks." In: arXiv: 1312.6229 [cs.CV].
- Shannon, C. (1949). "The synthesis of two-terminal switching circuits." In: *Bell System Technical Journal* 28.1, pp. 59–98.
- Sharp, T. (2008). "Implementing decision trees and forests on a GPU." In: *European Conference on Computer Vision (ECCV)*, pp. 595–608.
- Shotton, J., A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake (2011). "Real-time human pose recognition in parts from single depth images." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.
- Shotton, J., J. Winn, C. Rother, and A. Criminisi (2006). "Textonboost: Joint appearance, shape and context modeling for multi-class object recognition and segmentation." In: *European Conference on Computer Vision (ECCV)*.
- Shotton, J., M. Johnson, and R. Cipolla (2008). "Semantic texton forests for image categorization and segmentation." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.
- Shotton, J., T. Sharp, A. Kipman, A. Fitzgibbon, M. Finocchio, A. Blake, M. Cook, and R. Moore (2013). "Real-time human pose recognition in parts from single depth images." In: *Communications of the ACM*.
- Silberman, N., D. Hoiem, P. Kohli, and R. Fergus (2012). "Indoor segmentation and support inference from RGBD images." In: *European Conference on Computer Vision (ECCV)*.
- Silberman, N. and R. Fergus (2011). "Indoor scene segmentation using a structured light sensor." In: *Computer Vision Workshops (ICCV Workshops)*.
- Simonyan, K. and A. Zisserman (2014). "Two-Stream Convolutional Networks for Action Recognition in Videos." In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 568–576.
- Slat, D. and M. Lapajne (2010). "Random Forests for CUDA GPUs." PhD thesis. Blekinge Institute of Technology.

- Smolensky, P. (1986). "Information Processing in Dynamical Systems: Foundations of Harmony Theory." In: *Parallel Distributed Processing*. Vol. 1. Chap. 6, pp. 194–281.
- Spall, J. C. (1998). "An overview of the simultaneous perturbation method for efficient optimization." In: *Johns Hopkins APL Technical Digest* 19.4, pp. 482–492.
- Spinello, L. and K. O. Arras (2011). "People detection in RGB-D data." In: *Intelligent Robots and Systems (IROS), International Conference on*. IEEE.
- Springenberg, J. T., A. Dosovitskiy, T. Brox, and M. Riedmiller (2014). "Striving for Simplicity: The All Convolutional Net." In: *International Conference on Learning Representations (ICLR)*. arXiv: 1412.6806 [cs.LG].
- Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). "Dropout: A simple way to prevent neural networks from overfitting." In: *Journal of Machine Learning Research (JMLR)* 15, pp. 1929–1958.
- Stückler, J., N. Birešev, and S. Behnke (2012). "Semantic Mapping Using Object-Class Segmentation of RGB-D Images." In: *Intelligent Robots and Systems (IROS), International Conference on*.
- Stückler, J., B. Waldvogel, H. Schulz, and S. Behnke (2014). "Dense Real-Time Mapping of Object-Class Semantics from RGB-D Video." In: *Journal of Real-Time Image Processing*.
- Sundermeyer, M., R. Schlüter, and H. Ney (2012). "LSTM Neural Networks for Language Modeling." In: *Interspeech*, pp. 194–197.
- Sutskever, I. (2013). "Training Recurrent Neural Networks." PhD thesis. University of Toronto.
- Szegedy, C., A. Toshev, and D. Erhan (2013). "Deep Neural Networks for Object Detection." In: *Advances in Neural Information Processing Systems (NIPS)*.
- Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2015). "Going Deeper with Convolutions." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.
- Szegedy, C., W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus (2014). "Intriguing properties of neural networks." In: *International Conference on Learning Representations (ICLR)*. arXiv: 1312.6199 [cs.CV].
- Taskar, B., V. Chatalbashev, D. Koller, and C. Guestrin (2005). "Learning structured prediction models: A large margin approach." In: *International Conference on Machine Learning (ICML)*.

- Taylor, G., R. Fergus, Y. LeCun, and C. Bregler (2010). "Convolutional learning of spatio-temporal features." In: *European Conference on Computer Vision (ECCV)*, pp. 140–153.
- Tieleman, T. (2008). "Training restricted Boltzmann machines using approximations to the likelihood gradient." In: *International Conference on Machine Learning (ICML)*. Helsinki, Finland: ACM, pp. 1064–1071.
- Tieleman, T. and G. Hinton (2009). "Using fast weights to improve persistent contrastive divergence." In: *International Conference on Machine Learning (ICML)*. New York, NY, USA: ACM, pp. 1033–1040.
- Uijlings, J., K. van de Sande, T. Gevers, and A. Smeulders (2013). "Selective search for object recognition." In: *International Journal of Computer Vision (IJCV)* 104.2.
- Van der Maaten, L. and G. Hinton (2008). "Visualizing data using t-SNE." In: *Journal of Machine Learning Research (JMLR)* 9, pp. 2579–2605.
- Van der Walt, S., S. C. Colbert, and G. Varoquaux (2011). "The NumPy array: a structure for efficient numerical computation." In: *Computing in Science & Engineering* 13.2, pp. 22–30.
- Van Essen, B., C. Macaraeg, M. Gokhale, and R. Prenger (2012). "Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?" In: *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.
- Vatani, T., T. Raiko, H. Valpola, and Y. LeCun (2013). "Pushing Stochastic Gradient towards Second-Order Methods – Backpropagation Learning with Transformations in Nonlinearities." In: *arXiv:1301.3476 [cs.LG]*.
- Vincent, P., H. Larochelle, I. Lajoie, Y. Bengio, and P. Manzagol (2010). "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion." In: *Journal of Machine Learning Research (JMLR)* 11, pp. 3371–3408.
- Vincent, P. (2011). "A connection between score matching and denoising autoencoders." In: *Neural Computation* 23.7, pp. 1661–1674.
- Viola, P. and M. Jones (2001). "Rapid object detection using a boosted cascade of simple features." In: *Computer Vision and Pattern Recognition (CVPR), Conference on*.
- Wehenkel, L. and M. Pavella (1991). "Decision trees and transient stability of electric power systems." In: *Automatica* 27.1, pp. 115–134.
- Welling, M. and G. E. Hinton (2002). "A new learning algorithm for mean field Boltzmann machines." In: *Artificial Neural Networks*, pp. 351–357.

- Werbos, P. J. (1974). "Beyond regression: New tools for prediction and analysis in the behavioral sciences." PhD thesis. Boston, MA: Harvard University.
- Weston, J., F. Ratle, and R. Collobert (2008). "Deep learning via semi-supervised embedding." In: *International Conference on Machine Learning (ICML)*. Helsinki, Finland, pp. 1168–1175.
- Wiskott, L. and T. Sejnowski (2002). "Slow feature analysis: Unsupervised learning of invariances." In: *Neural Computation* 14.4, pp. 715–770.
- Yuille, A. (2005). "The Convergence of Contrastive Divergences." In: *Advances in Neural Information Processing Systems (NIPS)*. Ed. by L. K. Saul, Y. Weiss, and L. Bottou, pp. 1593–1600.
- Zeiler, M. D. and R. Fergus (2014). "Visualizing and understanding convolutional networks." In: *European Conference on Computer Vision (ECCV)*, pp. 818–833.
- Zeiler, M., G. Taylor, and R. Fergus (2011). "Adaptive deconvolutional networks for mid and high level feature learning." In: *International Conference on Computer Vision (ICCV)*. Barcelona, Spain, pp. 2018–2025.
- Zhang, Y., K. Sohn, R. Villegas, G. Pan, and H. Lee (2015). "Improving object detection with deep convolutional networks via bayesian optimization and structured prediction." In: arXiv: 1504.03293 [cs.CV].
- Zhu, X., C. Vondrick, D. Ramanan, and C. Fowlkes (2012). "Do We Need More Training Data or Better Models for Object Detection?" In: *British Machine Vision Conference*.