Universiteit Leiden
Opleiding ICT in Business

On the Impact of Data Set Size

in Transfer Learning using Deep Neural Networks

Name:	Deepak Soekhoe
Date:	25/08/2016
1st supervisor:	Peter van der Putten
2nd supervisor:	Aske Plaat

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

List of Figures

Acknowledgements

I would like to thank Aske Plaat for giving me the motivation to go with my instincts, and pick a thesis subject in artificial intelligence. I am also grateful to Jeroen Unger for giving me the freedom to investigate deep learning during my internship at Microsoft. Furthermore, a big thanks to Maarten de Jonge for helping me install the Caffe framework, which ended up to be a quite tedious process. Finally, I would like to express my appreciation and gratitude to my supervisor Peter van der Putten who was willing to take on this project, and supported me during the entire process. Thank you.

Abstract

In this thesis we study the effect of target set size on transfer learning in deep learning convolutional neural networks. This is an important problem as labelling is a costly task, or for new or specific classes the number of labelled instances available may simply be too small. We first discuss feedforward neural networks and convolutional networks to provide some context. In the main section of the thesis we present results for a series of experiments where we either train on a target of classes from scratch, retrain all layers, or subsequently lock more layers in the network, for the Tiny-ImageNet and MiniPlaces2 data sets. Our findings indicate that for smaller target data sets freezing the weights for the initial layers of the network give better results on the target set classes. We present a simple and easy to implement training heuristic based on these findings, and provide interesting directions for future research.

Chapter 1

Introduction

Current deep learning models such as convolutional neural networks and recurrent neural networks achieve state-of-the-art performance on all benchmark data sets. One can speak of a Renaissance of the artificial neural network algorithm class. While in the 1990s, neural networks went out of fashion due to limited computing power and lack of data, currently they are arguably the most popular learning algorithms in machine learning. Both industry and academia are highly interested in this branch of models as evidenced by the new research labs focused on deep learning, from the largest tech giants in the market.

With regards to computer vision tasks, the convolutional neural networks perform the best [25, 63, 66, 77]. Modern models make use of deep convolutional neural networks (CNN) such as AlexNet [38]. However, training these models on large data sets such as ImageNet [11] can take up a significant amount of time, and the number of labelled examples per class available may be limited, so learning from scratch has its downsides. One approach to overcome this problem is to use transfer learning. The objective of transfer learning is to use knowledge of a source task and *transfer* that to a new target task [49]. It provides considerable benefits over learning from scratch (i.e. from a random initialisation of the weights). One obvious advantage is that a model can learn more efficiently since it starts with a pre-initialised weight matrix.

In their study, Yosinski *et al.* [74] trained AlexNet on the ImageNet data set and found that the first three layers in a CNN contain generic and reusable features. Beyond the third layer, the features gradually become more specific with respect to the source data set. However, the authors did not take into account the size of the target data set, on which the model with the transferred features will be trained.

The size of the target data set plays an important role, since it affects how much impact transfer learning will have on the performance. Thus, it is logical to ask how well extracted features generalise to smaller data sets. It would be helpful to know at what data set size transfer learning would be still beneficial. More specifically, at what layer is the model still able to generalize to a small data set size? Therefore, it is of both academic and practical interest to investigate at what *target data set* size transfer learning can still provide any additional value. Furthermore, Yosinski *et al.* only used the ImageNet data set [74]. It would be interesting to find out whether the transfer learning properties are different when using a data set from a different domain.

In this work we will expand the study by [74], and measure the effect of target data set size on the transferability of parameters in convolutional neural networks. Our main contribution is to quantify the extent to which features are able to generalise to the target data set when we systematically reduce its size. We will investigate this for each individual layer by evaluating the accuracy as a function of the data set size. We will have three variants of this. First, we will obtain a base score, without applying any form of transfer learning. In the second condition we will completely fine-tune all the layers of the network. In the third one, we will freeze the transferred features per individual layer. We will investigate this for different sizes of the target

set. Moreover, we will test this on two different subsets of data sets, each with a different domain, ImageNet and Places2.
The rest of the thesis is structured as follows. We first describe how the neural network algorithm works, and next dive into the backpropagation and gradient descent algorithm. This provides the fundamental building blocks of the convolutional neural networks. In the following chapter we discuss in the detail convolutional neural networks and its operations. Moreover, we briefly discuss another important deep neural network model, the recurrent neural network. In chapter four we describe the methods used, followed by a discussion of relevant related work. In chapter six we describe in detail our experimental approach based on our methodology, and give information about the data sets used. In chapter seven we present the results, followed by a discussion of its implications. We end the thesis by providing a conclusion.

Chapter 2

Feedforward Neural Networks

In this chapter we will describe feedforward neural networks of the artificial neural network (ANN) class, backpropagation and gradient descent. These three algorithms provide the fundamental structure for convolutional networks.
The first ideas about ANNs were developed in 1943 by McCulloch and Pitts [44]. They proposed mathematical concepts of neurons and the "all-or-nothing" firing principle, found in neurons in the physiological brain. Thus if the input reached a certain threshold value, the model would output a one, otherwise it would stay zero. This idea got further expanded by Rosenblatt [56] who developed the "perceptron" in 1958 inspired by the work in [26]. The perceptron has weights, which allows it to "learn" a function. Moreover, using a method called ADALINE [72], these weights could be adjusted. However, the main disadvantage of these perceptrons was that they could only solve linearly separable problems, as described by Minsey & Papert [45]. This meant that multiple layer perceptrons (MLP) were infeasible. Perceptrons could not solve the boolean XOR function, and interest in ANN faded. Backpropagation, developed by Werbos [71] in 1974, and later popularized by Rumelhart *et al.* [73] solved this issue and interest in ANNs rose again during the 1980s. Moreover, during the 1970s and 1980s research was done on the gradient descent optimization in MLPs [39, 52, 58, 71]. This type of ANN is called the *feedforward neural network* (FNN), and is arguably the most simple type of ANN. There exist many variations on the traditional model of FNNs such as self organizing maps, spiking neural networks and radial basis function networks.

2.1 FNN computation

The feedforward neural network (FNN) lies at the core of convolutional neural networks. They are called *neural* networks since they are loosely inspired by the human brain. Given a labeled dataset $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ FNNs take an n-dimensional input vector and compute a predicted output value $h_{W,b}(x)$. This model is parametrized by the weights of the network W and a bias term b. In figure 2.1 we see a simple FNN with three layers, an input, hidden and output layer respectively. This network has three inputs x_1, x_2 and, x_3. The hidden layer consists of three nodes, a_1^2, a_2^2 and a_3^2 which represent the hidden activations. Finally, the output layer l has a single node a_1^3 which represents the predicted output value by the net. Concretely, the FNN computes a weighted sum of the inputs, z_i^l, as follows [65]:

$$z_i^l = \sum_{j=1}^{s_l} W_{ij}^l a_j^l + b_i^l \tag{2.1}$$

The weighted sum z_i^l then goes through an activation or transfer function, $f(\cdot)$, to finally output activation a_i^l:

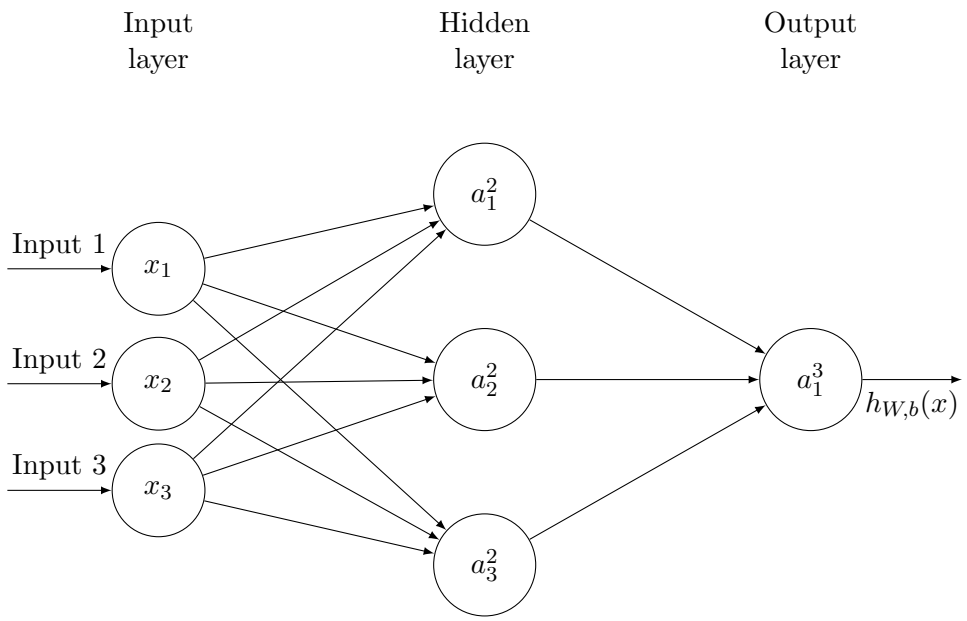


Figure 2.1: Schematic model of a feedforward neural network (We did not include the bias term in this illustration for simplicity).

$$a_i^l = f(z_i^l) = f(\sum_{j=1}^{s_l} W_{ij}^l a_j^l + b_i^l) = f(W^T x) \tag{2.2}$$

The activation function is commonly a sigmoid function $f(x) = (1 + e^{-x})^{-1}$. To compute a_1^3 in figure 2.1 we first calculate the activations of the hidden layer as follows:

$$a_1^2 = f(W_{10}^1 x_0 + W_{11}^1 x_1 + W_{12}^1 x_2 + W_{13}^1 x_3) \tag{2.3}$$
$$a_2^2 = f(W_{20}^1 x_0 + W_{21}^1 x_1 + W_{22}^1 x_2 + W_{23}^1 x_3) \tag{2.4}$$
$$a_3^2 = f(W_{30}^{(1} x_0 + W_{31}^1 x_1 + W_{32}^1 x_2 + W_{33}^1 x_3) \tag{2.5}$$

Finally, we can compute the output:

$$h_{W,b}(x) = a_1^3 = f(W_{10}^2 a_0^2 + W_{11}^2 a_1^2 + W_{12}^2 a_2^2 + W_{13}^2 a_3^2) \tag{2.6}$$

This process is also called the *forward pass* in the back-propagation algorithm (see section 2.4).

2.2 Cost function

We use a mean squared error cost function, $J(\theta)$ (2.8) (where θ denotes the weight matrix of the network) to determine how well the algorithm fits the training data. The objective is to minimize $J(\theta)$, by adjusting the set of weights. In other words, we try to minimize the difference or error between the predicted value $h_{W,b}(x)$, and actual output y for each training example (x^n, y^n). Here we use a cost function for regression, where the outputs of the network are real-valued numbers as opposed to classes in a classification task. For a single training example the cost function looks like (2.7), however when using n examples, we average over them to compute the cost (2.8).

One disadvantage of this cost function is that its error surface is non-convex. Therefore, the optimization algorithm might get stuck in local minima and not converge properly.

$$J(\theta) = \frac{1}{2}\|h_{W,b}(x) - y\|^2 \tag{2.7}$$

$$J(\theta) = \left[\frac{1}{n}\sum_{k=1}^{n}(\frac{1}{2}\|h_{W,b}(x) - y\|^2)\right] \tag{2.8}$$

When doing multi-class classification with K possible output classes, a common cost function to use is the cross-entropy loss:

$$J(\theta) = -\left[\sum_{i=1}^{n}\sum_{k=1}^{K} 1\left\{y^i = k\right\} \log \frac{\exp(\theta^{k\top}x^i)}{\sum_{j=1}^{K}\exp(\theta^{j\top}x^i)}\right] \tag{2.9}$$

2.3 Gradient descent

Various optimization algorithms exist to minimize our cost function, however the most used one is *gradient descent* (GD) (1). Using GD we update our weights by computing the gradient of the cost function with respect to the weights denoted by $\frac{\partial}{\partial\theta_j}J(\theta_j)$, where θ_j is the current weight to be updated, and multiplying that quantity by a learning rate α. This learning rate affects how large the weight update will be. If we decide on a small value for α then it might take a long time to converge. However, setting this hyper parameter too big GD might overshoot the minimum and fail to converge, or even diverge. Finally, we update the weights by moving in the negative direction of the gradient (2.13, 2.14). This process gets repeated until training gets terminated.

Algorithm 1 Gradient Descent

REPEAT UNTIL CONVERGENCE
  for $j = 1, \ldots, n$ **do**
    $\theta_j := \theta_j$ - α $\frac{\partial}{\partial\theta_j}J(\theta_j)$
  end for

2.3.1 Gradient descent schemes

There are generally three strategies for updating the parameters θ using GD, *batch GD* (2.10), *stochastic GD* (2.10) and *mini-batch GD* (2.10).
In batch GD we calculate the gradients with respect to θ, denoted by $\nabla_\theta J(\theta)$, for all the training examples and then do one update for θ. However, when the dataset is large, computing the gradients for every instance in the data set is not efficient and can be very slow. In contrast, stochastic GD updates θ after each single training example, where $\nabla_\theta(J(\theta, x^i, y^i)$ indicates the gradient. This method greatly reduces training time since we can find the minimum much quicker due to the frequency of the updates. However, the behaviour of the cost function is less consistent and oscillates significantly. Finally, we can take an approach which lies in between the aforementioned strategies, and use mini-batch GD where we take a mini-batch of size n training examples and compute the gradient, $\nabla_\theta(J(\theta, x^{i+n}, y^{i+n})$ from that batch. For deep learning models mini-batch GD is a popular approach.

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta) \tag{2.10a}$$

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta, x^i, y^i) \tag{2.10b}$$

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta, x^{i+n}, y^{i+n}) \tag{2.10c}$$

A way to optimize GD is by adding a hyper parameter *momentum* η, which increases the rate of convergence. When the error surface of the cost function contains steep narrow valleys or ravines, GD will oscillate across such a ravine very slowly. Adding the momentum term will significantly reduce oscillations along the ravine by creating larger weight updates [54, 57]. Specifically, we update the weights by adding a fraction of the gradient from the previous time step, denoted by v_{t-1}. This term is controlled by a hyper parameter η. Finally, we subtract the resulting term v_t from θ to perform the update.

$$v_t = \alpha \nabla_\theta J(\theta, x^i, y^i) + \eta v_{t-1} \tag{2.11}$$
$$\theta = \theta - v_t \tag{2.12}$$

Alternative methods to optimize GD include Nesterov accelerated gradient (NAG) [46], Adagrad [13], Adadelta [76], Adam [37] and RMSprop [28]. However, in our experiments we only use momentum.
Other optimization algorithms have proven to be faster and more stable than GD, such as conjugate gradient (CG) and Limited-memory BFGS (L-BFGS) [47]. This is due their ability to run in a distributed fashion in combination with a line search schema. However, these are more complex to implement.

2.4 Backpropagation algorithm

In order to compute the gradients of the cost function $J(\theta)$ with regards to the parameters, we use a method called *backpropagation* [57, 58]. Backpropagation consists of two phases, the forward pass and the backward pass. In the forward pass we compute the activations $a_i^{(l)}$ for every neuron (2.2). In the backward pass we compute an "error" term δ_i^l (C.6) for every unit i in layer l, starting with the output layer n_l, and progressively move further back into the network (see Appendix C for full derivation). Once we have the terms a_i^l and δ_i^l we can compute the partial derivatives (see algorithm 2).
Now that we have computed the gradients via backpropagation we can update our parameters, where W_{ij}^l is the weight for the j^{th} neuron in the $l-1^{th}$ layer, going to the i^{th} neuron in l^{th} layer. Similarly, b_i^l is the weight that corresponds to the bias neuron in layer $l-1$ that is going to the i^{th} neuron in layer l layer.

$$W_{ij}^l = W_{ij}^l - \alpha \frac{\partial}{\partial W_{ij}^l} J(\theta) \tag{2.13}$$

$$b_i^l = b_i^l - \alpha \frac{\partial}{\partial b_i^l} J(\theta) \tag{2.14}$$

Algorithm 2 Backpropagation

FORWARD PASS

Compute the activations a_i^l for all nodes i in layer l in the network

BACKWARD PASS

for unit i in output layer o_l **do**

$\delta_i^{o_l} = \frac{\partial J}{\partial a_i^{o_l}} \frac{\partial a_i^{o_l}}{\partial z_i^{o_l}} = (a_i^{o_l} - y_i) f'(z_i^{o_l})$

{ $\delta_i^{o_l}$ denotes the error in the output layer}

end for

for nodes i in layers $l = o_l - i$ **do**

$\delta_i^{(l)} = \sum_{j=1}^{s_{l+1}} \delta_j^{l+1} W_{ij}^l f'(z_i^l)$

end for

COMPUTE GRADIENTS

$\frac{\partial}{\partial W_{ij}^l} J(\theta) = a_i^l \ \delta_i^{l+1}$

$\frac{\partial}{\partial b_i^l} J(\theta) = \delta_i^{l+1}$

Chapter 3

Deep Learning

In the main part of this chapter we describe in detail how convolutional neural networks operate. This include the convolutional layer, the pooling layer and how backpropagation works. Furthermore, we discuss recurrent neural networks which are used for predicting sequences of data.
Convolutional neural networks (CNNs) are the most popular deep learning (DL) models for computer vision tasks [8, 19, 36, 60, 67, 69]. Whereas previously, features had to be manually engineered, with CNNs the model learns representations on its own. The model takes raw input pixels and transforms them into internal representation in each of the subsequent layers. The deeper we get in the model, the more abstract the features become. They achieve state-of-the-art results on benchmark datasets such as ImageNet [11] (see figure 3.1). The reason that in 2012 DL took off can be contributed to two factors. The first one is the availability of large labeled datasets, specifically ImageNet, which contains 1.2 million labeled images distributed over 1,000 classes. Deep learning models generally perform better with more data. And two, there is more computing power available. The advances in graphics processing units (GPUs) allowed DL models to train much faster and efficiently, in particular technology company NVIDIA[1] which specializes in GPU production, played an important role in the development of DL. DL models can contain millions of parameters and training them using CPUs is not manageable. Using GPUs makes training time orders of magnitude faster due to their parallel architecture. Thus, sufficient amount of labeled data and an increase in computational power caused the DL inflexion point.

3.1 Traditional computer vision

In traditional computer vision features had to be "hand engineered." However, the problem was that these features never had the representational power to be suited for more complex computer vision tasks. While research on computer vision began in the 1960s [50] as a summer project at MIT, it only took off in the late 1990s with algorithms such as *SIFT* (Scale Invariant Feature Transform), [42] or later HOG (Histogram of Oriented Gradients) [10] and *SURF* (Speeded Up Robust Features) [4]. These algorithms act as feature and object detectors. Moreover, algorithms to compute or detect edges and blobs (i.e. areas where the brightness is above or below a value) were needed as well. Furthermore, researchers required experience to decide which features the most optimal to use. In sum, there was a significant amount of feature engineering necessary. This is in stark contrast with current techniques, where we just need one model to compute all the features automatically.

[1] `http://www.nvidia.com/`

3.2 Background CNN

CNN models are characterized by local connections, spare connectivity, shared weights, and the use of many layers in a hierarchical manner. Similar to neural networks, CNNs found their inspiration from biology as well. In their seminal paper Hubel & Wiesel [32] discovered that visual stimulations are processed in a hierarchical fashion by the visual cortex. Neurons in the lower area of the cortex respond to simple features such as edges and lines. Cells in the higher area of the visual cortex are activated by more complex features, such as combinations of edges and lines. Fukushima [15] proposed the neocognitron neural network model which was inspired by the work of Hubel and Wiesel and can be seen as the precursor of modern CNNs. Lecun *et al.* [40] developed the first successful CNN model (LeNet-5) which was able to recognize handwritten digits (see 3.2). The model consists of 7 layer, where the first four alternate between convolutions and pooling layers. The final three consist of of fully connected layers followed by the output layer. However, we note that LeNet-5 is almost identical to modern CNNs, the main differences are that modern CNNs contain more layers and have more data to train on. Advances in computing power have greatly contributed to the rise of deep CNNs as well.

3.3 Convolutional layer

In this section we will describe what the convolutional layer does. Mathematically, convolution computes the amount of overlap of a function g when this is slided over another function f where f and t are real-valued [5]:

$$h(t) = (f * g)(t) = \int f(a)g(t-a)da \tag{3.1}$$

In CNNs the convolution operation occurs with *filters* or *kernels* over all spatial locations of the input image. These filters can be seen as a matrix of parameters and this is what the model learns. A filter is a small patch of size w x h x d, where w and h represent the spatial width and height respectively, and d represents the dimension of the filter. It slides across the input image of size W_j x H_j x D_j, where W_j and H_j are the width and height of the input image, and D_j the dimension of the image. For colour images D_j would be equal to three, since the image would have RGB channels. The filter then computes a matrix multiplication (the Hadamard product) between the its spatial dimensions and the current image region. From the resulting matrix we sum all elements and return a single activation value for that particular region of the image. Since the filter moves along the whole image, we compute the activation for every location of the image, which results in an *activation map* of size W_{j+1} x H_{j+1} x D_{j+1}. We can compute the size of each activation map with the following formula:

$$W_{j+1} = \frac{(W_j - F + 2P)}{S+1} \tag{3.2}$$

$$H_{j+1} = \frac{(H_j - F + 2P)}{S+1} \tag{3.3}$$

$$D_{j+1} = K \tag{3.4}$$

Note that is very similar to regular feedforward networks, however in this case we have a whole activation map, rather than just a single activation neuron. This process is repeated with K filters, to obtain K activation maps. The filter moves along the input with a certain step size called stride, S. The stride dictates how many pixels the filter moves across the input. Furthermore, we can zero-pad the activation maps with hyper parameter P, in order to retain the spatial dimension. Every time we perform a convolution, the dimensions of the activation maps

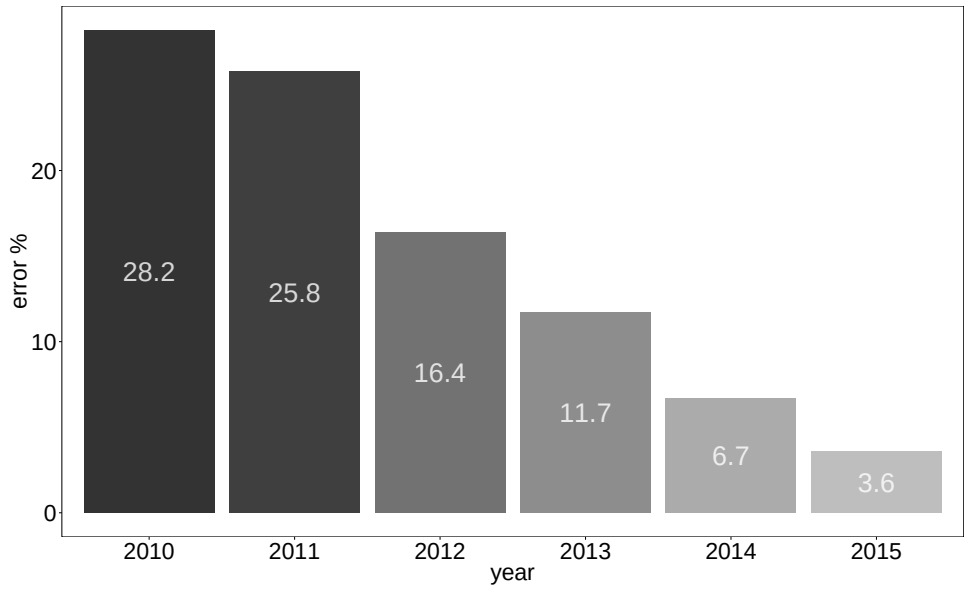


Figure 3.1: ImageNet Large Scale Visual Recognized Challenge (ILSVRC) error scores for the ImageNet object classification task. There is a significant drop in error in 2012, the year Hinton *et al.* used a deep CNN [38] and won the competition. The winning teams for the following years were Clarifai (2013), Google (2014) [66] and Microsoft Research Asia (2015) [25].

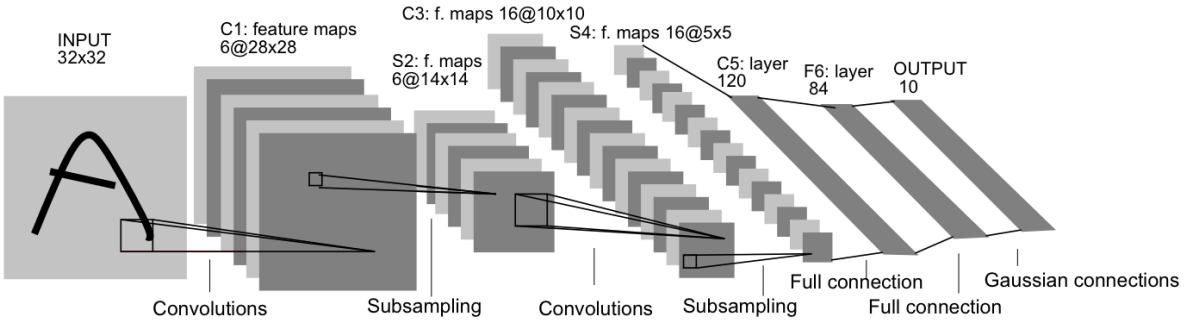


Figure 3.2: LeNet-5 architecture (illustration taken from [40]).

shrink in size however this leads to smaller representations the deeper we go into the network. In appendix E we provide the backpropagation algorithm for CNNs.

3.3.1 Sparsity and weight sharing

In normal neural networks every neuron connects to all the layers in the hidden layer, however CNNs use sparse connectivity. This means that not all the weights are connected to the input. This is necessary in order to reduce the parameters in the network. An image with a dimension of 500 x 500 x 3 would have 750,000 parameters. It would become unmanageable to train such a dense network. Moreover, each activation map has identical weights. This is done with the assumption that a feature at one location, for example an edge, might also be useful at another location.

3.3.2 ReLU

CNNs do not employ a sigmoid activation function, rather a rectified linear unit (ReLU) is used $f(z) = max(0, z)$. ReLUs output zero for every activation that is zero or lower, when the activation is higher than zero it returns the activation value. The main reasons for choosing ReLU is one, they do not suffer from the vanishing gradient problem [30]. The sigmoid activation functions outputs values a where $a \in [0, 1]$. However, due to these small values, the gradients will become small and reach zero, or "vanish," during backpropagation in deep networks. Thus, the problem becomes that these parameters do not get updated. And two, it creates sparsity in the network. Since activations that are equal or smaller than zero return zero, the network will contain activations with value zero. This will make the network less parameter dense and speeds up learning.

Batch normalization Initialization of the parameters in deep forward can make a significant difference in the performance of the network. The authors in [33] introduce a method called *batch normalization.* This initialization strategy reduces the internal covariate shift by normalizing the inputs.

3.3.3 Pooling and Fully-Connected layer

After the convolution layer each activation map undergoes a downsampling process. The pooling or subsample layer reduces the dimension W_{j+1} x H_{j+1} of the activation map by splitting the activation map into regions of m x n. Over these regions we take the maximum, minimum or average value of the parameters. This is done to reduce the number of weights which speeds up computing, and it also prevents overfitting (see figure 3.4). Finally, after several convolution and pooling layers CNNs contain one or two fully connected layer (FC). Similarly to a regular neural network, in the FC layer all the neurons are fully connected to the nodes in the previous layer. This part of the network contains the majority of the weights. The only layer which follows after the FC is the output layer where the CNN makes its classification.

3.3.4 Dropout

A popular method to prevent overfitting in deep neural networks is *dropout* [29]. The idea is to stochastically remove or "drop," the activations and their weights to zero with a probability p = 0.5 during training. In deep neural networks neurons will co-adapt, therefore dropping units will break their relationships and improve generalization. A neural net with n nodes, results in 2^n possible models with dropout. Each training example gets fed into such a network. Rather than averaging each outcome for each model in the test phase, the weights that were preserved

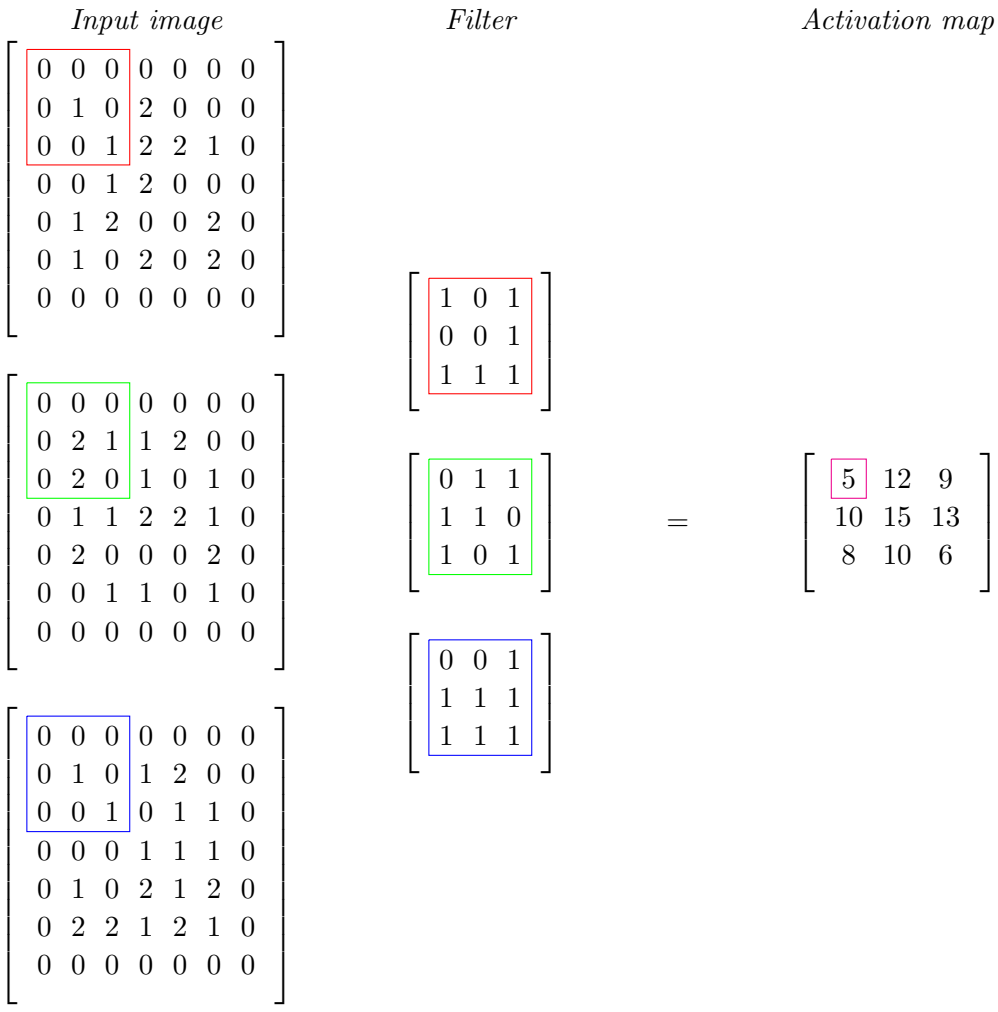


Figure 3.3: The convolution operation. In this figure we take an input image of size 7 x 7 x 3 and convolve it with a filter of size 3 x 3 x 3 , where $K = 1$ and stride $S = 2$. We apply a padding of $P = 1$. Since $K = 1$, we end up with one activation map of size 3 x 3 x 1. This activation will be the input for the following filter. We note that only the first layer of filters have access to the raw input pixels. The filters in the hidden layers convolve on the activation maps. The weight sharing scheme allows the layer to have 3 x 3 x 3 x 1 = 27 parameters connecting to the activation map, rather than 27 x 9 = 243. We omit the bias node for simplicity.

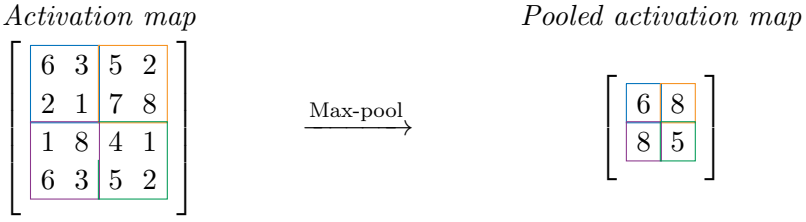


Figure 3.4: The max-pool operation. The activation map is divided into regions of 2 x 2, and from these regions we take the maximum value. We end up with a pooled activation map which contains fewer parameters.

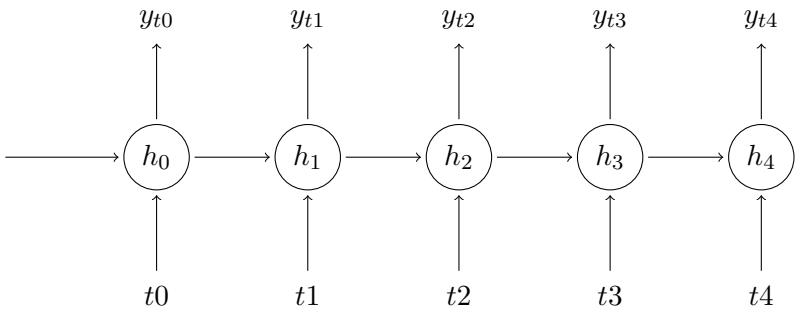


Figure 3.5: RNN hidden state h_i computation time line. Each hidden state get forwarded in the next time step.

during training will get multiplied by that probability p. A modification of dropout is adaptive dropout [3], where p is different for the different hidden nodes and depends on the activations. The study showed that this method of dropout is more effective than regular dropout.

3.4 Recurrent Neural Networks

Modelling sequences of data cannot be done using a CNN, instead *recurrent neural networks* (RNNs) [21] are used. In RNNs the hidden nodes gets fed back into the input nodes.Thus, RNNs contain a loop where information gets "stored." For instance, this is useful when the task is to predict the next word in a sentence. In such a case it would be useful to know which words came before it in order to make a proper prediction.
RNNs have proven to be extremely powerful and can be used to do language translation [2], text prediction [27], image captioning [35], speech recognition [21], image generation [23] or even to evaluate short lines of programming code [75]. Since we are modeling sequences, there is a temporal factor t involved. In its simplest form, a RNN can be formulated by 3.6, where h_t is the current hidden state, h_{t-1} is the state of the previous time step, $\theta_{xh}x_t$ is the current input, y_t is the output, and where g is usually a hyperbolic tangent transfer function [5, 20].

$$h_t = g(\theta_{hh}h_{t-1} + \theta_{xh}x_t) \tag{3.5}$$
$$y_t = \theta_{hy}h_t \tag{3.6}$$

In figure 3.5 we can see a high level view of how at each time step t_i the hidden state h_i gets propagated to the next time step t_{i+1}. Moreover, at each time step a sequenced output y_{ti} gets computed.
However, RNNs are trained with backpropagation which makes it susceptible to the vanishing gradient problem, where the weights of the gradients will become very small in the early layers of a deep feedforward networks [30]. This happens because backpropagation has to be performed over many steps back through time. Moreover, the gradients will grow exponentially when the parameters are large [6]. In [53] a formal description is provided on why this problem occurs (see appendix D). Several solutions are Hessian Free optimization or using echo state machines. However, the most common method to these problems are the Long Short Term Memory (LSTM) networks.

3.4.1 Long Short Term Memory networks

The LSTM model was developed by Schmidhuber & Hochreiter [31] and consists of three main "gates," an input (i_t), output (o_t) and a forget gate (o_t) that interact on modules called *memory*

cells (C^l) which store information. These cells have the capability to preserve information for a long time, and they determine what the activation of the new hidden states will be.
For each gate, the hidden activation of the previous time step h_{t-1} and the current input x_t get multiplied with the weights W, and fed through a sigmoid function σ. This output is a quantity between zero and one and will affect the value of the memory cell C_t^l by scaling it. The input gate together with equation g can add something to the cell. The forget gate determines how much of the previous value of the cell C_{t-1}^l will be added to the current value by taking the element wise product. Finally, to compute the activations of the new hidden layer h_t, the value of the cell get fed through a hyperbolic tangent function and get element wise multiplied with the final output gate o_t. Thus, the LSTM learns with gradient descent and backpropagation, how its gates must scale the values of the cell in order to compute the hidden activation state. Moreover, there are many variants of the LSTM network. In [22] the authors analyze in detail eight different LSTM models. However, the study found that none of the models significantly outperform the original LSTM architecture.

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{3.7}$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{3.8}$$
$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{3.9}$$
$$g = tanh(W_g[h_{t-1}, x_t] + b_g \tag{3.10}$$
$$C_t^l = f_t \odot C_{t-1}^l + i_t \odot g \tag{3.11}$$
$$h_t = tanh(C_t^l) \odot o_t \tag{3.12}$$

Chapter 4

Methods

In the first part of this chapter we give a description of the overall methodological approach we took. In the second part of the chapter we provide an overview of related works of transfer learning in deep neural networks.

4.1 General approach

We will transfer features from a CNN trained on a source task, to a target task, i.e. data sets with disjunct outcome classes. We will consider the scenario where the target data set is the same size, as well as smaller in size as the source data set. The latter condition is the conventional setting in transfer learning [49]. Hypothetically the value of transfer learning should increase with smaller transfer data sets. Moreover, for each scenario we will investigate the first case where we will fine-tune all the layers with the transferred features. In the second case we will transfer the features but freeze the network weights in the first layers.
The first step in our approach is to split the data set into a *source* and *target* halve (see section 5.1. Once we have the target task, we can sample the smaller target splits from it. We train our CNN model on the source split first to obtain the features. Next, we transfer the features from all the layers of the trained source model, to the corresponding layers of the task model. In this condition we *retrain* the weights in all layers during training time. Since we have six target tasks of variable size, we repeat this six times (section 5.2).
In the second and main experiment, we extract the learned features per layer from the source model, and transfer them to the corresponding layer of the model which will be trained on one of the smaller target data sets (see 4.1)Moreover, these transferred features will not be updated during the backpropagation process. Our model CNN model consists of eight layers (see section 5.1.1), thus we repeat this seven times for each target task. Moreover, we repeat the experiments however, now we use the source split as our target split and vice versa Finally, we conduct the experiments described on a second data set as a means to better generalize our results. We will implement the model using the popular Caffe library (section 5.3).
We hypothesize that a transfer learning approach by fixing the first layers is more valuable if the target set is smaller, and that for larger data sets updating all layers will give better results, and validate this on data sets from two different image classification domains.

4.2 Related work

Several studies have investigated the generalizability of features and have proven the success of transfer learning [16,48,61]. A popular strategy for transfer learning is fine-tuning, by training a linear classifier on top of the final layer of a CNN. Zeiler *et al.* [77] examined this by pre-training

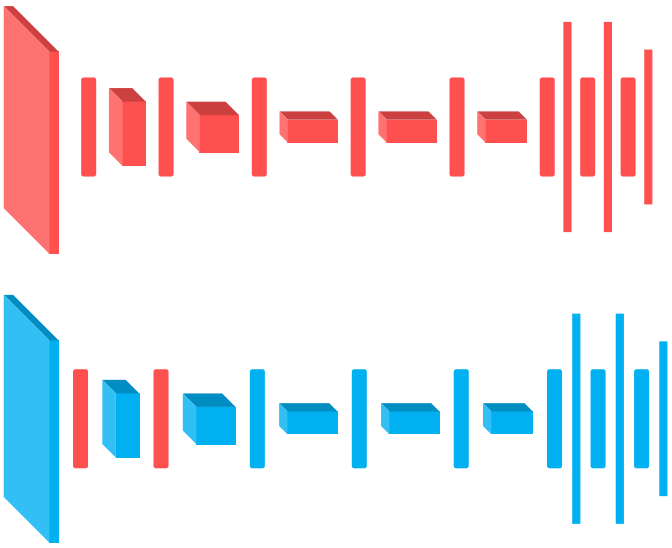


Figure 4.1: We transfer the features from a source task (red) to a target task (blue). The vertical bars indicate the weight vectors. The cubes represent the activations. In this figure we have transferred and locked the first two layers of features from the source to the target, $S2T$. We let the other six parameter layers of the target task initialize randomly. In the condition $FTall$ we transfer the parameters from the first seven layers to the target network.

a CNN on ImageNet, and then training a linear classifier on three target data sets, PASCAL VOC 2012, Caltech-101 [14] and Caltech-256 [24]. They varied the target data set size, as well as the layer from which the classifier is trained on. They found that the model generalizes extremely well to Caltech-101 and Caltech-256, however less so to PASCAL. Nonetheless, the study proved the benefits of applying transfer learning. Similarly, good results were yielded in [55] using this approach of transfer learning. The authors pre-trained on ImageNet in combination with a SVM classifier, and use Pascal VOC and MIT-67 Indoor Scenes as target tasks.
In [12] the researchers investigated how well features transfer to different domain target problems, and they investigated at what layer in the network this is most optimal. They first trained AlexNet on the ImageNet data set, and tested these features on a basic object recognition task using the Caltech-101 [14] data set. Second, they tested the network on domain adaptation, where there is a small amount of data is available, using the Office database [59]. Thirdly, they tested how well their model performs on a more fine grained data set, using the Caltech-UCSD birds data set [70]. Since the images in this data set are very similar to each other, this is a rather difficult image classification task. Finally, the authors tested their model on the SUN-397 Large-Scale Scene Recognition database. This task is quite different from the source task, where the task was to classify objects. The objective of the SUN-397 data set is to classify scenic categories. In every experiment the authors improved the benchmark scores, indicating that the features learned from ImageNet provide substantial generalisable properties.
In [1] the authors develop a relatively easy way to create new related tasks on which a CNN can train using the Caltech 101 data set. The features learned from this are then used for the target task. The CNN architecture consisted of two convolutional layers, each followed by a max pooling layer. The pseudo-tasks consists of taking a small random patch from an image, and generating a filter to perform a convolution on all training examples. The study found that transfer learning improved performance.

The authors in [9] use a CNN to learn Chinese characters from Latin and the other way around. The researchers first apply transfer learning for Latin characters only, where they pre-train the CNN on digits. They transfer the learned parameters to a new task where the objective is to classify uppercase letters. The authors vary the size of the target task (10, 50, 100, 500, 1000) and find that as the data set grows the effect of transfer learning decreases, however it still perform better than starting from scratch. Next, the main experiment is conducted, where the source and target task are from different data sets namely Chinese characters and Latin characters. The results were that transfer learning again gives a better performance. Moreover, in these experiments the transferred parameters were frozen at each layer in the CNN. Thus, the authors have proven the reusability of parameters.

Furthermore, in [62] the authors apply transfer learning to medical image data (thoraco-abdominal lymph node and interstitial lung disease). To this end they transferred parameters, which were pre-trained on ImageNet, to all CNN layers except the last one. These parameters were then further fine-tuned to the new task. They found that transfer learning resulted in a better performance over random initialization of the parameters. This shows that transfer learning is helpful across disparate data sets.

Moreover, [7] notes that overfitting to a particular class may occur during transfer learning. In [17] the authors investigate how representations can transfer across 22 different domains in sentiment classification for product reviews. This kind of problem is referred to by the authors as domain adaptation. Stacked denoising auto-encoders (SDAs) were used to perform unsupervised feature extraction on the domains that had no labels. The learned features were then used as feature vectors to the new task, rather than parameters. Similarly, [79] attempts to increase transferability of features by training deep autoencoders, with two encoding layers.

In [41] the researcher attempt to reduce the task specificity of features which occurs in deeper layers of an ANN when doing transfer learning. To this end, they develop a Deep Adaptation Network (DAN) architecture which increases the transferability of task specific features. DAN is an extension of AlexNet, where the first three layers are frozen, and layer four and five are fine-tuned. On the final three fully connected layers, a multiple kernel maximum mean discrepancies (MK-MMD) is added which make sure the source and target distribution of the data become similar in these layers. The experiments were done on the Office-31 [59], Office-10 and Caltech-10 [18] data sets. DAN was compared to five other models, and out-performed them in most of the transfer tasks.

Our research is a direct extension of the work by Yosinski *et al.* [74]. They investigated how transferable features are between layers in the AlexNet architecture. To this end they trained two networks, N_1 and N_2, each on a random split of the ImageNet data set containing half of the data, split A and split B. After both networks were trained on their respective splits, the features of the first layer from network N_1, the base, were *transferred* to the first layer of network N_2, the target. The remaining layers in network N_2 were randomly initialised. Finally, network N_2 gets trained on the B partition of the ImageNet data set. Thus, what happens is that network N_2 does not train from scratch, but rather, it uses the pre-initialised features from network N_1. The researchers do this for layer one to seven in the network, transferring both from A to B as well as from B to A. They found that the features in the first three layers are fairly general and could be transferred and boost performance. However, features in deeper layers of the network are more specific to the source task and therefore, transferring them worsens the performance.

Chapter 5

Experiments

In this chapter we describe the experimental setup in detail. We will discuss the steps we took to pre-process the data and describe the model used. Moreover, we will explain the features transfer process and our implementation of the model.

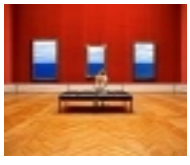
5.1 Data pre-processing

In our experiments, we use a subset of the ImageNet data set [11], **Tiny-ImageNet**. This data set contains 100,000 images with 200 classes, where each class contains 500 images, each of size 64 x 64 pixels (see appendix A). The data set contains images of a wide range of objects such as cats, parking meters, cliffs and rugby balls. The validation set contains 10,000 separate images. Moreover, we extend the work by Yosinski *et al.* [74] by also repeating the experiments on a second data set, **MiniPlaces2**. This is a scaled down version of the larger MIT Places database [78]. The data set is made up of images with settings such as a food court, golf course, an office, and ice skating rink. It contains 100,000 images with 100 classes (see appendix B). Each class consists of 1000 pictures of size 128 x 128 pixels, however we resize them to 64 x 64 pixels to keep the image size consistent with Tiny-ImageNet. Again, the validation set contains 10,000 images. In figure 5.1 we present several image classes of both data sets to underline the difference between the two domains. For example, with regards to the top row (Tiny-ImageNet), the network will use features learned from class *lighthouse*, and use them to predict class *umbrella*. Thus, we will

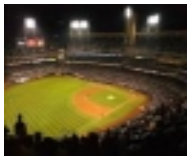
(a) Lighthouse

(b) Sulphur butterfly

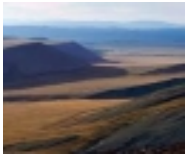
(c) Umbrella



(d) Museum



(e) Baseball field



(f) Valley

Figure 5.1: *Top:* A sample of training images from the Tiny-ImageNet data set. *Bottom:* A sample of training images from the MiniPlaces2 data set.

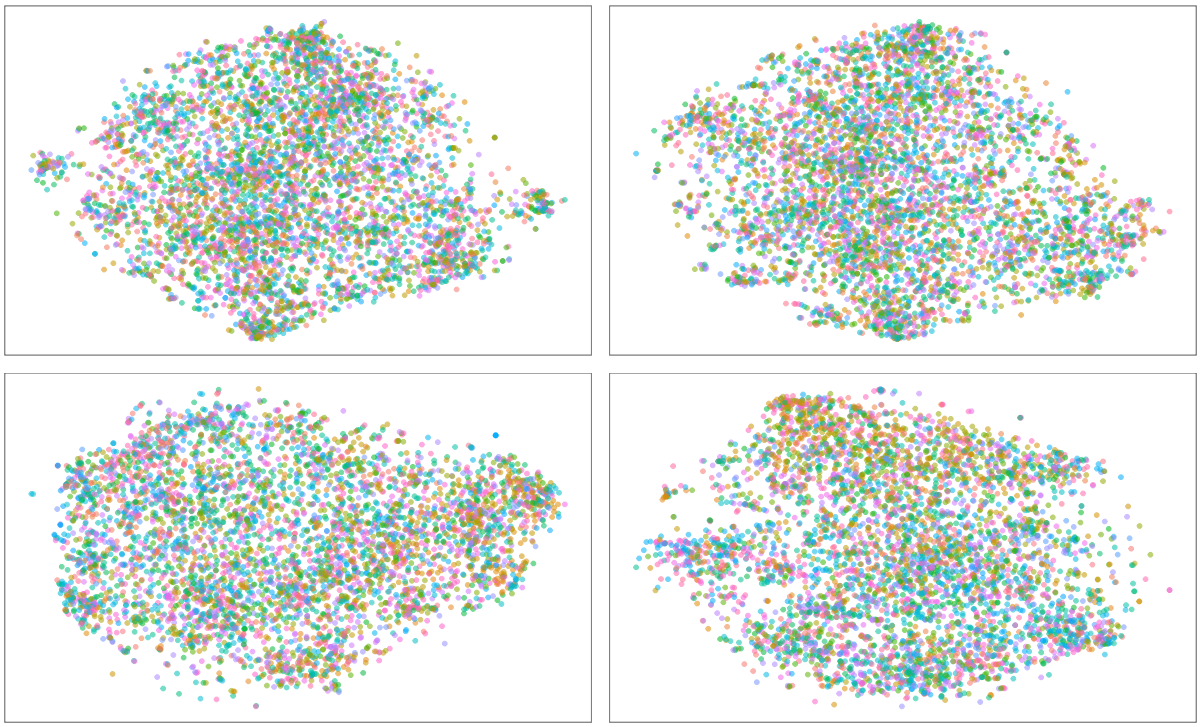


Figure 5.2: Visualisation of the separate validation set splits obtained by the t-SNE algorithm [43]. The first row displays the source and target split of Tiny-ImageNet respectively. Likewise, the bottom row shows the source and target split of MiniPlaces2.

investigate how well the network is able to transfer learned from one class to another within a data set.
To measure the effect of data set size on the generalizability of features, we transfer the features from a source task to a target task, where the latter has a variable size. We will test this on a subset of the ImageNet and Places2 data set. We use a subset of the data sets rather than training on the full data sets of ImageNet and Places2 (respectively containing 1.2 million and 8.1 million images for training) due to computational limitations. We denote our target data set as N_{target}. Moreover, we define the data set splits with a variable size as M_{target_i} where $M_{target_i} \subseteq N_{target}$. To obtain M_{target_i} from N_{target} we execute the following procedure:

1) We randomly split the entire data set into a source and a target partition, N_{source} and N_{target} respectively, where each partition contains 50,000 images. In both the source and target partition the images are equally distributed over $k = 100$ classes with 500 images per class for Tiny-ImageNet. In MiniPlaces2 the split is $k = 50$ classes per partition, with 1,000 images per class.

2) We artificially reduce N_{target} by drawing random samples of size M_{target_i} from each class k, where i equals 500[1], 400, 300, 200, 100 and 50 in case of Tiny-ImageNet. For MiniPlaces2 i equals 1000[1], 900, 800, 700, 600 and 500.

Moreover, for both Tiny-ImageNet and MiniPlaces2, we split the respective validation sets in half to create V_{target} and V_{source}, each containing 5,000 test images. The classes in V_{target} correspond to the classes in N_{target}. Therefore, V_{target} will be the validation set for all M_{target_i}

[1]Note that in the case where $i = 500$ and $i = 1{,}000$ we do not reduce N_{target} for Tiny-ImageNet and MiniPlaces2 respectively.

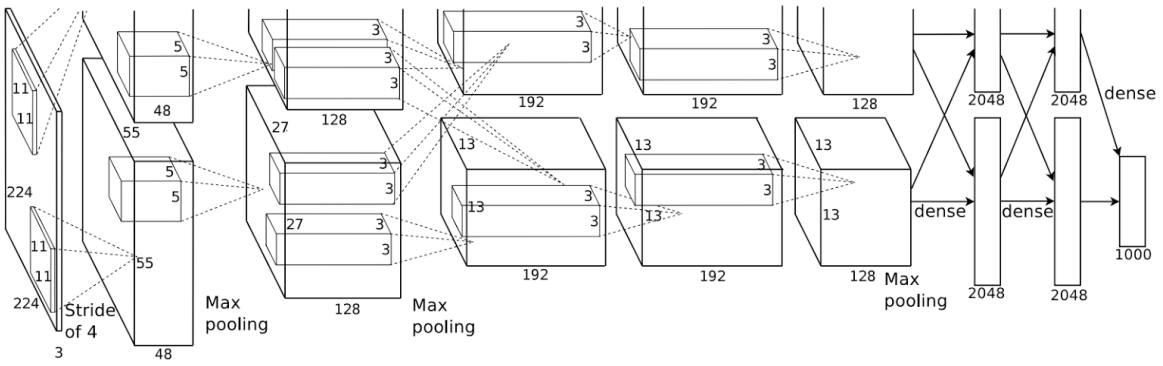


Figure 5.3: AlexNet architecture (illustration taken with permission from [38]).

sizes in our experiments. The other validation half, V_{source}, contains classes corresponding to N_{source}. In sum, we train our model on M_{target_i}, and evaluate it on a separate validation set V_{target}, to obtain our accuracy a.

5.1.1 CNN model

The CNN architecture we will use is AlexNet, developed by Krizhevsky *et al.* [38] (see figure 5.3), which was the winning model in the ImageNet Large Scale Visual Recognition Challenge 2012.
The model consists of five convolutional layers and three fully connected layers. The first two convolutional layers are followed by a max pooling layer and a normalization layer respectively. The fifth convolutional layer is followed only by a max pooling layer. The first two fully connected layers contain 4,096 neurons. The final fully connected layer contains 1000 neurons for the target class scores. It is interesting to note that the authors used Rectified Linear Units (ReLUs) as activation functions instead of the regular sigmoid. Moreover, they applied a regularization technique called dropout to reduce overfitting [64].

5.2 Transferring features

To create a model from which we can transfer the features, we first train our network on N_{source}. The parameters of the source model are stored in a Caffemodel object (see section 5.3), which we use to transfer the parameters from the source model to the target model.
To obtain our baseline score we do not apply any transfer learning at all, and let the model train on the given training set. In our first experiment we fine-tune the network by transferring all the features from the source task to the model, and continue with backpropagation on the new task.
However, since we are also interested in at what layer l of the network features are able to generalize, we transfer the features from the source to the target task, one layer at a time. AlexNet has eight layers in total. Therefore, we transfer from layer $l = 1$, up until layer $l = 7$. When we transfer the parameters to the target model, we keep them fixed. That is to say, we do not update the parameters by gradient descent. The remaining 8 - l layers of the network we randomly initialize and let the errors backpropagate through the layers.
Finally, to get a mean accuracy score, we run the experiments again by following the same procedure, but now use N_{source} as N_{target} and vice versa.

5.3 Training

To conduct our experiments, we use the Caffe deep learning framework developed at UC Berkeley [34]. We make use of a single Nvidida GTX Titan X graphics card to enable Caffe in GPU mode, to speed up our training time. We use the AlexNet reference model which is included in Caffe. Detailed information about the model architecture can be found in [38]. Moreover, we follow the same training regime as specified by the reference model.
Furthermore, in terms of data augmentation we take a random crop in the training phase and use random mirroring as specified by Caffe. In the test phase we take a center crop of the images. Since our input images are 64 x 64, we change the crop size to 57, rather than upscaling the images to 256 x 256 and applying the default crop size of 227. Thus, we stay consistent with the ratio used in the AlexNet reference model. Moreover, we subtract the image mean from each image.
Finally, to determine for how many iterations we should train the models, we trained on N_{source} of both data sets and validated on the respective V_{source}, without applying any form of transfer learning.
We found that the model began to overfit on the training data around 10,000 iterations (see figure 5.4 and 5.5). Therefore, we found it reasonable for subsequent experiments to let each model run for 10K iterations in order to measure the positive effect of transfer learning. Moreover, the more we reduce N_{target}, the faster the model will reach the point of overfitting, which is evidenced by the decreasing accuracy of the *base* training conditions across our experiments.

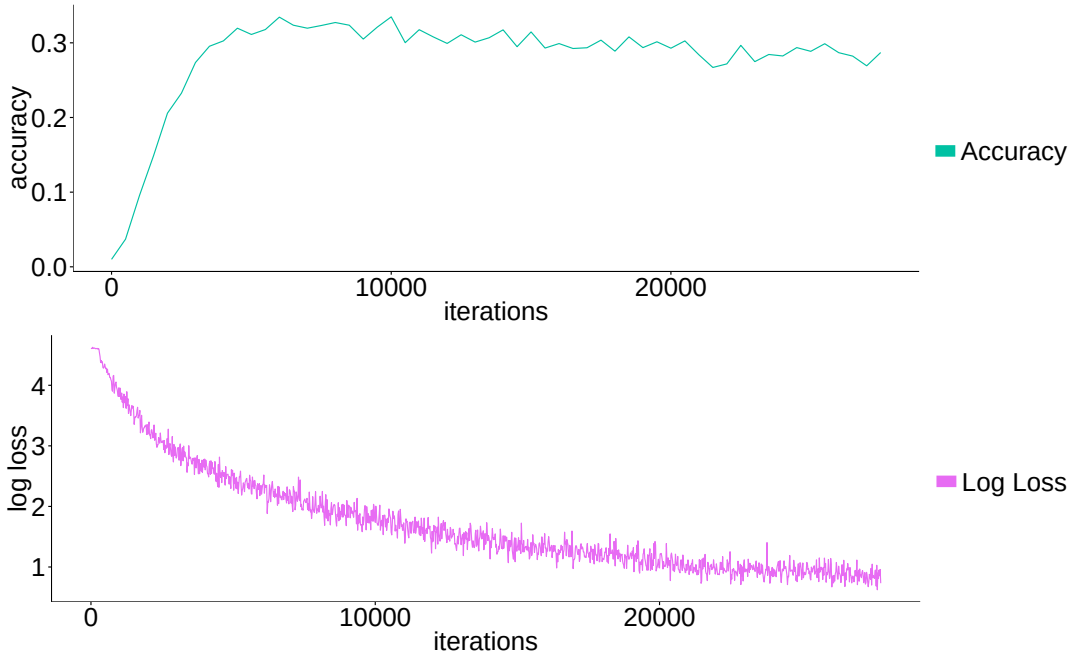


Figure 5.4: *Top:* The accuracy on V_{source} after training on N_{source} of the Tiny-ImageNet data set after 25K iterations. This split contains 100 classes, with 500 images per class. *Bottom:* The log loss over the training set with the identical split.

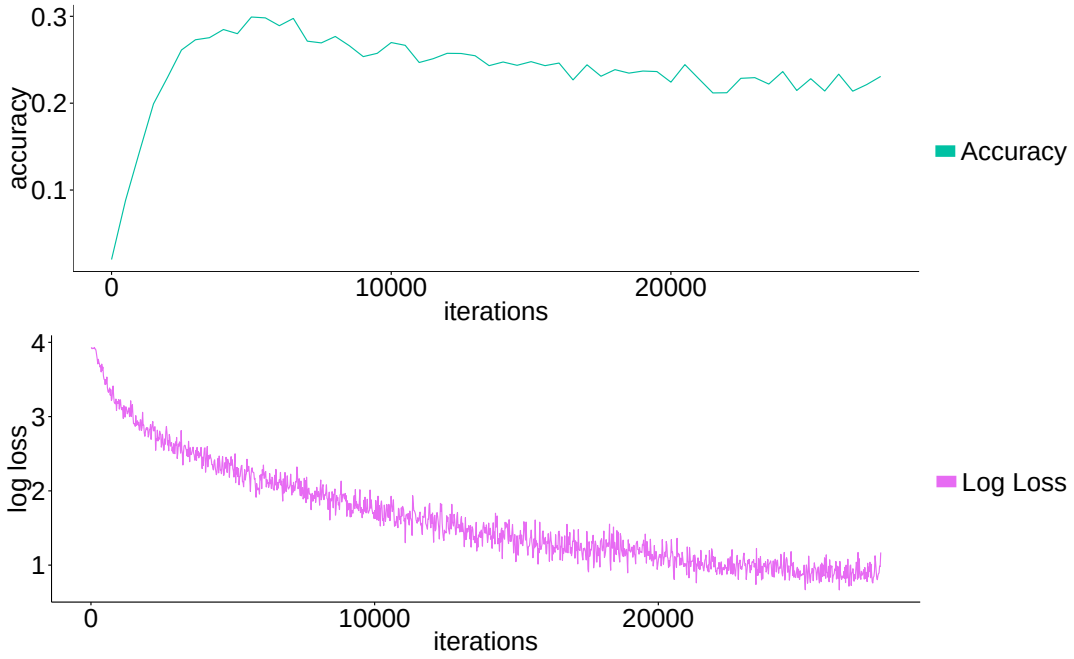


Figure 5.5: *Top:* The accuracy on V_{source} after training on N_{source} of the MiniPlaces2 data set after 25K iterations. This split contains 50 classes, with 1000 images per class. *Bottom:* The log loss over the training set with the identical split.

Chapter 6

Results

In this chapter we present the results from our experiments.

6.1 Results Tiny-ImageNet

In figure 6.1 we see the results of transfer learning on different data set sizes. The plot shows the accuracy on the validation set after 10K iterations of training. The first two conditions are the base case (*base*) and fine-tune all ($FTall$). The condition *base* indicates we did not apply transfer learning. Condition $FTall$ means we fine-tuned through all the layers, and the notation SnT denotes up until which layers we freeze the transferred features from the source in the target model. For instance, $S3T$ implies we transferred the first three feature layers from the model trained on N_{source} to the model trained on M_{target_i}. The final seven scores are the accuracies where we transfer the parameters per layer from the source, and freeze that particular layer.
We notice an effect of data set size on the accuracy of the baseline score. As we decrease the data set size, we find that the accuracy decreases as well. In figure 6.1 we observe that the accuracy worsens as we keep more layers fixed when transferring parameters from the source task. Moreover, we observer a peculiar spike at layer $l = 2$ for each data set size. Furthermore, there is a sever drop in accuracy at layer $l = 4$. Though after this layer, accuracy seems to increase again at $l = 5$. However, the deeper we get into the network, the worse the performance becomes. Furthermore, as we decrease the data set size, the difference between the base condition and FTall increases.

6.2 Results MiniPlaces2

As can be seen from figure 6.2, even though this is a task from a different domain, the results follow a pattern very similar to Tiny-ImageNet. We notice the same spike at layer $l = 2$ and drop at layer $l = 4$ as we did on the Tiny-ImageNet data set. With smaller target set sizes the benefits of locking the first few layers increases. We find that in higher layers of the model the accuracy drops. Only for $M_{target_{1000}}$ the graphs seem to indicate that training from scratch is better, but this is truly just a baseline. Moreover, as we decrease the data set size the condition FTall only has a better performance for $M_{target_{500}}$. For the rest of the data set sizes the base condition has a higher accuracy than FTall. In a real deployment one would probably expect that the source classes also still need to be recognized, and performance of tuning all layers is still lower than locking some of the initial layers.

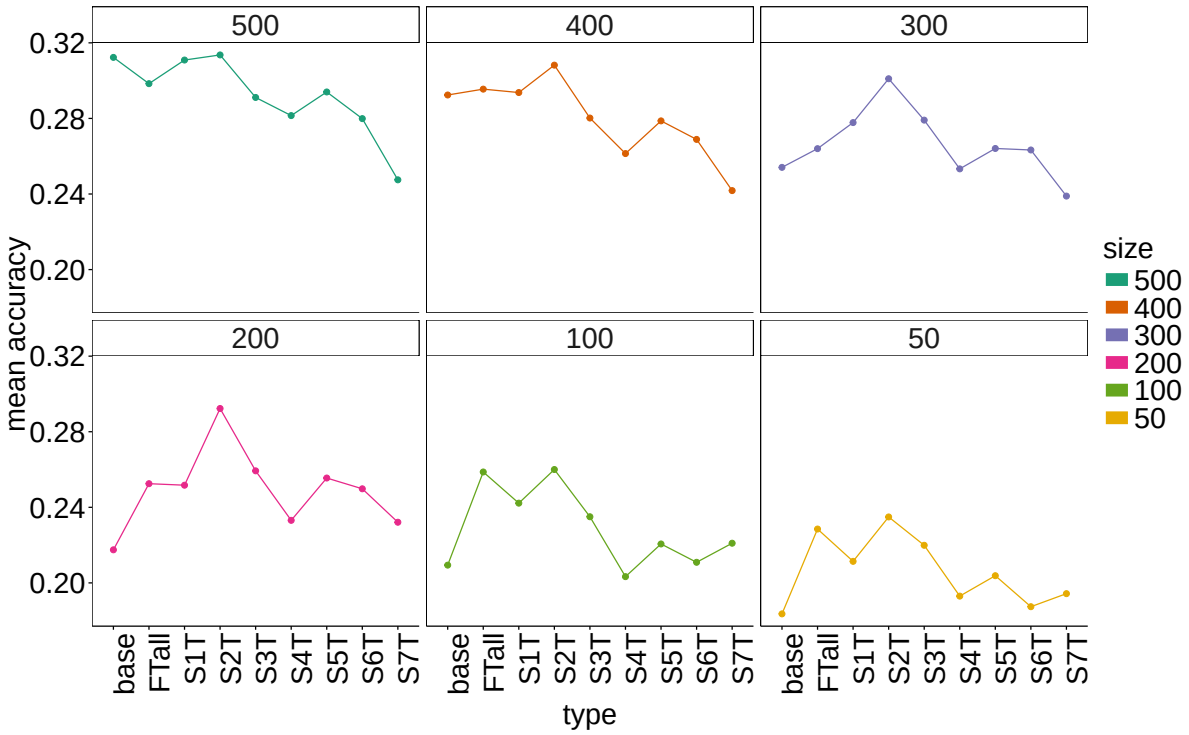


Figure 6.1: Mean accuracy obtained after training on the target splits of Tiny-ImageNet where i in M_{target_i} equals 500, 400, 300, 200, 100 and 50 and validating on V_{target}. Note that we ran the same experiments again, but used N_{source} as N_{target} and vice versa. Thus, we obtained our mean accuracies by averaging the scores.

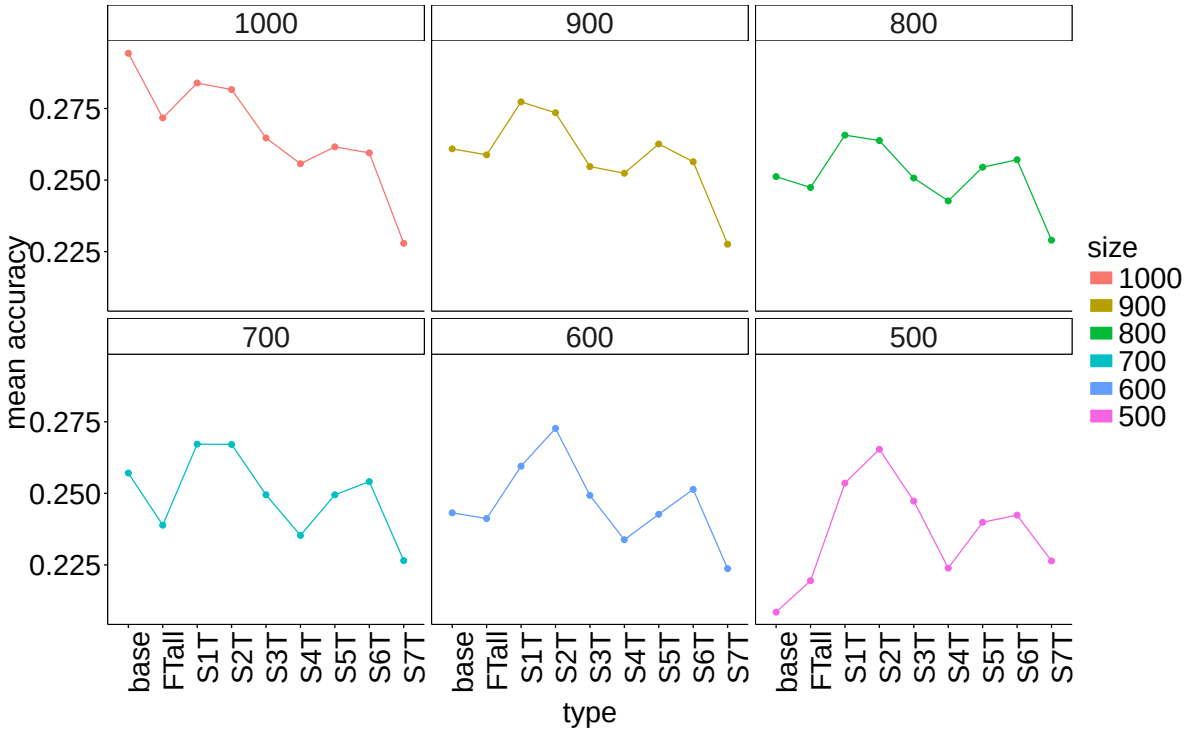


Figure 6.2: Mean accuracy after training on the target splits of MiniPlaces2, where i in M_{target_i} equals 1000, 900, 800, 700, 600 and 500 and validating on V_{target}.

Chapter 7

Discussion

In this chapter we interpret our results, and conclude with several suggestions for future research directions.

Our results reveal that data set size affects the accuracy in transfer learning with deep convolutional neural networks. The first effect we notice is on the baseline case (to repeat, just training the network with randomly initialized weights). We can see that the model starts to overfit on the training data when we artificially reduce the data set size, which leads to a steady decline in accuracy on both Tiny-ImageNet as well as MiniPlaces2. This can be explained by a sub-optimal parameter configuration as a result of overfitting on a small data set size.

Furthermore, fine-tuning all the layers only appears to have a positive effect with smaller data sets for Tiny-ImageNet where i in M_{target_i} ranges from 400 until 50, and MiniPlaces2 where M_{target_i} equals 500. This is an interesting result, as a network for which all layers can be adapted still benefits from potentially valuable initialization of the weights. We speculate that the source features are important for the target data set splits as well. Thus, the effect of initializing the model with parameters obtained from a model trained on a larger data set clearly shows its advantage. Moreover, we notice a visible spike in accuracy in all our graphs, when we transfer parameters from the first two layers. This layer seems to consist of important features which determine the network's performance. Likewise, there is a considerable decline in accuracy when transferring four layers, compared to transferring the first three layers.

The results in figure 6.1 generally follow the findings of the study by Yosinski *et al.* [74]. As we transfer more and more features (layers) from the source task, the accuracy initially goes up but then decreases. This can be attributed to feature specificity with regards to the source task. However, we observe a second positive spike in the accuracy at layer $l = 5$ in nearly all of our experiments. This result is quite surprising since the features have become substantially specific to the source, and yet generalize well to the new task. Evidently, the transferred features from the source task in this layer hold the same, or even superior, representational power compared to the features solely learned from a target data set. Furthermore, our results for Tiny-ImageNet with regards to condition base and FTall, seem to be consistent with [9] where the authors observed that with fewer samples per class, the effect of transfer learning has more effect. However, we do not find this effect for MiniPlaces2. This indicates that the transferred weights were initialized, such that further training let them converge to sub-optimal local minima.

All these results can be summarized into a fairly straightforward heuristic. For the first n instances of a new class, freeze the first l layers of the network. Once you have obtained more than n instances for new class, training can simply affect all layers. Obviously the values for n and l depend on the data and task at hand, in our experiments freezing the first 3 layers until 300 (Tiny-ImageNet) and respectively 900 (MiniPlaces2) instances per class gave the best results.

Our study could have benefited from having more samples per data point, by running repeated experiments. Since the initialization of the parameters happens at random, the parameters might

converge at different local minima each time the model is run. This could effect the accuracy score in the test phase. Our results still indicate that transferring features from a larger source data set to a smaller target data set adds value by reducing the risk of overfitting, and improves performance.

7.1 Future work

Future research directions might include more investigation into developing metrics for quantifying the transferability of parameters in CNNs. Another direction might be to research whether features in other deep ANN are transferable. There has been work done to investigate transfer learning in reinforcement learning models. In [51], the researchers transfer features learned from one Atari Arcade game to a different Atari game. It would be interesting to see to what extent transfer learning can be applied to RNNs, similar to [68]. Moreover, one might find a difference in transferability of features depending on the model or domain, such as image captioning and video analysis. One more direction could involve, applying the pre-trained networks and measuring their performance on the original source task.
A final direction might be to research which filters are best suited to be transferred in each layer. In our experiments we simply take all features per layer of the network and transfer them. However, it may be worthwhile to select m candidate filters from the K filters in a layer of a CNN. Only these m get transferred to the new network. In [38] there are $K = 96$ filters in the first layer. A strategy could be to randomly, or via a cluster algorithm, take $m = 72$ filters from the source model, and transfer those to the target network. Thus, the target network will keep 96 - 72 = 24 filters of its own which can be randomly initialized. Selecting only m filters from the source might help improve generalizability. This will hold especially true in the deeper layers of the network, where features become more task specific. This strategy might prevent co-adaptation of neurons in these deeper layers.

Chapter 8

Conclusion

In this thesis we reviewed feedforward neural networks, convolutional neural networks and recurrent neural networks. In the main experiment we investigated the effect of data set size on the generalizability of features in deep convolutional neural networks. To this end, we transferred features from a pre-trained network to a new network. We systematically reduced the size of the target training set and trained our new network on these splits with the pre-initialized features. In support for a general rule of thumb heuristic, we found empirical evidence that freezing the first two to three layers of features results in a significant performance boost over the baseline score, especially for smaller target set sizes under a thousand instances per class. Finally, we provided potential future research directions related to transfer learning in deep neural networks.

Bibliography

[1] Ahmed, A., Yu, K., Xu, W., Gong, Y., Xing, E.: Training hierarchical feed-forward visual recognition models using transfer learning from pseudo-tasks. In: European Conference on Computer Vision. pp. 69–82. Springer (2008)

[2] Auli, M., Galley, M., Quirk, C., Zweig, G.: Joint language and translation modeling with recurrent neural networks. In: EMNLP. vol. 3, p. 0 (2013)

[3] Ba, J., Frey, B.: Adaptive dropout for training deep neural networks. In: Advances in Neural Information Processing Systems. pp. 3084–3092 (2013)

[4] Bay, H., Tuytelaars, T., Van Gool, L.: Surf: Speeded up robust features. In: European conference on computer vision. pp. 404–417. Springer (2006)

[5] Bengio, I.G.Y., Courville, A.: Deep learning (2016), `http://www.deeplearningbook.org`, book in preparation for MIT Press

[6] Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks 5(2), 157–166 (1994)

[7] Bengio, Y., et al.: Deep learning of representations for unsupervised and transfer learning. ICML Unsupervised and Transfer Learning 27, 17–36 (2012)

[8] Cireşan, D.C., Giusti, A., Gambardella, L.M., Schmidhuber, J.: Mitosis detection in breast cancer histology images with deep neural networks. In: International Conference on Medical Image Computing and Computer-assisted Intervention. pp. 411–418. Springer (2013)

[9] Cireşan, D.C., Meier, U., Schmidhuber, J.: Transfer learning for Latin and Chinese characters with deep neural networks. In: The 2012 International Joint Conference on Neural Networks (IJCNN). pp. 1–6. IEEE (2012)

[10] Dalal, N., Triggs, B.: Histograms of oriented gradients for human detection. In: 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05). vol. 1, pp. 886–893. IEEE (2005)

[11] Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. pp. 248–255. IEEE (2009)

[12] Donahue, J., Jia, Y., Vinyals, O., Hoffman, J., Zhang, N., Tzeng, E., Darrell, T.: Decaf: A deep convolutional activation feature for generic visual recognition. arXiv preprint arXiv:1310.1531 (2013)

[13] Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research 12(Jul), 2121–2159 (2011)

[14] Fei-Fei, L., Fergus, R., Perona, P.: Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. Computer Vision and Image Understanding 106(1), 59–70 (2007)

[15] Fukushima, K.: Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological cybernetics 36(4), 193–202 (1980)

[16] Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 580–587 (2014)

[17] Glorot, X., Bordes, A., Bengio, Y.: Domain adaptation for large-scale sentiment classification: A deep learning approach. In: Proceedings of the 28th International Conference on Machine Learning (ICML-11). pp. 513–520 (2011)

[18] Gong, B., Grauman, K., Sha, F.: Connecting the dots with landmarks: Discriminatively learning domain-invariant features for unsupervised domain adaptation. In: ICML (1). pp. 222–230 (2013)

[19] Goodfellow, I.J., Bulatov, Y., Ibarz, J., Arnoud, S., Shet, V.: Multi-digit number recognition from street view imagery using deep convolutional neural networks. arXiv preprint arXiv:1312.6082 (2013)

[20] Graves, A.: Neural networks. In: Supervised Sequence Labelling with Recurrent Neural Networks, pp. 15–35. Springer (2012)

[21] Graves, A., Mohamed, A.r., Hinton, G.: Speech recognition with deep recurrent neural networks. In: 2013 IEEE international conference on acoustics, speech and signal processing. pp. 6645–6649. IEEE (2013)

[22] Greff, K., Srivastava, R.K., Koutník, J., Steunebrink, B.R., Schmidhuber, J.: Lstm: A search space odyssey. arXiv preprint arXiv:1503.04069 (2015)

[23] Gregor, K., Danihelka, I., Graves, A., Rezende, D.J., Wierstra, D.: Draw: A recurrent neural network for image generation. arXiv preprint arXiv:1502.04623 (2015)

[24] Griffin, G., Holub, A., Perona, P.: Caltech-256 object category dataset (2007)

[25] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385 (2015)

[26] Hebb, D.O.: The organization of behavior: A neuropsychological theory. Psychology Press (2005)

[27] Hermans, M., Schrauwen, B.: Training and analysing deep recurrent neural networks. In: Advances in Neural Information Processing Systems. pp. 190–198 (2013)

[28] Hinton, G., Srivastava, N., Swersky, K.: Lecture 6a overview of mini–batch gradient descent

[29] Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.R.: Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580 (2012)

[30] Hochreiter, S.: Untersuchungen zu dynamischen neuronalen netzen. Diploma, Technische Universität München p. 91 (1991)

[31] Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation 9(8), 1735–1780 (1997)

[32] Hubel, D.H., Wiesel, T.N.: Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. The Journal of physiology 160(1), 106–154 (1962)

[33] Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167 (2015)

[34] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the ACM International Conference on Multimedia. pp. 675–678. ACM (2014)

[35] Karpathy, A., Fei-Fei, L.: Deep visual-semantic alignments for generating image descriptions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 3128–3137 (2015)

[36] Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., Fei-Fei, L.: Large-scale video classification with convolutional neural networks. In: Proceedings of the IEEE conference on Computer Vision and Pattern Recognition. pp. 1725–1732 (2014)

[37] Kingma, D., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)

[38] Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. pp. 1097–1105 (2012)

[39] LeCun, Y.: Une procedure d'apprentissage pour reseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks) (1985)

[40] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE 86(11), 2278–2324 (1998)

[41] Long, M., Wang, J.: Learning transferable features with deep adaptation networks. CoRR, abs/1502.02791 1, 2 (2015)

[42] Lowe, D.G.: Object recognition from local scale-invariant features. In: Computer vision, 1999. The proceedings of the seventh IEEE international conference on. vol. 2, pp. 1150–1157. Ieee (1999)

[43] Van der Maaten, L., Hinton, G.: Visualizing data using t-SNE. Journal of Machine Learning Research 9(2579-2605), 85 (2008)

[44] McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics 5(4), 115–133 (1943)

[45] Minsky, M., Papert, S.: Perceptrons. (1969)

[46] Nesterov, Y.: A method of solving a convex programming problem with convergence rate o (1/k2)

[47] Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Le, Q.V., Ng, A.Y.: On optimization methods for deep learning. In: Proceedings of the 28th International Conference on Machine Learning (ICML-11). pp. 265–272 (2011)

[48] Oquab, M., Bottou, L., Laptev, I., Sivic, J.: Learning and transferring mid-level image representations using convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 1717–1724 (2014)

[49] Pan, S.J., Yang, Q.: A survey on transfer learning. Knowledge and Data Engineering, IEEE Transactions on 22(10), 1345–1359 (2010)

[50] Papert, S.: The summer vision project (1966)

[51] Parisotto, E., Ba, J.L., Salakhutdinov, R.: Actor-mimic: Deep multitask and transfer reinforcement learning. arXiv preprint arXiv:1511.06342 (2015)

[52] Parker, D.B.: Learning logic (1985)

[53] Pascanu, R., Mikolov, T., Bengio, Y.: On the difficulty of training recurrent neural networks. ICML (3) 28, 1310–1318 (2013)

[54] Qian, N.: On the momentum term in gradient descent learning algorithms. Neural networks 12(1), 145–151 (1999)

[55] Razavian, A., Azizpour, H., Sullivan, J., Carlsson, S.: Cnn features off-the-shelf: an astounding baseline for recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. pp. 806–813 (2014)

[56] Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review 65(6), 386 (1958)

[57] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. Tech. rep., DTIC Document (1985)

[58] Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Cognitive modeling 5(3), 1 (1988)

[59] Saenko, K., Kulis, B., Fritz, M., Darrell, T.: Adapting visual category models to new domains. In: Computer Vision–ECCV 2010, pp. 213–226. Springer (2010)

[60] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., LeCun, Y.: Overfeat: Integrated recognition, localization and detection using convolutional networks. arXiv preprint arXiv:1312.6229 (2013)

[61] Sharif Razavian, A., Azizpour, H., Sullivan, J., Carlsson, S.: Cnn features off-the-shelf: an astounding baseline for recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. pp. 806–813 (2014)

[62] Shin, H.C., Roth, H.R., Gao, M., Lu, L., Xu, Z., Nogues, I., Yao, J., Mollura, D., Summers, R.M.: Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning. IEEE transactions on medical imaging 35(5), 1285–1298 (2016)

[63] Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)

[64] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research 15(1), 1929–1958 (2014)

[65] Svozil, D., Kvasnicka, V., Pospichal, J.: Introduction to multi-layer feed-forward neural networks. Chemometrics and intelligent laboratory systems 39(1), 43–62 (1997)

[66] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 1–9 (2015)

[67] Taigman, Y., Yang, M., Ranzato, M., Wolf, L.: Deepface: Closing the gap to human-level performance in face verification. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (June 2014)

[68] Tang, Z., Wang, D., Zhang, Z.: Recurrent neural network training with dark knowledge transfer. In: 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). pp. 5900–5904. IEEE (2016)

[69] Toshev, A., Szegedy, C.: Deeppose: Human pose estimation via deep neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 1653–1660 (2014)

[70] Welinder, P., Branson, S., Mita, T., Wah, C., Schroff, F., Belongie, S., Perona, P.: Caltech-ucsd birds 200 (2010)

[71] Werbos, P.: Beyond regression: New tools for prediction and analysis in the behavioral sciences (1974)

[72] Widrow, B., et al.: Adaptive" adaline" Neuron Using Chemical" memistors.". (1960)

[73] Williams, D., Hinton, G.: Learning representations by back-propagating errors. Nature 323, 533–536 (1986)

[74] Yosinski, J., Clune, J., Bengio, Y., Lipson, H.: How transferable are features in deep neural networks? In: Advances in Neural Information Processing Systems. pp. 3320–3328 (2014)

[75] Zaremba, W., Sutskever, I.: Learning to execute. arXiv preprint arXiv:1410.4615 (2014)

[76] Zeiler, M.D.: Adadelta: an adaptive learning rate method. arXiv preprint arXiv:1212.5701 (2012)

[77] Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks. In: Computer vision–ECCV 2014, pp. 818–833. Springer (2014)

[78] Zhou, B., Lapedriza, A., Xiao, J., Torralba, A., Oliva, A.: Learning deep features for scene recognition using places database. In: Advances in neural information processing systems. pp. 487–495 (2014)

[79] Zhuang, F., Cheng, X., Luo, P., Pan, S.J., He, Q.: Supervised representation learning: Transfer learning with deep autoencoders. In: Int. Joint Conf. Artif. Intell (2015)

Appendices

Appendix A

Tiny-ImageNet classes

Classes Tiny-ImageNet			
goldfish	bison	dumbbell	sombrero
European fire salamander	bighorn	flagpole	space heater
bullfrog	gazelle	fountain	spider web
tailed frog	Arabian camel	freight car	sports car
American alligator	orangutan	frying pan	steel arch bridge
boa constrictor	chimpanzee	fur coat	stopwatch
trilobite	baboon	gasmask	sunglasses
scorpion	African elephant	go-kart	suspension bridge
black widow	lesser panda	gondola	swimming trunks
tarantula	abacus	hourglass	syringe
centipede	academic gown	iPod	teapot
goose	altar	jinrikisha	teddy
koala	apron	kimono	thatch
jellyfish	backpack	lampshade	torch
brain coral	bannister	lawn mower	tractor
snail	barbershop	lifeboat	triumphal arch
slug	barn	limousine	trolleybus
sea slug	barrel	magnetic compass	turnstile
American lobster	basketball	maypole	umbrella
spiny lobster	bathtub	military uniform	vestment
black stork	beach wagon	miniskirt	viaduct
king penguin	beacon	moving van	volleyball
albatross	beaker	nail	water jug
dugong	beer bottle	neck brace	water tower
Chihuahua	bikini	obelisk	wok
Yorkshire terrier	binoculars	oboe	wooden spoon
golden retriever	birdhouse	organ	comic book
Labrador retriever	bow tie	parking meter	plate
German shepherd	brass	pay-phone	guacamole
standard poodle	broom	picket fence	ice cream
tabby	bucket	pill bottle	ice lolly
Persian cat	bullet train	plunger	pretzel
Egyptian cat	butcher shop	pole	mashed potato
cougar	candle	police van	cauliflower
lion	cannon	poncho	bell pepper

brown bear	cardigan	pop bottle	mushroom
ladybug	cash machine	potter's wheel	orange
fly	CD player	projectile	lemon
bee	chain	punching bag	banana
grasshopper	chest	reel	pomegranate
walking stick	Christmas stocking	refrigerator	meat loaf
cockroach	cliff dwelling	remote control	pizza
mantis	computer keyboard	rocking chair	potpie
dragonfly	confectionery	rugby ball	espresso
monarch	convertible	sandal	alp
sulphur butterfly	crane	school bus	cliff
sea cucumber	dam	scoreboard	coral reef
guinea pig	desk	sewing machine	lakeside
hog	dining table	snorkel	seashore
ox	drumstick	sock	acorn

Table A.1: 200 classes sampled from the ImageNet data set.

Appendix B

MiniPlaces2 classes

Classes MiniPlaces2			
abbey	bus_interior	gas_station	racecourse
airport_terminal	butchers_shop	golf_course	railroad_track
amphitheater	campsite	harbor	rainforest
amusement_park	candy_store	highway	restaurant
aquarium	canyon	hospital_room	river
aqueduct	cemetery	hot_spring	rock_arch
art_gallery	chalet	ice_skating_rink_outdoor	runway
assembly_line	church_outdoor	iceberg	shed
auditorium	classroom	kindergarden_classroom	shower
badlands	clothing_store	kitchen	ski_slope
bakery_shop	coast	laundromat	skyscraper
ballroom	cockpit	lighthouse	slum
bamboo_forest	coffee_shop	living_room	stadium_football
banquet_hall	conference_room	lobby	stage_indoor
bar	construction_site	locker_room	staircase
baseball_field	corn_field	market_outdoor	subway_station_platform
bathroom	corridor	martial_arts_gym	supermarket
beauty_salon	courtyard	monastery_outdoor	swamp
bedroom	dam	mountain	swimming_pool_outdoor
boat_deck	desert_sand	museum_indoor	temple_east_asia
bookstore	dining_room	office	track_outdoor
botanical_garden	driveway	palace	trench
bowling_alley	fire_station	parking_lot	valley
boxing_ring	food_court	phone_booth	volcano
bridge	fountain	playground	yard

Table B.1: 100 classes sampled from the Places2 data set.

Appendix C

Backpropagation derivation

In this section we provide the derivation of the backpropagation algorithm for a single training example (x^i, y^i), where $\frac{\partial}{\partial \theta_{ij}^l} J(\theta)$ represents the partial derivative of the cost function $J(\theta)$ with respect to weight θ_{ij}^l. Similarly, equation C.2 shows the calculation of the gradient for the bias term b_i^l.

$$\frac{\partial}{\partial \theta_{ij}^l} J(\theta) = \frac{1}{n} \sum_{k=1}^{n} \frac{\partial}{\partial W_{ij}^l} J(\theta; x^i, y^i) \tag{C.1}$$

$$\frac{\partial}{\partial b_i^l} J\theta) = \frac{1}{n} \sum_{k=1}^{n} \frac{\partial}{\partial b_i^l} J(\theta; x^i, y^i) \tag{C.2}$$

We can rewrite the expressions C.1 and C.2 as follows:

$$\frac{\partial}{\partial \theta_{ij}^l} J(\theta) = \frac{\partial J(\theta)}{\partial z_i^{l+1}} \frac{\partial z_i^{l+1}}{\partial \theta_{ij}^l} \tag{C.3}$$

$$\frac{\partial}{\partial b_i^l} J(\theta) = \frac{\partial J(\theta)}{\partial z_i^{l+1}} \tag{C.4}$$

The second term in C.3 can simply be equated to the activation of node i in layer l:

$$\frac{\partial z_i^{l+1}}{\partial \theta_{ij}^l} = a_i^l \tag{C.5}$$

The key of the backpropagation algorithm is the computation of the "error" or delta term δ_i^l. This can be equated to the first term in equation C.3:

$$\delta_i^l = \frac{\partial J(\theta)}{\partial z_i^l} \tag{C.6}$$

To compute δ_i^l we must we apply the chain rule. We first decompose δ_i^l as follows:

$$\delta_i^l = \frac{\partial J(\theta)}{\partial a_i^l} \frac{\partial a_i^l}{\partial z_i^l} = \frac{\partial J(\theta)}{\partial a_i^l} f'(z_i^l) \tag{C.7}$$

The computation for the delta term is different for the nodes in the output layer than for the nodes in the hidden layers. With respect to the output layer $l = n_l$, we compute $\delta_i^{o_l}$ for any output node i as follows:

$$\delta_i^{o_l} = \frac{\partial J(\theta)}{\partial a_i^{o_l}} \frac{\partial a_i^l}{\partial z_i^l} = (y_i - a_i^{o_l}) f'(z_i^{o_l}) \tag{C.8}$$

For the hidden nodes we first compute a weighted average of the error terms that use a_i^l (these are the nodes in layer $l = l + 1$). Thus we set the first term in equation C.7 to:

$$\frac{\partial J(\theta)}{\partial a_i^l} = \sum_{j=1}^{s_{l+1}} \frac{\partial J(\theta)}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial a_i^l} = \sum_{j=1}^{s_{l+1}} \delta_j^{l+1} W_{ji}^l. \tag{C.9}$$

This gives the following computation for δ_i^l:

$$\delta_i^l = \sum_{j=1}^{s_{l+1}} \delta_j^{l+1} W_{ji}^l f'(z_i^l) \tag{C.10}$$

The equations to compute the gradient then become:

$$\frac{\partial}{\partial \theta_{ij}^l} J(\theta) = \delta_i^l a_i^l \tag{C.11}$$

$$\frac{\partial}{\partial b_i^l} J(\theta) = \delta_i^l \tag{C.12}$$

Appendix D

Vanishing and exploding gradients

In this section we will explain the vanishing and exploding gradient problem which occurs in RNNs as described in [53].

The partial derivative of the cost function E with respect to parameters θ over S time steps is expressed by:

$$\frac{\partial E}{\partial \theta} = \sum_{t=1}^{S} \frac{\partial E_t}{\partial \theta} \tag{D.1}$$

The computation of the gradient for the cost term E_t with respect to θ can be decomposed as follows:

$$\frac{\partial E_t}{\partial \theta} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial \theta} \tag{D.2}$$

The term $\frac{\partial h_t}{\partial h_k}$ is in itself a chain rule, concretely it is a products of Jacobian terms:

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} \theta^T \text{diag}[\phi'(h_{i-1}] \tag{D.3}$$

The upper bound of the derivative of the activation function is denoted by γ_ϕ, and γ_θ denotes the l^2-norm of $\left\|\theta^T\right\|$ which corresponds to the largest eigenvalue.

$$\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\| \leq \left\|\theta^T\right\| \left\|\text{diag}[\phi'((h_{i-1})]\right\| \leq \gamma_\theta \gamma_\phi \tag{D.4}$$

The norm of the Jacobian matrix $\left\| \frac{\partial h_t}{\partial h_k} \right\|$, is denoted by $(\gamma_\theta \gamma_\phi)^{t-k}$. However, if this term is smaller than one, it goes to zero due to the exponent. Likewise, if it larger than one, the gradient will blow up.

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| \leq (\gamma_\theta \gamma_\phi)^{t-k} \tag{D.5}$$

Appendix E

Backpropagation in CNNs

The computation for the delta term in a fully connected layer l is identical to the computation in feedforward networks (see C.10, C.11, C.12).

In the convolutional and pooling layer we have to upsample the delta terms δ_k^l in every k^{th} filter. Similar to C.10 the term $f'(z_k^l)$ denotes the derivative of the activation function (equation E.1 shows a vectorized implementation, where $\bullet$ represents the Hadamard product):

$$\delta_k^l = \text{upsample}\left((W_k^l)^T \delta_k^{l+1}\right) \bullet f'(z_k^l) \tag{E.1}$$

To compute the partial derivatives for the filter maps, we must "flip" the delta matrix obtained from E.1, as per the mathematical definition of convolution. Where $\nabla_{W_k^l}$ denotes the gradient of weight W and $\nabla_{b_k^l}$ represents the gradient of bias term b in layer l respectively. The term a^l denotes the activation of layer l:

$$\nabla_{W_k^l} J(\theta) = \sum_{i=1}^{m} (a_i^l) * \text{rot90}(\delta_k^{l+1}, 2) \tag{E.2}$$

$$\nabla_{b_k^l} J(\theta) = \sum_{a,b} (\delta_k^{(l+1)})_{a,b}. \tag{E.3}$$