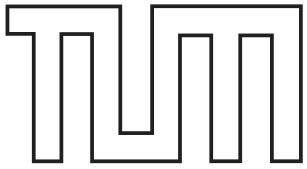


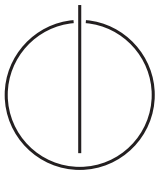
DEPARTMENT OF INFORMATICS

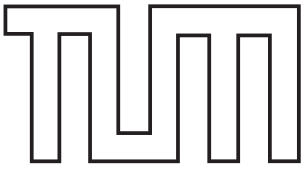
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Evaluation of Deep Residual Learning for Image Recognition

Maximilian Mumme





DEPARTMENT OF INFORMATICS

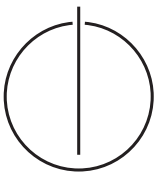
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Evaluation of Deep Residual Learning for Image Recognition

Implementierung und Evaluierung von Deep Residual Learning zur Bilderkennung

Author:	Maximilian Mumme
Supervisor:	Univ.-Prof. Dr. Thomas Huckle
Advisor:	Moritz August
Submission Date:	13/05/2016



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 13/05/2016 Maximilian Mumme

Acknowledgements

I would like to thank some people who supported me during the making of this thesis. First I thank Maximilian Tharr, Camille Mainz and Robin Gloster for proofreading my thesis. For providing me parts of the hardware to run my computations on I thank Maximilian Tharr, Camille Mainz, Robin Gloster and my father Erich Mumme. Therefore, I also thank the Leibniz Supercomputing Centre (LRZ) of the Bavarian Academy of Sciences and Humanities (BAdW) for the provisioning and support of Cloud computing infrastructure essential to this thesis.
I would like to thank all of my friends and family for supporting me during the creation of this thesis. The greatest thanks go to my advisor Moritz August who answered all of my questions quickly and reliably and who had a tip on hand for every problem I was facing.

Abstract

Image recognition has become an important field of research in the past decade. In this thesis we investigate on the recently released paper "Deep Residual Learning for Image Recognition" by He *et al.* [He+15a]. Nowadays the image recognition task is usually tackled using deep convolutional neural networks. However, performance of those networks suffers from the degradation problem. This term refers to the counterintutive effect that for models deeper than a certain threshold both training and test error start to increase again. He *et al.* cope with this problem by introducing deep residual learning. We implement this novel concept using Google's machine learning library TensorFlow and reproduce the training results from [He+15a] with our implementation.
We give an introduction to fully-connected, convolutional and residual neural networks in general. Using the model architecture presented in [He+15a] we describe some advanced concepts of deep learning that are relevant in practice as well as data preprocessing and model training and evaluation. For the most important ideas we also provide sample TensorFlow code.
Evaluating our implementation on the CIFAR-10 dataset [Kri09] we are able to verify the absence of the degradation problem. With all models but one we reach test error rates similar to the paper. Applying our implementation of deep residual models to the Yelp Restaurant Photo Classification dataset [Yel15] we show that deep residual learning is also applicable to a real world multi-label classification problem.

Contents

1 Introduction

Computer vision is a subdomain of computer science that has heavily gained scientific interest in the last years. One basic prerequisite for computer vision is a machine's ability to recognize images, i.e. to process a picture from the real world and make statements about its content.

Image recognition has a great variety of applications in lots of different areas. For example a rather simple image recognition task is the identification of handwritten letters and digits. Nowadays, machines recognizing handwritings work so well that they are used by post offices to identify adresses or by banks to process cheques.
Another application can be found in the emergent research area of autonomous driving. To be able to navigate through traffic without a human driver a car needs to continuously keep track of parameters like the course of a road as well as immediately recognize (possibly dangerous) exceptional situations, e.g. when a pedestrian unexpectedly crosses the road. Thus in the domain of autonomous driving (and robotics in general) correctly working image recognition is crucial for safety of systems and humans involved.
The same applies for image recognition in medicine and biology. For example take the field of computer aided diagnosis (CAD) where wrong classification of a disease can be fatal.
Finally image recognition also plays a role in user-machine interaction, e.g. for controlling computer devices with gestures, or in augmented reality, where a machine has to detect markers to combine real and virtual space.

Since image recognition has so many applications there is a great interest in research on this topic. Nowadays researchers agree that the task of image recognition is addressed best using a machine learning approach, i.e. building a model for a certain problem and training it with known examples so that it can generalize on unseen instances of the problem. Of course this approach will never work perfectly but will always introduce a certain amount of error. Minimizing this error for all the different peculiarities of the image recognition task is the current challenge in research.
One certain machine learning technique that has been found to work exceptionally well for image recognition is the approach of neural networks and deep learning. Since working with deep neural networks is extremely hardware intensive it is a relatively new area of research.
Roughly since 2010 international research teams, mostly backed by large global player companies like Google, Microsoft or Facebook, compete for the lowest error rates on different image recognition tasks and achieve new breakthroughs almost monthly. In

2015 Microsoft researchers presented the first image recognition model that was able to surpass human performance on a standard image classification task [He+15b].

In this thesis we are going to take a look at the latest success of the Microsoft research team led by Kaiming He. In December 2015 they published their work "Deep Residual Learning for Image Recognition" [He+15a] in which they slightly alter the currently prevailing deep learning approach to image recognition. This advance set new standards for lowest error rates in image classification and was able to win many prestigeous image recognition competitions in 2015.
We are going to implement their approach using Google's recently released numerical computation library TensorFlow [Aba+15]. First we try to reproduce their results on the standard image classification dataset CIFAR-10 [Kri09]. Then we are going to apply the implementation to a real world image recognition task, the Yelp Restaurant Photo Classification challenge [Yel15].

The rest of this thesis is organized as follows: In Chapter 2 we are going to give an overview of the basic concepts of machine learning, from supervised learning in general to neural networks and deep learning in particular. We are going to describe the evolution of neural networks from fully-connected networks over convolutional networks to the residual networks presented in [He+15a]. Chapter 2 will conclude with a short introduction to TensorFlow.
In Chapter 3 we are going to look at one concrete application of the basic concepts presented in Chapter 2, namely the models used in [He+15a] for image recognition on the CIFAR-10 dataset. We are going to describe their architecture as well as some advanced concepts of deep learning that become relevant in practice. Next we are going to look at data preprocessing and our settings in a model training environment. For the most important concepts of this chapter we are also going to present a sample implementation in TensorFlow.
In Chapter 4 we are going to describe hardware settings and methodology for our experiments with the models presented in Chapter 3. We will show our results and compare them to the results from [He+15a].
In Chapter 5 we are going to describe our implementational changes for the Yelp Restaurant Photo Classification challenge. We are going to look at the single training run conducted on the new dataset and present its results.
Finally, Chapter 6 will summarize what we did in this thesis and give an outlook on possible further work on the topic.

2 Basics

In this chapter we are going to describe the basics of machine learning and neural networks. We will give an introduction to supervised learning and present the idea of neural networks and deep learning, starting at a single neuron and proceeding over fully-connected and convolutional networks to residual neural networks.

2.1 Supervised learning

In machine learning there are three paradigms of learning, namely: *unsupervised learning*, where patterns in the input are learnt without any feedback, *reinforcement learning*, where learning is driven by a series of rewards and punishments, and *supervised learning* [Alt15]. Supervised learning can informally be described as the procedure of learning from examples. Formally a supervised learning scenario consists of a set $\mathcal{D}$ of n ordered pairs

$$\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_n, y_n)\} \subseteq X \times Y, \tag{2.1}$$

where the $\boldsymbol{x}_i$ are called *feature vectors* and y_i is the output belonging to a certain feature vector. This set is split into a *training set* $\mathcal{D}_{train}$ and a *test set* $\mathcal{D}_{test}$. Based on this the goal of supervised learning is to use training data to learn a function $h\colon X \to Y$ from the feature space X to the output space Y (called the *model*) that generalizes well on test data [Alt15; Lea14].
The emphasis in the previous sentence lies on *generalization*. If we train our model to fit training data too well, this may decrease performance on test data. This phenomenon is called *overfitting* [Sma]. The task to find a tradeoff between precision on training data and generalization on test data is the main challenge of supervised learning.
Training in supervised learning is usually done by minimizing some kind of loss function, which in its most general form can be written as

$$L(h \mid \mathcal{D}_{train}, \theta) = \sum_{(\boldsymbol{x},y)\in\mathcal{D}_{train}} l(h(\boldsymbol{x}) \mid y, \theta), \tag{2.2}$$

where $l(h(\boldsymbol{x}) \mid y, \theta)$ is the individual loss per data sample and θ are the model parameters. The goal of the training procedure is to adjust θ in a way that L is minimal. We are going to look at concrete examples of loss functions and training strategies in later sections.
The two central problems in supervised learning are regression and classification. In this thesis, we will restrict ourselves to classification, where the output space is a finite set of known size m. The m elements of the set are called *labels*.

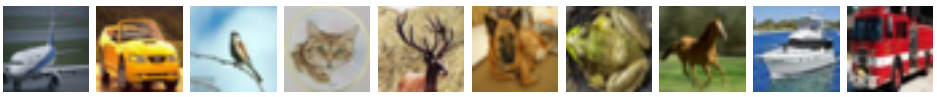


Figure 2.1: Sample images from the CIFAR-10 dataset.

The CIFAR-10 dataset for image recognition

The machine learning methods presented in the following all tackle one popular supervised classification problem, the task of image recognition. The challenge with this is, given a set of images and class labels describing the content of each image, to build a model that predicts the classes of unknown images as well as possible.

One possibility to fit image classification in the formal definition above is to take the RGB values of image pixels as features and the the content-describing labels as outputs. This is the task that the methods presented in the following sections address.

A standard dataset to measure and compare performance of image classifiers is the CIFAR-10 dataset [Kri09]. It consists of 60k ($= 60 \cdot 10^3$) RGB images, each of size 32×32 pixels. Each image belongs to exactly one out of 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.

The classes are of equal size, i.e. there are 6k images of each class. The dataset is split into a 50k images training set and a 10k test set. The equal distribution of classes is maintained by the split, meaning that there are 5k images of each class in the training set and 1k of each class in the test set.

Since CIFAR-10 is a simple yet popular dataset, it will be used for evaluation and comparison of the models introduced in this thesis.

2.2 Neural networks and deep learning

One approach in supervised learning that has been found to perform exceptionally well on image classification tasks is the concept of neural networks. This section will give an overview starting from the basic element of such a network, called a *neuron*, up to the state-of-the-art deep neural models that have been implemented for this thesis.

2.2.1 From the perceptron to feedforward neural networks

The idea of a neuron can be illustrated best using one special instance of such, the *perceptron*, developed by F. Rosenblatt in the early 1960s [Ros62]. A perceptron takes a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ of binary inputs and produces a single binary output y. This output being 0 or 1 is determined by whether the weighted sum of the inputs $\sum_i w_i x_i$ is

greater or less than a threshold t, i.e.

$$y = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq t \\ 1 & \text{if } \sum_i w_i x_i > t \end{cases}. \tag{2.3}$$

So by adjusting the weights of several features, one can alter the importance of these for the output decision, and by increasing (decreasing) the threshold, one can increase (decrease) the overall chance of the perceptron to fire (= to output 1). To simplify notation, we introduce $b = -t$, so that we can change the output function to

$$y = \begin{cases} 0 & \text{if } \sum_i w_i x_i + b \leq 0 \\ 1 & \text{if } \sum_i w_i x_i + b > 0 \end{cases}. \tag{2.4}$$

b is then called the *bias* and can be seen as a measure of how hard it is to make the neuron fire [Nie15].

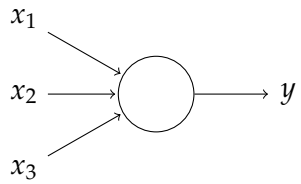


Figure 2.2: A perceptron with $n = 3$.

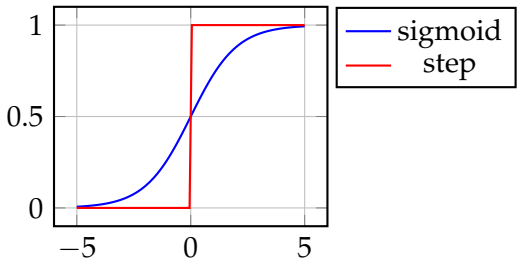


Figure 2.3: The sigmoid function as a smoothed version of the step function.

However, this is not the type of neuron being used in neural networks. The *step function* the perceptron uses for getting an output of 0 or 1 is not differentiable and hence will not work with the learning algorithms proposed later.
Therefore, we introduce a differentiable, non-linear *activation function* h, so that

$$y = h(\boldsymbol{w}\boldsymbol{x} + b) \tag{2.5}$$

[Bis06] with the weighted sum written as the dot product of weight vector $\boldsymbol{w}$ and feature vector $\boldsymbol{x}$ for clarity. The most prominent example for such an activation is the *sigmoid function* $\sigma(\boldsymbol{w}\boldsymbol{x} + b) = \frac{1}{1+e^{-(\boldsymbol{w}\boldsymbol{x}+b)}}$. Now since the output of a neuron is real-valued, we need to introduce a threshold on the activation to decide if the neuron should figure. By inspection of Figure 2.3 in case of the sigmoid function the most obvious choice for the threshold would be 0.5, i.e. the neuron outputs 1 for $y > 0.5$ and 0 otherwise.

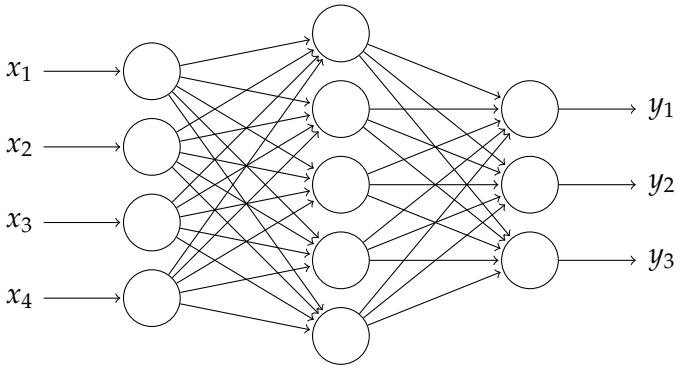


Figure 2.4: A small neural network with one hidden layer.

A single neuron is only capable of making a simple "yes-or-no" decision based on the weighted features. To be able to make more complex decisions like the classification of an image, one chains multiple layers of neurons together by feeding the output of one layer as features into the next layer. Figure 2.4 shows a *fully-connected feedforward neural network* with an *input layer*, a *hidden layer* and an *output layer* [Nie15]. The term "feedforward" originates from the fact that information in the network only flows from one layer to its successor, there are no backward loops or connections inside a layer. Note that the input layer is a rather formal construction. It does neither have weights nor biases and just distributes the inputs to each of the hidden neurons. Regarding classification tasks the output layer has one neuron for each class. The class assigned to a certain example is then simply chosen by the neuron having the maximum activation output.
Although it is proven that such a network with one hidden layer "can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has a sufficiently large number of hidden units" [Bis06, p. 230], the upper model is usually extended by adding more hidden layers. Those models are then called *deep neural networks* and are preferred over the single-hidden-layer networks due to their better applicability to complex problems like image recognition [Nie15]. This originates from the fact that there exist functions that deep neural networks can model using a polynomial number of neurons, while a shallower network would need an exponential number of neurons [BS14].
All models, algorithms and tools that implement machine learning with deep neural networks are summarized under the term *deep learning* [Nie15].

Learning in neural networks

As usual in supervised learning, when training neural networks one tries to minimize a certain loss function computed over the training set. Let L be such a loss function defined as the sum over all single-sample losses like in equation 2.2. A numeric optimization algorithm that has been found to be very simple and computationally cheap yet effective

regarding neural networks is *gradient descent*. In general its iterative update rule can be written as

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla L, \tag{2.6}$$

where t is the iteration step, ∇L is the gradient (the vector of all partial derivatives) of the loss function, and η is called *learning rate* [Bis06]. As before θ accumulates all the model parameters. In terms of neural networks these are the weight matrices $\boldsymbol{w}$ and bias vectors $\boldsymbol{b}$ for each layer. The optimization has finished when it *converges*, i.e. when after a certain number of repetitions of the update rule the change in parameters θ drops below a certain threshold. However, this is not always the case. Whether an optimization converges strongly depends on the given model, dataset and other hyper-parameters.

One problem when using gradient descent is that the loss is a function of the whole training set which can take very long to compute for large amounts of training data. So in order to increase performance one computes the loss only over a randomly selected subset $\mathcal{B}$ of the training data, called a *mini-batch*. In this case, equation 2.2 changes to

$$L_{\mathcal{B}}(h \mid \mathcal{D}_{train}, \theta) = \sum_{(\boldsymbol{x},y)\in\mathcal{B}} l(h(\boldsymbol{x}) \mid y, \theta) \text{ for any } \mathcal{B} \subset \mathcal{D}_{train}. \tag{2.7}$$

If the mini-batch size is chosen large enough, this yields a good estimate of the loss over the whole training set. Although more sophisticated optimization methods for neural networks have been developed, this algorithm called *stochastic gradient descent* (with some small improvements) is still widely used in recent research [Nie15].

There is one challenge remaining from the previous paragraphs: We need to compute the gradient of L, i.e. the partial derivatives of L with respect to each model parameter in θ to be able to apply the update rule from equation 2.6. An algorithm tackling this problem has been developed by Rumelhart *et al.* in 1986 [RHW86]. It is called *error backpropagation* and is still relevant today due to its computational efficiency.
The idea of the backpropagation algorithm is to first *forward propagate* an input vector $\boldsymbol{x}$ through the network to obtain the activations for each layer and then, starting from the output layer, *backpropagate* the loss to update weights and biases. The fact that the loss in a layer is computed based on the (already calculated) loss of its predecessor during the backpropagation phase is what makes the algorithm that efficient.

For the mathematical derivation of the algorithm, adapted from [Bis06], it is easier to look at the weights and biases component-wise. We introduce the notation w^l_{ji} for the weight of the connection from the i^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer. Note the flip in the subscript indices that seems a bit unintuitive at first sight. However, ordering the indices this way round makes it easier to transform the component-wise representation into the matrix form since otherwise we would need to transpose the weight matrix to perform the matrix multiplications. Respectively, by b^l_j we denote the bias of the j^{th} neuron in the l^{th} layer.

Also recall the loss function being defined as the sum of per-sample losses. We are only going to define the backpropagation procedure for the loss l of a single sample since the derivative of the total loss can easily be computed as the sum of the single-sample loss derivatives. Because the formulas of the algorithm work equivalently for each layer (except of the output layer), we are also going to drop the layer index l for weights and biases.
So given the network's forward propagation rule

$$a_j = \sum_i w_{ji} z_i + b_j, \qquad z_j = h(a_j) \tag{2.8}$$

we would like to compute the partial derivatives of the loss function with respect to each layer weight and bias, i.e. $\frac{\partial l}{\partial w_{ji}}$ and $\frac{\partial l}{\partial b_j}$.
Therefore we apply the chain rule for partial derivatives which yields

$$\frac{\partial l}{\partial w_{ji}} = \frac{\partial l}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}, \qquad \frac{\partial l}{\partial b_j} = \frac{\partial l}{\partial a_j}\frac{\partial a_j}{\partial b_j}. \tag{2.9}$$

Using the definition of a_j (2.8) the second terms can be written as

$$\frac{\partial a_j}{\partial w_{ji}} = z_i, \qquad \frac{\partial a_j}{\partial b_j} = 1. \tag{2.10}$$

For the first terms we introduce the notation

$$\delta_j := \frac{\partial l}{\partial a_j} \tag{2.11}$$

and refer to it as the *error* of a neuron j. By substituting 2.10 and 2.11 into 2.9, we get

$$\frac{\partial l}{\partial w_{ji}} = \delta_j z_i, \qquad \frac{\partial l}{\partial b_j} = \delta_j. \tag{2.12}$$

Now we are going to define a method with which we can backpropagate the errors δ_j from one layer to its predecessor, starting at the output layer. For the output layer, we can directly compute the error as

$$\delta_j = \frac{\partial l(z_j)}{\partial a_j} = \frac{\partial l(h(a_j))}{\partial a_j} = \frac{\partial l}{\partial z_j} h'(a_j), \tag{2.13}$$

where h' is the first derivative of h. For the input and hidden layers, we apply the chain rule once again. With the total differential we then obtain

$$\delta_j = \frac{\partial l}{\partial a_j} = \sum_k \frac{\partial l}{\partial a_k}\frac{\partial a_k}{\partial a_j}, \tag{2.14}$$

where the summation is over all neurons k in the next layer that neuron j is connected to. Now substituting the definition of δ_k (2.11) and using the fact that $a_k = \sum_j w_{kj} z_j + b_k = \sum_j w_{kj} h(a_j) + b_k$ (2.8) and therefore $\frac{\partial a_k}{\partial a_j} = w_{kj} h'(a_j)$ we get

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k. \tag{2.15}$$

Now we have a formula to compute the error δ_j of the neurons in one layer depending on the error δ_k of the neurons in the succeeding layer. To sum things up, the backpropagation algorithm works as follows:

1. Forward propagate an input through the network and collect the neurons' activations using 2.8.
2. Directly compute δ_j for the output neurons with 2.13.
3. Backpropagate the δ_j through the network using 2.15.
4. With 2.12, calculate the derivatives of all weights and biases.

Since we have all the required derivatives, we can now collect them in the gradient vector, sum the gradients over the whole training data $\mathcal{D}_{train}$ or the mini-batch $\mathcal{B}$ and plug the result into 2.6 to compute the weight and bias update.

Despite in theory all presented techniques also work fine for deep fully-connected networks, there are some issues that make those networks harder to train in practice, e.g. the effect of vanishing and exploding gradients [Nie15], which we will address later in more detail. In the next section we are going to introduce an improved network architecture that is more applicable in practice and thus is the means of choice in state-of-the-art image recognition.

2.2.2 Convolutional neural networks

One big downside of fully-connected neural networks is that they do not make use of the two-dimensional structure of an image. They just connect each neuron (one neuron represents one image pixel in the case of image input) to each other, no matter if they are close or far away in the original image.
A *convolutional neural network* does take the spatial arrangement of neurons into account. Consider a neural network that takes a two-dimensional array of neurons as inputs, and each neuron in the following layer is only connected to an $m \times n$ area of neurons (see Figure 2.5). This area is called the *local receptive field* of the next layer neuron. In order to obtain a two-dimensional output layer, the local receptive field is then shifted over the input layer, each step by a certain number of neurons called the *stride* length [Nie15]. For simplicity, we are going to assume a stride of 1 at first.

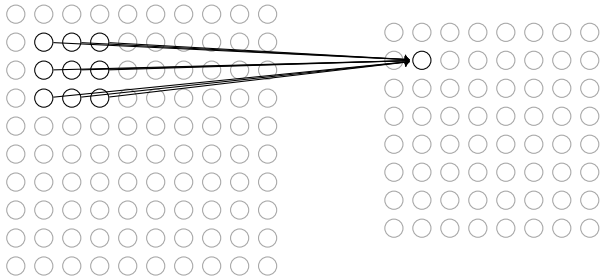


Figure 2.5: A 3×3 local receptive field of a neuron on a 10×10 input layer. To produce the displayed output layer a stride of 1 is used.

Of course since we have two-dimensional inputs and only local connections between the layers, we need a different way to compute the activation of a neuron. The following equations are adapted from [GBC16]. Given an $m \times n$ local receptive field, depending on the activations in the $(l-1)^{\text{th}}$ layer the activation z_{ij}^l in the l^{th} layer can be calculated as

$$a_{ij}^l = b + \sum_{r=1}^{m} \sum_{c=1}^{n} w_{rc} z_{i+r-1,j+c-1}^{l-1}, \qquad z_{ij}^l = h(a_{ij}^l). \tag{2.16}$$

Note that the weights and bias used do not depend on the spatial location of the neuron. This is another property of convolutional layers. They use the same weights and bias for each convolution operation. This means we only have $m \times n$ weights and one bias in total. We call these shared weights and bias a *kernel*. The intuition behind this is that one convolution with its specially trained kernel detects one certain *feature*, e.g. an edge or a circle, no matter where this feature is placed in the image [Nie15].
But if a convolutional layer could only detect one feature, we would need an awful lot of layers to perform complex image recognition. This is the reason why convolutional layers introduce a third dimension in our data. We call this dimension *feature maps* or *channels*. Each channel has its own kernel and thus a convolutional layer with k channels can detect k features in total [Nie15]. Using feature maps we need to extend the upper formula 2.16. Again, given a kernel of size $m \times n$, the shared weights $\boldsymbol{w}_{fg}$ connecting the f^{th} feature map in layer $l-1$ to the g^{th} feature map in layer l, and the bias b_g, the activation z_{ijg}^l can be computed as

$$a_{ijg}^l = b_g + \sum_{d=1}^{f} \sum_{r=1}^{m} \sum_{c=1}^{n} w_{rcdg} z_{i+r-1,j+c-1,d}^{l-1}, \qquad z_{ijg}^l = h(a_{ijg}^l). \tag{2.17}$$

Note that each feature map in one layer is connected to each feature map in the succeeding layer which gives us a nice analogy to fully-connected networks. The intuition of that is clear, too: Depending on the simple features detected in the $(l-1)^{\text{th}}$ layer one can combine them to detect more complex features in the l^{th} layer.
Up to now, we have always assumed the stride to be 1. Sometimes we might want to reduce the spatial resolution of the input to decrease data complexity. Therefore we take

a stride $s > 1$, i.e. we move the local receptive fields in steps of s neurons. Once again we need to adjust equation 2.17 to its most general form

$$a^l_{ijg} = b_g + \sum_{d=1}^{f} \sum_{r=1}^{m} \sum_{c=1}^{n} w_{rcdg} z^{l-1}_{s(i-1)+r,s(j-1)+c,d}\,, \qquad z^l_{ijg} = h(a^l_{ijg}). \tag{2.18}$$

Note that a convolution with stride 1 also reduces the spatial resolution of the input. Assume we have a $h \times w$ input image and apply a convolution with kernel size $m \times n$ then the resulting output will only have a resolution of $(h - m + 1) \times (w - n + 1)$. This fact (visualized by Figure 2.5) can be cumbersome since a declining resolution limits the maximum number of convolutional layers that can be used in a deep network. To circumvent that issue the input is usually padded with zeros in a way that height and width of the input are preserved throughout the convolution. For a kernel size of $m \times n$ padding of $m - 1$ rows and $n - 1$ columns is necessary. Usually these extra rows and columns are distributed equally to the left/right and top/bottom of the input. To obtain desired output shapes sometimes padding is also needed for strides greater than 1 [GBC16].

Pooling layers

In addition to convolutions, a convolutional neural network often uses *pooling layers*. Pooling works similar to convolution as it also utilizes local receptive fields. But instead of weighting the inputs with a kernel, pooling neurons simply compute a statistical function over their receptive field. The most common pooling functions are *max-pooling* that returns the maximum input from the receptive field, and *average-pooling* that calculates the arithmetic mean over the inputs [GBC16].
Pooling layers do not have any parameters. This means they neither have ability to learn nor do they introduce additional feature maps. Their use case is, once again, simplifying data by reducing the spatial resolution. The intuition behind pooling is the following: By placing it behind one or more convolutions, pooling neglects the exact positional information of a detected feature. In image recognition this boosts the network's ability to generalize because it makes the network robust to translations in the image input, i.e. the presence of a feature and its relative position to other features is more important than its pixel-precise location in the image [Nie15].
To be able to achieve any spatial resolution required pooling also can be applied strided. Of course because we still use local receptive fields also padding could be necessary. Since with pooling size reduction is intentional, most of the time the same-size padding described above is not applied, but rather padding of a few rows and columns to obtain a desired output height and width when using a certain stride.

Having introduced a totally new architecture of neural networks our previously derived backpropagation algorithm will not work anymore. However, it is possible to modify the upper equations to work for convolutional and pooling layers although doing so would go beyond the scope of this thesis.

2.2.3 Residual neural networks

Up to 2015 the best performing models for image recognition like VGG-16 [SZ14] or GoogLeNet [Sze+15] all had a plain convolutional architecture. In a recent publication [He+15a] released in December 2015 He *et al.* presented a new kind of architecture that set another milestone in the accuracy on standard image classification tasks. Their approach called *deep residual learning* that is going to be implemented and evaluated in this thesis was able to make the first place in the ILSVRC[1] and MS COCO[2] 2015 image recognition competitions.
Although research has shown that a convolutional neural network's performance on test data can be improved by adding more layers [Sze+15; SZ14], it has also found that there is a limit on test set accuracy with increasing depth [SGS15; He+15a]. At this limit, usually somewhere between 15 and 30 layers, test error is not only stagnating. When adding even more layers a decline in test data performance is observed. This issue is known as the *degradation problem*.

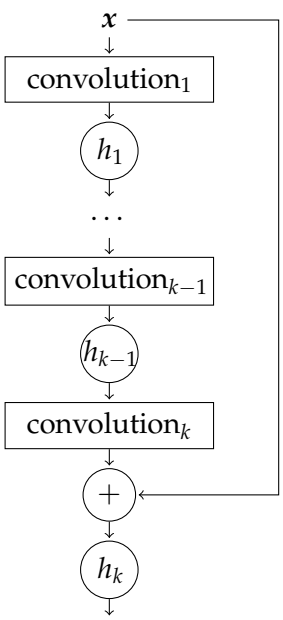


Figure 2.6: A residual building block as proposed in [He+15a] in general form.

He *et al.* address that problem in their paper. They argue that the degradation cannot be caused by overfitting since besides an increasing test error also a rise in the training error can be observed. From the increase of the training error they conclude that those deeper models must be harder to optimize than their shallower counterparts using current optimizers (like equation 2.6 and its enhancements).
To circumvent that difficulty they propose an approach that supports optimizers in minimizing the error on blocks of convolutional layers. Assume an optimized relatively shallow neural network. One can always construct a deeper model with the same training and test error by adding layers performing identity mappings. Thus in theory a model with more layers can never have a lower accuracy than one with fewer layers. He *et al.* state that optimizers therefore must have difficulties in approximating identity mappings.
Assume a stack of convolutional layers computing the function $\mathcal{H}(\boldsymbol{x})$ on an input vector $\boldsymbol{x}$ (or an input matrix in the case of image recognition) in the optimal state. If $\mathcal{H}$ is the identity function (or is at least close to it), an optimizer will not be able to approximate it very well. Now suppose we could change the network

[1] `http://image-net.org/challenges/LSVRC/2015/`
[2] `http://mscoco.org/dataset/#detections-challenge2015`

architecture so that the optimizer only has to train the *residual function*

$$\mathcal{F}(\boldsymbol{x}) = \mathcal{H}(\boldsymbol{x}) - \boldsymbol{x}. \tag{2.19}$$

If now $\mathcal{H}$ is close to the identity function, $\mathcal{F}$ will be close to the zero function. This should be much easier to approximate for the optimizer. By rearranging 2.19 the original function can be written as

$$\mathcal{H}(\boldsymbol{x}) = \mathcal{F}(\boldsymbol{x}) + \boldsymbol{x}. \tag{2.20}$$

So by adding the input $\boldsymbol{x}$ after the stack of convolutions we can make the optimizer approximate the residual function $\mathcal{F}$ and still obtain the desired output $\mathcal{H}$.

In a network architecture residual learning is implemented by so called *shortcut* connections that bypass a stack of convolutional layers. Figure 2.6 shows a residual building block with k layers. As He *et al.* suggest the last activation function h_k is applied after the addition. So convolutions 1 to $k-1$ are computed as in equation 2.18, for the last convolution one applies the first part of 2.18 and then evaluates

$$\boldsymbol{z} = h_k(\boldsymbol{a} + \boldsymbol{x}), \tag{2.21}$$

written in matrix form for simplicity.
Since the addition in 2.21 is simple component-wise addition it will only work if the output of the convolutions $\boldsymbol{a}$ and the input $\boldsymbol{x}$ have the same dimensions. For what to do if one or more convolutions in the building block reduce the spatial resolution of the input and increase the number of feature maps He *et al.* suggest three options:

A *Identity mapping*. For downsizing the input 1×1 max-pooling with a proper stride is used. Missing feature maps are padded with zeros.

B *Linear projection*. For all kinds of dimension adjustments a linear projection is used so that equation 2.21 changes to

$$\boldsymbol{z} = h_k(\boldsymbol{a} + \boldsymbol{w}_s\boldsymbol{x}). \tag{2.22}$$

Because the projection matrix $\boldsymbol{w}_s$ is learnt like a normal weight matrix this option introduces *residual learning*.

C *Always use linear projections*. Linear projections like in option B are used, but also when dimensions do not change. This creates even more learnable parameters.

This approach made it possible for He *et al.* to benefit from accuracy gains of very deep models without experiencing the degradation problem.

2.3 Machine learning with TensorFlow

Before we are going to implement the residual models from [He+15a] in this section we will give a short general overview on the framework used for the implementation.
TensorFlow [Aba+15] is an open-source numerical computation library originally developed by Google. It was released in November 2015 and is currently in version 0.8. However, for the implementation part of this thesis we used the slightly older version 0.7.1. The framework is written in C++ for maximum efficiency and is accessible through a very high-level Python API. TensorFlow comes in two versions: one that only supports CPUs and another with additional GPU support. Since most numerical computations involve a lot of matrix operations GPUs are the means of choice when aiming for fast and efficient computations.

The central type of data in TensorFlow are *tensors*. Tensors are the generalization of matrices in more that two dimensions. For example we are going to work with four-dimensional tensors when implementing our neural models, where the first dimension is the batch offset, second and third dimension are the row and column indices of the image and the fourth dimension is the channel offset. Those tensors then traverse a data flow graph, hence the name of the framework. In this graph, nodes are computational operations and edges indicate the directions the tensors flow.
TensorFlow is ideally, but not exclusively, suitable for implementing neural networks and all the overhead needed for training and evaluating them. There exist efficient implementations of lots of standard neural network components like convolutions, pooling and activation functions as well as functions assisting the programmer in data preprocessing and output evaluation.

```
1  import tensorflow as tf
2
3  a = tf.constant([27, 35, 8], dtype=tf.int32, shape=[3], name='a')
4  b = tf.constant([15, 7, 34], dtype=tf.int32, shape=[3], name='b')
5  c = tf.add(a, b)
6
7  sess = tf.Session()
8  result = sess.run(c)
9  print result
10 sess.close()
11
12 # prints [42, 42, 42]
```

Listing 2.1: A simple example of graph definition and execution in TensorFlow.

The first thing to get comfortable with when using TensorFlow is its two-phases principle. In the first phase the data flow graph is built using the Python API, in the second phase computations defined by the graph are executed on the compiled C++ code. Let's look at the simple example from Listing 2.1.

In the graph building phase first two constant tensors `a` and `b` are created and initialized with some arbitrary values. Here the data type is explicitly set to `int32` and the shape is defined as "1D tensor with length 3" although this is not necessary. TensorFlow can infer shape and data type from the given initialization. Then `c` is defined as the (component-wise) sum of `a` and `b`. Note that `c` does not hold the result of the addition but rather an operation (or *op* in the TensorFlow jargon) instructing the C++ library to compute `a + b`.
The execution phase starts with opening a `Session`. By passing the operation `c` to the session's `run` method one can tell TensorFlow to compute all steps necessary to obtain the result of `c` - in our case only the addition of two constants `a` and `b`. The type of the `result` variable is a *Numpy* integer array that can be handled like a normal Python list.

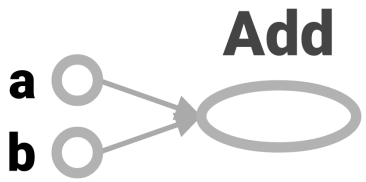


Figure 2.7: The execution graph of Listing 2.1.

TensorFlow comes with two utilities that support the programmer in debugging and analyzing the often lengthy network training processes. Firstly, it offers the possibility to store *checkpoints* of the model parameters at any time in the training progress. This allows for an analysis of the model at a certain training time even though the training has already finished as well as resuming training from a particular step in time. We will use checkpoints to create a chart of the test error over training time after the actual training has already been done.
The second utility is *TensorBoard*, a powerful statistical analysis tool for training progress and network performance. It is a standalone server application running independently from TensorFlow execution graphs. By adding *summaries* to a graph, TensorFlow can write any value produced during graph execution to the disk, once again at any step in time. These values are then collected by TensorBoard and processed nicely into charts that can be accessed through the browser interface of the utility. Also a grahical view of the whole execution graph is available (see Figure 2.7 for a representation of the graph constructed in listing 2.1). All charts in Chapter 4 are created using the CSV export functionality of TensorBoard.

From the view of the author, TensorBoard is one of the great advantages of TensorFlow. It saves the developer the trouble of having to implement evaluation charts that are necessary in nearly every machine learning application on his own. Another benefit is TensorFlow's automatic hardware placement. It detects available hardware (CPUs and GPUs) and puts operations where they perform best. This makes it possible to switch from the CPU to the GPU version of TensorFlow without changing a single line of code. Additionally, operations are run in parallel (if possible) to ensure full usage of the hardware for maximum computation speed. Which operations to execute in parallel is inferred from the graph so once again the programmer does not have to think about

it. Another thing TensorFlow can infer from the graph are tensor shapes. This is very useful for debugging since many bugs caused by non-matching tensors are detected immediately and not after hours of training.
Since TensorFlow is relatively new its documentation is not very good yet. This does not mean that features are not very well documented (indeed in most cases they are) but that the online community of TensorFlow is not very big. There are not many developers using it yet so that it is rather hard to find solutions to your issues on forums like Stackoverflow[3], simply because nobody has experienced the same problem up to now. Also due to it being fairly new there are still bugs and unsupported features in the TensorFlow implementation, e.g. 1×1 convolution and the Nesterov accelerated gradient optimizer. Of course it is just a matter of time that both of these downsides are going to vanish.

[3] `http://stackoverflow.com`

3 Implementation

In the last chapter we discussed the theoretical backgrounds of neural networks in general and of deep residual learning in particular. In this chapter we are going to look at one special residual network architecture and its implementation in TensorFlow. We will also introduce some advanced concepts that are not necessary for understanding the idea of neural networks, but are of great importance for deep learning in practice since they dramatically increase model performance in terms of learning speed and classification accuracy. We are also going to show the implementation of CIFAR-10 image preprocessing as well as model training and evaluation routines.

3.1 Residual network architecture

In this section we are going to look at the residual network architecture that is used by He *et al.* in [He+15a] for their studies on the CIFAR-10 dataset. In some aspects we are going to differ from the original implementation and follow the suggestions of [GW16] that implemented the residual models in Torch[1] and open-sourced their code on GitHub[2].

3.1.1 Architecture overview

When using CIFAR-10 the input images are always represented as a three-dimensional tensor with size $32 \times 32 \times 3$. Note that we already have three feature maps in the input. These are the red, green and blue color channels of the RGB encoded image. The first and second dimension are height and width of an image, which is 32×32 pixels in the case of CIFAR-10. Because we want to use a variant of stochastic gradient descent for optimization, we will finally input a batch of 128 images into our network. This means that the actual input tensor is a $128 \times 32 \times 32 \times 3$ tensor, although we are going to neglect the batch dimension for architecture description.

[1] `http://torch.ch/`

[2] `https://github.com/facebook/fb.resnet.torch`

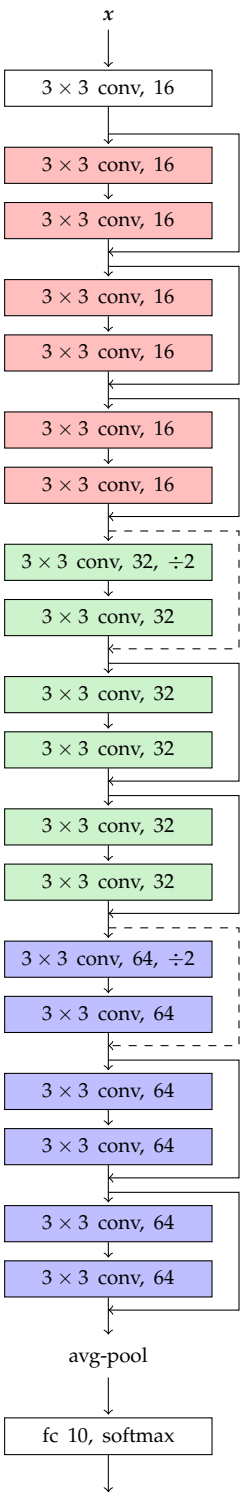


Figure 3.1: The architecture of ResNet-20.

He *et al.* use a systematic scheme to build residual models of different sizes. They first place a single 3×3 convolutional layer to get from 3 to 16 feature maps. Then they use $6n$ convolutions, where $n \in \{3, 5, 7, 9, 18, 200\}$. Each two of them are grouped in a residual building block, i.e. a shortcut connection bypasses two layers. These $6n$ layers are split into $2n$ layers with a spatial resolution of 32×32, $2n$ with 16×16 and $2n$ with 8×8. Each time they halve the input size the number of feature maps "is doubled so as to preserve the time complexity per layer" [He+15a, p. 3]. So starting at 16 channels they get 32 and 64 channels, respectively. Downsizing the input is done by convolution with stride 2. Since the strided convolution lies within a residual building block we also need to downsize the bypassed input. Following [He+15a] we use downsizing option A here, i.e. identity mapping with 1×1 max-pooling and stride 2 for resolution reduction, and zero padding to add the missing feature maps. This does not increase computational complexity since no residual learning parameters are introduced.
After the convolutions a global average pooling is placed. Global pooling means pooling on the whole input, so we have 8×8 average pooling here. This operation outputs 64 feature maps of size 1×1 that are then fed into a fully-connected layer with 10 neurons for the 10 classes of CIFAR-10. The fully-connected layer uses a special activation function called *softmax* which we are going to introduce later.
In total, this general residual model has $6n + 2$ layers, the initial convolution, $6n$ layers in the residual building blocks and the fully-connected layer. Average pooling is not counted as a separate layer since it does not have learnable parameters. With the chosen values of n He *et al.* build models with $k \in \{20, 32, 44, 56, 110, 1202\}$ layers. They refer to the network with k layers by ResNet-k. Table 3.2 gives an overview on the models used. As an example Figure 3.1 shows the complete architecture of the smallest network, ResNet-20.

Having described the overall architecture of the networks we are now going to delve into some implementational details. For that we are going to look at the

layers	output resolution	feature maps
$1 + 2n$	32×32	16
$2n$	16×16	32
$2n$	8×8	64

Table 3.1: General architecture of the residual models. Table adapted from [He+15a].

n	total layers	name
3	20	ResNet-20
5	32	ResNet-32
7	44	ResNet-44
9	56	ResNet-56
18	110	ResNet-110
200	1202	ResNet-1202

Table 3.2: Residual networks used in [He+15a].

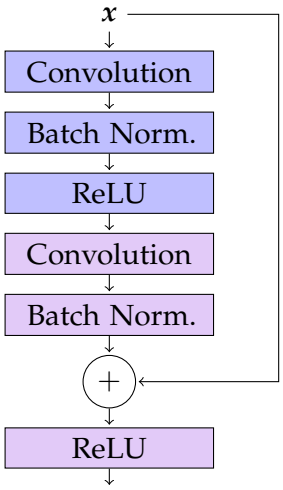


Figure 3.2: A residual building block as proposed in [He+15a]. Figure adapted from [GW16].

detailed implementation of a residual building block in Figure 3.2. As we can see the shortcut bypasses the already mentioned two convolutional layers. But in some aspects the implementation differs from the general form displayed in Figure 2.6. Firstly, an operation called *batch normalization* is placed after each convolution. Secondly, the general activation function h is replaced by a special instance of such called *rectified linear unit* (ReLU). Both are advanced concepts of deep learning that we are going to look at now.

3.1.2 Rectified linear units

The aforementioned problem of *vanishing gradients* (and its counterpart, the *exploding gradients* problem) does also arise in convolutional and residual neural networks when using sigmoid activation functions. Suppose during training the value for a particular activation is near 0 or 1 (the corresponding neuron *saturates* near 0 or 1). When looking at Figure 2.3 we see that close to 0 or 1 the slope of the sigmoid function is very flat, i.e. its gradient tends towards zero [Nie15].
When recalling equations for the backpropagation algorithm, we see from equation 2.12 that the gradient of the loss function with respect to weights and biases multiplicatively depends on δ_j. This δ_j is computed by function 2.15 that contains the gradient h' of the activation as a factor. This means when h' is near zero (as described above) then δ_j and hence the gradient of the loss function also go towards zero [Nie15].
Now also recall that δ_j multiplicatively depends on the δ_k of the succeeding layer. So when the δ_k are already close to zero, δ_j is even closer. The strength of this effect

increases the further one goes back in the network. The gradient of the loss function vanishes [Nie15].
When applying gradient descent as in equation 2.6 vanishing gradients cause that weights and biases of the upper layers are only updated to a very small extent. This leads to a *learning slowdown*, i.e. the number of training steps until convergence increases [Nie15].

One approach to this issue is introducing a new kind of activation function, the *rectified linear function*

$$h(a) = \max(0, a). \tag{3.1}$$

Neurons using this activation function are called *rectified linear units* (ReLUs). Research has shown that using ReLUs can dramatically speed up training and allows much deeper networks to converge [NH10; He+15b; GBB11; MHN13; Zei+13].

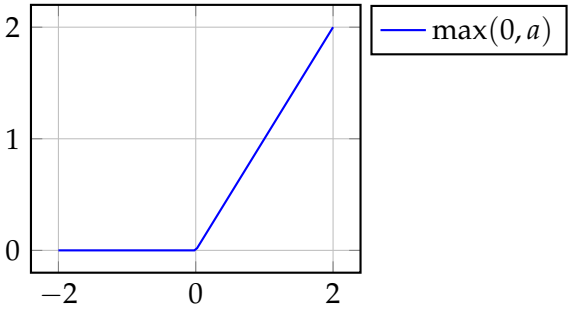


Figure 3.3: The rectified linear function.

There are multiple reasons for this behaviour. In positive direction those neurons do not saturate (the value for h goes to infinity with growing input a), i.e. there is no cause for vanishing gradients. By returning zero for each input $a \leq 0$ ReLUs also introduce *sparsity* into the network which is beneficial for learning [GBB11]. Sparsity means that there are many connections with plain zero weigths and only a few with positive or negative weigths. Additionally since the derivative of the rectified linear function can be written as

$$h'(a) = \begin{cases} 0 & \text{if } a < 0 \\ 1 & \text{if } a > 0 \end{cases} \tag{3.2}$$

[Sed14], its nature of being only a case differentiation reduces computational complexity compared to activations like the sigmoid [Zei+13]. Note that the function is non-differentiable at 0, although this does not matter in practice since it suffices to compute the derivative for any value arbitrarily close to zero[3].
Because we can write the derivative as above it is also easy to adapt the backpropagation algorithm to rectified linear units. This enables us to still use gradient descent with the ReLU.

[3] `https://en.wikipedia.org/wiki/Rectifier_(neural_networks)`

3.1.3 Batch normalization

Another effect reducing the convergence speed of a training procedure is called *internal covariate shift*. This term refers to the phenomenon that the stochastic distribution of the input to each layer changes with a change in the parameters of its predecessor. This has been found to be obstructive to learning because neurons need to adapt to a new distribution with each training step. The effect is particularly strong when using saturating nonlinearities like the sigmoid activation [IS15].

In their paper [IS15] Ioffe & Szegedy address this problem by introducing so called *batch normalization*. Batch normalization is a method to recreate a normal distribution on the output after having applied one or more layers' transformations.
Therefore they make use of the fact that network training is usually done in batches (using stochastic gradient descent). Assume that a layer applies the transformation $\boldsymbol{a}^l = \boldsymbol{w}\boldsymbol{z}^{l-1} + \boldsymbol{b}$. Then the output is normalized using the standard normalization formula

$$\hat{\boldsymbol{a}}^l = \frac{\boldsymbol{a}^l - \mathbb{E}[\boldsymbol{a}^l]}{\sqrt{\mathrm{Var}[\boldsymbol{a}^l]}} \tag{3.3}$$

before applying the activation h to obtain $\boldsymbol{z}^l$. Mean and variance are computed over the current mini-batch, i.e.

$$\mathbb{E}[\boldsymbol{a}^l] = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{a}^l \in \mathcal{B}} \boldsymbol{a}^l, \qquad \mathrm{Var}[\boldsymbol{a}^l] = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{a}^l \in \mathcal{B}} (\boldsymbol{a}^l - \mathbb{E}[\boldsymbol{a}^l])^2. \tag{3.4}$$

However, by giving the example of inputting the normalized values into a sigmoid activation Ioffe & Szegedy then show that normalization would restrict the sigmoid to outputs only from its linear area (because the normalized values have mean 0 and variance 1 which mostly yields outputs in the range -1 to 1 where the sigmoid is nearly linear, see Figure 2.3). This means normalization weakens the representational power of a neuron.
To address this they introduce two new learnable parameters $\boldsymbol{\gamma}^l$ and $\boldsymbol{\beta}^l$ that are capable of reverting the normalization to some extent by applying the linear transformation

$$\tilde{\boldsymbol{a}}^l = \boldsymbol{\gamma}^l \hat{\boldsymbol{a}}^l + \boldsymbol{\beta}^l. \tag{3.5}$$

In fact, they could even recover the unnormalized state by learning $\boldsymbol{\gamma}^l = \sqrt{\mathrm{Var}[\boldsymbol{a}^l]}$ and $\boldsymbol{\beta}^l = \mathbb{E}[\boldsymbol{a}^l]$.
With convolutional layers inputs are normalized channel-wise, i.e. mean and variance are computed over the batch and the whole two-dimensional image. When placing batch normalization directly after the convolution operation its bias (the b_g in equation 2.18) can be neglected. This is because the subtraction of the mean in equation 3.3 cancels out the effect of the bias. Instead, the parameter $\boldsymbol{\beta}^l$ works as an overall bias of the batch normalized layer.
Note that in our convolutional building block 3.2 we have placed the second batch normalization before the addition of the bypassed input. One could argue that addition

of the input shifts the distribution again and it would be beneficial to put batch normalization after the addition, directly before the second ReLU. However, experiments in [GW16] show that putting batch normalization after the addition yields a significantly worse performance.
Since our models already use ReLUs instead of sigmoid neurons the effect of the internal covariate shift is not very strong. Batch normalization is used anyway because it also has another effect. It works as a *regularizer* for the network. Neural networks severely tend to overfit training data. In order to prevent this, there exist a number of methods summarized under the term *regularization*. Thus batch normalization reduces overfitting of the model. In our case, it replaces another well known regularization technique called *dropout*.

3.1.4 Weight decay

Batch normalization is not the only regularizer that we have used in our network. *Weight decay* or *L2 regularization* is a completely different approach as it modifies the loss function. Therefore an extra term is added to the sum of single-sample losses (in this case over a batch of training data) so that the loss function looks as follows:

$$L_{\mathcal{B}}(h \mid \mathcal{D}_{train}, \theta) = \sum_{(\boldsymbol{x},y)\in\mathcal{B}} l(h(\boldsymbol{x}) \mid y, \theta) + \frac{\lambda}{2}\sum_{\boldsymbol{w}} \boldsymbol{w}^2 \tag{3.6}$$

[Nie15]. The second summand is the sum over all squared weights, where λ is called *weight decay factor*[4]. Note that the $\boldsymbol{w}$ are only weights, not biases. We do not impose weight decay on the biases since it has shown that this can lead the model to underfit [GBC16].
The intuition behind L2 regularization is that it penalizes large weights during training. It makes sure that only important weights are allowed to grow, while all other weights are pushed towards zero. This simplifies the network's learned function since it depends on less parameters, because many of them have weights close to zero. Evidently, simpler functions are more general and hence are less likely to overfit [Nie15].
In our implementation we use a weight decay factor of 10^{-4}, according to [He+15a]. But unlike He *et al.* who only use weight decay for the weights of convolutional layers we also apply it to fully-connected and batch normalization layers, following the practice in [GW16].

3.1.5 Parameter initialization

There is one last aspect of the network implementation we have to take a look at. Up to now we have never considered the initialization of parameters, i.e. weights and biases of the single layers in our network, although it seems rather intuitive that a good parameter initialization can boost convergence to a critical extent.

[4] L2 regularization is derived by imposing a Gaussian prior on the weights $\boldsymbol{w}$. Then λ is the variance of this prior. To keep this identity one introduces the division by 2.

A naïve way to initialize weights and biases would be to fill them with ones and zeros respectively. But there exist better ways, for example the *Xavier* initialization [GB10] that has been very popular in recent research. However, for the extremely deep models presented in [He+15a] this initialization did still not lead to convergence. So He *et al.* referred to their own paper [He+15b] where they present an even more sophisticated weight initialization for convolutional layers.

In this paper they initialize weights normally distributed with zero mean. For the standard deviation of the distribution they derive an initialization considering the properties of rectified linear unit activations. The full derivation with all its assumptions would go beyond the scope of this thesis, so we are going to give a short overview.
Assume we have the l^{th} convolutional layer with a $k \times k$ kernel and d output feature maps. Recall the definition of the error in a layer from section 2.2.1 as $\delta^l = \frac{\partial L}{\partial \boldsymbol{a}^l}$, where L is the loss function. He *et al.* state that one can compute the variance of the error in layer l depending on the variance of the error in layer $l+1$ as

$$\text{Var}[\delta^l] = \frac{1}{2}\hat{n}^l \text{Var}[\boldsymbol{w}^l]\text{Var}[\delta^{l+1}], \tag{3.7}$$

given that the activation of layer l is a rectified linear function. $\boldsymbol{w}^l$ is the (three-dimensional) weight tensor of the convolution and $\hat{n}^l = k^2 d$. Then they compute the variance of the first weighted layer (which they call layer 2 since the first layer is just the unweighted inputs), given that we have m layers in total, as

$$\text{Var}[\delta^2] = \text{Var}[\delta^{m+1}] \left(\prod_{l=2}^{m} \frac{1}{2}\hat{n}^l \text{Var}[\boldsymbol{w}^l] \right). \tag{3.8}$$

Now they aim to keep $\text{Var}[\delta^2]$ from growing or shrinking exponentially with the number of layers. This can be reached by setting

$$\frac{1}{2}\hat{n}^l \text{Var}[\boldsymbol{w}^l] = 1 \quad \forall l. \tag{3.9}$$

This is the equation our initialization should fulfill. So by rearranging equation 3.9 for initializing the standard deviation σ of the Gaussian distribution we get

$$\sigma = \sqrt{\frac{2}{\hat{n}^l}}. \tag{3.10}$$

Note that the initialization is only derived for convolutional layers. For the parameters of batch normalization we stick to [GW16] and initialize γ as ones and $\boldsymbol{\beta}$ as zeros. For the fully-connected layer we set biases to zero and use Xavier initialization for weights, also according to [GW16].

3.1.6 Residual network architecture in TensorFlow

Now that we have described all basic and advanced concepts of residual neural networks, we are going to take a look at some code excerpts showing the implementation of residual models in TensorFlow.

```
def inference(self, x, n):
    x = conv_layer(x, out_channels=16, ksize=3, relu=True, stride=1, name='conv1')
    x = add_k_conv3x3_blocks(x, 2 * n, in_channels=16, out_channels=16, name='conv2')
    x = add_k_conv3x3_blocks(x, 2 * n, in_channels=16, out_channels=32, name='conv3')
    x = add_k_conv3x3_blocks(x, 2 * n, in_channels=32, out_channels=64, name='conv4')
    x = pooling_layer(x, tf.nn.avg_pool, ksize=8, stride=1, name='avg_pool')
    x = fc_layer(tf.squeeze(x), out_channels=10, activation_fn=None, name='fc')

    return x
```

Listing 3.1: Implementation of residual models for CIFAR-10.

Listing 3.1 shows the general implementation of the residual models according to Table 3.1. Implementing a neural network in TensorFlow works just as one would expect: Given an input `x` one passes it into a layer and then assigns the result to `x` again to make the output of one layer the input of the next.
The functions `conv_layer` and `fc_layer` are wrappers for the TensorFlow built-ins `tf.nn.conv2d` and `tf.matmul`, respectively. Those wrappers take care of initializing weight and bias variables, correctly inserting parameters into the TensorFlow functions and, if necessary, applying an activation function. Similarly, `pooling_layer` is a wrapper for the specified pooling function, in this case `tf.nn.avg_pool`. Note that the fully-connected layer does not have the softmax activation (see section 3.3.1) in this implementation. That is because in TensorFlow softmax can be applied more efficiently in combination with the error function.

```
def conv3x3_block(x, in_channels, out_channels, name):
    if in_channels != out_channels:
        stride = 2
    else:
        stride = 1

    f = conv_layer(x, out_channels, ksize=3, relu=True, stride=stride, name=name + '_1')
    f = conv_layer(f, out_channels, ksize=3, relu=False, stride=1, name=name + '_2')

    if x.get_shape() != f.get_shape():
        x = identity_mapping(x, x.get_shape(), f.get_shape(), name)

    # this is the residual addition
    x += f
    return tf.nn.relu(x, name=name + '_ResidualReLU')
```

Listing 3.2: Construction of a residual building block.

The function `add_k_conv3x3_blocks` is another wrapper calling `conv3x3_block` k times with the correct parameters. The function `conv3x3_block` implemented in Listing 3.2 constructs a residual building block as depicted in Figure 3.2.
After setting up the two convolutional layers (batch normalization is also added by the `conv_layer` wrapper) there needs to be checked if convolutions changed the dimensions of the input. If so we need to apply dimension reduction as described in section 2.2.3 before performing the residual addition. Since we only use dimension reduction option A in CIFAR-10 models we always apply the function `identity_mapping`.

Implementing dimension reduction was challenging because at the time of implementation TensorFlow did not support pooling and convolution with a stride greater than the kernel size[5]. Therefore it was impossible to directly implement 1×1 pooling with stride 2 as described in section 2.2.3. To tackle this we made use of the following trick: Since 1×1 pooling with stride 2 always keeps the upper left pixel of a 2×2 area we split the pooling operation in two steps. First we apply a mask to the input that set every pixel but the upper left of a 2×2 area to zero. The result of that is then plugged into a 2×2 max-pooling with stride 2. To ensure correctness of that algorithm we need to assume that the upper left values are always non-negative. This assumption holds because the tensors where dimension reduction is applied to are always outputs of a ReLU which guarantees values greater or equal to zero.

Now that we have seen how a residual neural network is implemented in TensorFlow we are going to describe the framework needed to train and evaluate the models in the next sections.

3.2 Preprocessing of the CIFAR-10 dataset

Another important process in every machine learning task is the preprocessing of training and evaluation data, i.e. transforming data from real world representation into a machine readable format.

The first step here is always *feature selection*. That means simplifying data by aggregating or deleting measurements (that are often continuous) to obtain discrete *features* that a model can be trained on. However, in image recognition there is no need for active feature selection since digital images are already represented in a format models can easily be trained on.
The images of CIFAR-10 are encoded using the RGB format. That means the color of each image pixel is represented by the values of the three color channels *red*, *green* and *blue*. Additive color mixing tells us that one can obtain any color by adding red, green and blue with different intensities. In RGB those intensities are represented by an integer between 0 and 255. So we simply take the image pixels as our features, each of them having the three-stack of RGB color intensities as its value.

[5] `https://github.com/tensorflow/tensorflow/issues/889`

Further preprocessing for image recognition includes data *normalization* and *augmentation*. Normalization means transforming training or test data to have a normal distribution, i.e. subtracting the mean and dividing by the standard deviation of pixel values over the whole training or test set.
When describing normalization, He *et al.* leave it open whether to compute mean and standard deviation across all three color channels or separately for each channel. In this case we stick to [GW16] where mean and standard deviation are calculated for each channel on its own. The reason for applying normalization is, once again, faster convergence since neurons of the first layer do not have to adapt to a new stochastic distribution in each training step [IS15].

Data augmentation is another regularizer for our models. It is quite obvious that one can get a more general model and thus can avoid overfitting by training it with a greater training set. But obtaining more data is expensive. So one artifically enlarges the training set by applying some often randomized transformations to training data [Nie15].
In our experiments with CIFAR-10 we follow the data augmentation strategy from [Lee+14] that is also used in [He+15a]. It proposes the following steps: First there are four pixels of zeros padded on each side of an image. The resulting 40×40 pixels image is then horizontally flipped with a probability of 50%. Finally the padded image is randomly cropped back to 32×32 pixels.

For the implementation of preprocessing we use TensorFlow's queue framework that allows a complete decoupling of image preprocessing and network training. TensorFlow handles all the challenges coming with concurrency like read/write access control internally, so once again the programmer does not have to consider this.

```
# NUM_TRAINING_IMAGES = 50000, NUM_THREADS = 3, BATCH_SIZE = 128

def inputs(self):
    filename_queue = tf.train.string_input_producer(FILENAMES, name='filename_queue')

    image, label = read_image(filename_queue)
    image = preprocess_for_training(image)

    min_num_examples_in_queue = int(0.4 * NUM_TRAINING_IMAGES)
    image_batch, label_batch = tf.train.shuffle_batch(
        [image, label],
        shapes=[[32, 32, 3], []],
        batch_size=BATCH_SIZE,
        num_threads=NUM_THREADS,
        capacity=min_num_examples_in_queue + (NUM_THREADS + 2) * BATCH_SIZE,
        min_after_dequeue=min_num_examples_in_queue,
        name='example_queue'
    )
    return image_batch, label_batch
```

Listing 3.3: Data preprocessing routine using the queue framework.

Listing 3.3 shows our implementation training data preprocessing. The functions `tf.train.string_input_producer` and `tf.train.shuffle_batch` each define a queue. The first queue holds the filenames of the CIFAR-10 image files. A consuming thread can now obtain a single image as a three-dimensional tensor of shape `[height, width, channels]` and its corresponding label as a scalar tensor. Then it executes the preprocessing steps described above that are encapsulated in `preprocess_for_training` and pushes the results to the second queue.
This queue returns batches of images as a tensor of shape `[batch size, height, width, channels]` that can then be fed into a neural network. This queue created by `tf.train.shuffle_batch` has the property to shuffle its content so that not always the same examples appear together in a batch. It also introduces an upper (`capacity`) and a lower (`min_after_dequeue`) bound for the number of images to be held in the queue. This ensures good shuffling of the examples on one hand and limits the queue's memory usage on the other hand. The settings for those two parameters are recommendations from the TensorFlow documentation. The best value for the number of threads `NUM_THREADS` that perform the preprocessing operations was determined by experiment, the batch size of 128 is in line with [He+15a].

3.3 Training and evaluation

In this last section we are going to look at some aspects to make the neural network training procedure described in section 2.2.1 applicable in practice.

3.3.1 Cross-entropy and softmax

Cross-entropy loss is a special instance of the loss function L. It is defined as

$$L(h \mid \mathcal{D}_{train}, \theta) = -\sum_{(\boldsymbol{x},y)\in\mathcal{B}} (y \ln z + (1-y)\ln(1-z)) \text{ for any } \mathcal{B} \subset \mathcal{D}_{train}, \tag{3.11}$$

where y is the true label of instance $\boldsymbol{x}$ (either 0 or 1) and $z = h(\boldsymbol{x} \mid \theta)$ is the network's output value for the predicted class when feeding $\boldsymbol{x}$ into our model. Cross-entropy has a convenient property which is why it is the loss function of choice in deep learning research. Its derivative with respect to weights and biases is close to zero when there are few wrong predictions, but gets larger as the number of wrong predicitions increases. This means after having started training parameters tend towards an optimal value very fast, which boosts convergence speed [Nie15].
Note that cross-entropy only behaves like a loss function when the network's output values are between 0 and 1. This is where the previously mentioned *softmax* comes into play.
Softmax makes use of the fact that each sample $\boldsymbol{x}$ only has exactly one true label. It scales the outputs of the last network layer in a way that they form a probability distribution

and is defined as

$$z_j = \frac{e^{a_j}}{\sum_{k=1}^{n} e^{a_k}}, \tag{3.12}$$

where n is the number of classes (and hence the number of output neurons). The exponentiation of the weighted neuron outputs is there to get rid of possibly negative weights [Nie15]. The value to be plugged into cross-entropy loss is then $z = \max_j z_j$.

He *et al.* did not specify which loss function they used in [He+15a]. Although the use of softmax suggests another efficient loss function called *log-likelihood* (that considers the fact that the output of softmax is a probability distribution) [Nie15] we stuck to cross-entropy for two reasons. First because it is also used in the re-implementation in [GW16] and second because with `tf.nn.softmax_cross_entropy_with_logits` TensorFlow provides an efficient implementation of softmax in combination with cross-entropy.

3.3.2 The Adam optimization algorithm

Now that we have a loss function we need an algorithm minimizing it. Despite the previously introduced stochastic gradient descent algorithm working fine in theory, more sophisticated derivates of it have been developed that exhibit better properties in convergence speed and finding the global minimum.
[He+15a] and [GW16] both use a gradient descent variant called *Nesterov accelerated gradient*. Unfortunately TensorFlow does not provide an implementation of this algorithm, so we had to deviate from both publications here. After some experiments with three different optimizers (stochastic gradient descent with momentum, RMSProp and Adam optimizer) we chose to stick to the Adam optimization algorithm for the rest of the experiments because it yielded the best training results.

Adaptive moment estimation (Adam) was published by Kingma & Ba in 2015 [KB14]. They derived the algorithm by applying two modifications to the gradient descent update rule in equation 2.6. First they do not consider the plain gradient from iteration step t to update parameters but rather compose the gradient of a fraction of the gradient from the previous time step $t-1$ and a fraction of the current gradient. They define

$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1)\nabla L^{(t)}, \tag{3.13}$$

where for β_1 they suggest a value of 0.9. A computation as in equation 3.13 is often called *exponential moving average* and is widely used in machine learning. This gradient update is similar to the technique used in stochastic gradient descent with momentum and has shown to increase convergence speed and decrease the risk of getting stuck in local minima [Rud].
The second modification is inspired by another optimization algorithm called *Adadelta*. The idea behind Adadelta is to adjust the learning rate η depending on each parameter in

a way that more frequent parameters get smaller updates and less frequent parameters get larger updates. Therefore, Kingma & Ba track the moving average of squared gradients, i.e.

$$v^{(t)} = \beta_2 v^{(t-1)} + (1 - \beta_2)(\nabla L^{(t)})^2, \tag{3.14}$$

where β_2 is recommended to be 0.999. Then they set the new learning rate to $\frac{\eta}{\sqrt{v^{(t)}+\varepsilon}}$, where ε is a small correction term to avoid division by zero, e.g. 10^{-8}. In fact, $m^{(t)}$ and $v^{(t)}$ are estimates of the first and second moment (mean and uncentered variance) of the gradient, which gives the algorithm its name [Rud].
There is one challenge left resulting from the initialization of the moving averages $m^{(t)}$ and $v^{(t)}$. Since they are initialized as vectors of zeros they are biased towards zero, especially in the first few interation steps. Therefore Kingma & Ba introduce bias-corrected versions of $m^{(t)}$ and $v^{(t)}$ as

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1 - \beta_1^t}, \qquad \hat{v}^{(t)} = \frac{v^{(t)}}{1 - \beta_2^t}. \tag{3.15}$$

The effective parameter update rule of Adam then derives as

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\hat{v}^{(t)} + \varepsilon}} \hat{m}^{(t)} \tag{3.16}$$

[Rud].
In TensorFlow different optimization algorithms are all implemented as a subclass of `tf.train.Optimizer`. Performing a parameter update step is as easy as calling the `tf.train.Optimizer.minimize()` method.
For our implementation we chose `tf.train.AdamOptimizer` with the default parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$, but set $\varepsilon = 0.1$. TensorFlow documentation suggests this value as a good choice when training state-of-the-art image recognition networks. We experimentally verified that this indeed yields better training results.

3.3.3 Training and evaluation in practice

Before we are going to look at the implementation of the training and evaluation loops, we need to discuss the last remaining free parameters. These are the total number of training steps and the learning rate decay schedule.
Choosing the right learning rate is critical for training success. When it is chosen too large the fluctuations in the parameters can hamper convergence, when it is chosen too small training will take too long because parameters are updated in only tiny steps. It has also been found that when training starts to plateau at a certain loss value decaying the learning rate can help to boost learning again.
Unfortunately there are no rules defining when to decay the learning rate and which values to use, as well as there is no definitive setting on how many steps training should take in total. Those parameters strongly depend on model, optimizer and dataset used and have to be determined by experiment.

Since we are trying to reproduce the results from [He+15a] we adapt those parameters accordingly. They use a total number of 64k steps and an initial learning rate of 0.1. During training they divide the learning rate by 10 two times so that it becomes 0.01 after 32k steps and 0.001 after 48k steps. These might not be the best settings, though, but they allow us to directly compare our results to the ones from the paper.

```
# INITIAL_LEARNING_RATE = 0.1, TOTAL_TRAINING_STEPS = 64000,
# N could be e.g. 3 for ResNet-20

# graph setup
images, true_labels = inputs()
predictions = inference(images, N)
loss_op = loss(predictions, true_labels) # computes the sum of
# tf.softmax_cross_entropy_with_logits and the weight decay terms

lr = INITIAL_LEARNING_RATE
learning_rate = tf.placeholder(dtype=tf.float32, shape=[])
global_step = tf.Variable(0)
optimizer = tf.train.AdamOptimizer(learning_rate, beta1=0.9, beta2=0.999, epsilon=0.1)
train_op = optimizer.minimize(loss_op, global_step=global_step)

# graph execution
sess = tf.Session()
coord = tf.train.Coordinator()
sess.run(tf.initialize_all_variables())
threads = tf.train.start_queue_runners(sess=sess, coord=coord)

while not coord.should_stop():
    sess.run(train_op, feed_dict={learning_rate: lr}) # the actual training operation
    step = sess.run(global_step)
    if step == TOTAL_TRAINING_STEPS:
        coord.request_stop()
        break

    lr = update_lr(lr, step) # checks if the learning rate needs to be decayed

coord.join(threads)
sess.close()
```

Listing 3.4: Data flow graph setup and model training loop.

Listing 3.4 shows a simplified version of the training routine implemented in TensorFlow. As usual the code is split in two phases. In graph construction phase we first define image preprocessing by calling the `inputs` function from Listing 3.3 and the model using `inference` from Listing 3.1. The function `loss` computes the total loss as the sum of the result of `tf.softmax_cross_entropy_with_logits` and the L2 losses of each weight variable. It is left as a blackbox for brevity.
The next part of code defines learning rate and the optimizer. Since we want to compute learning rate decay outside the TensorFlow running graph we use a `tf.placeholder` to dynamically feed the current learning rate into the graph. By passing a variable

`global_step` to the optimizer's `minimize` function it keeps track of how many optimization steps there already have been performed.

In the next block of code the graph execution phase is started by opening a `tf.Session`. The `tf.train.Coordinator` is a construct that takes care of all threads populating input queues. After having initialized all parameter variables and having started the input queue threads the training loop begins. The code in the loop should be self-explanatory. The function `update_lr` is a pure Python function that compares the current step to our defined step limits and decays the learning rate if necessary.

The code in Listing 3.4 is simplified because we skipped the whole logging functionality. In the real trainings we log loss and training error computed over the current batch as well as timing information such as images per second and seconds per step to the console. Additionally every 100 steps we save a `tf.scalar_summary` of loss and training error as raw values and as an averaged version (using `tf.ExponentialMovingAverage`). Also every 100 steps we store a checkpoint of all training parameters. This checkpoint is then used in our evaluation routine to determine error on the test set at an arbitrary step in the training procedure.

```
# MAX_NUM_EXAMPLES = 1000, BATCH_SIZE = 128

global_step, ckpt = has_new_checkpoint(CHECKPOINT_PATH, last_global_step)
if ckpt:
    saver.restore(sess, os.path.join(CHECKPOINT_PATH, ckpt))
else:
    return global_step

try:
    num_iter = int(math.ceil((1.0 * MAX_NUM_EXAMPLES) / BATCH_SIZE))
    true_count = 0
    total_sample_count = num_iter * BATCH_SIZE
    step = 0
    while step < num_iter and not coord.should_stop():
        predictions = sess.run(eval_op) # the actual evaluation operation
        true_count += numpy.sum(predictions)
        step += 1

    accuracy = (true_count * 1.0) / total_sample_count
    test_error = 1 - accuracy

    summary = sess.run(summary_op, feed_dict={test_err: test_error})
    summary_writer.add_summary(summary, global_step)
except Exception as e:
    coord.request_stop(e)

return global_step # return global step of the current checkpoint to track progress
```

Listing 3.5: Evaluation procedure for a single training step.

The evaluation routine is designed so that it can be executed in parallel to training as well as after training has finished. If it is executed in parallel the code in Listing 3.5 is run every 100 seconds. Then first the function `has_new_checkpoint` checks if there has been a new checkpoint written. If not, evaluation returns without doing anything. Else a `tf.train.Saver` restores the model state at the detected checkpoint and the evaluation loop begins.
We chose to compute test error only over a subset of the whole test set. We found that a ratio of 10% (= 1000 images) of the test set yields a good enough estimate of the overall test error. Evidently, this has the advantage of reduced computational effort. The evaluation loop then executes `eval_op` for `num_iter` batches and accumulates the results. The function `eval_op` simply feeds a batch of images into the model with the restored parameters and compares its predictions to the true class labels. After having processed all `MAX_NUM_EXAMPLES` images the resulting test error is written to a scalar summary.

Having described how to build, train and evaluate residual neural networks in TensorFlow, in the next chapter we are going to look at the results of the real training run and compare them to the results from [He+15a].

4 Evaluation

In this chapter we will first describe the hardware setup and methodology used for training and evaluation of the residual networks. Then we are going to present the results from our experiments with the models from section 3.1.1 on the CIFAR-10 dataset and compare them to the results from [He+15a].

4.1 Hardware setup

Since due to the huge number of parameters training of neural networks is highly computationally intensive, the use of the right hardware is critical for training success. Although highly optimized numerical computation libraries like TensorFlow exist, some experiments can still be infeasible if the underlying hardware is not powerful enough. Evidently it holds that deeper networks and larger input will result in longer training duration.
Many operations in neural network training, e.g. multiplying the inputs of a layer with the corresponding weights, can be computed more efficiently using matrix arithmetic (which seems intuitive since we have always used matrix notation in our description of the mathematical background). So the means of choice to train neural networks are GPUs. Despite the fact that GPUs can speed up training to very large extents — we experienced a speedup of up to 400% compared to the CPUs used — they introduce another restriction that has to be taken care of.
The main bottleneck of a GPU is its memory. Apart from the computational intensity of neural networks they are also extremely memory consuming, especially at training time where in addition to the parameters, the gradient of former has to be stored, too. Thus the GPU memory has to be large enough to be able to hold the whole model with all gradients and additionally some preprocessed images in the input queue (which are neglegible in the case of CIFAR-10 compared to the model memory consumption). Analogous to training time memory usage increases with increasing depth and input size. See [Det14] for an impressive calculation of the memory usage of a fully-connected neural network.
To circumvent the memory restrictions of GPUs deeper neural networks are often trained on GPU clusters. Since those clusters are very costly it was infeasible to acquire one for this thesis. Making things even harder TensorFlow only supports the most recent GPUs, specifically those supporting CUDA version greater or equal to 3.0[1]. This made it impossible to draw profit from the GPU cluster service of the *Leibniz Rechenzentrum*

[1] `https://developer.nvidia.com/cuda-gpus`

(LRZ)[2]. On one hand it has too old GPUs and on the other hand it has an old LIBC version installed that TensorFlow is not able to work with. The LIBC problem also holds for the Linux CPU cluster[3] of the LRZ.

Considering these restrictions the hardware setup for the experiments was as follows: Early experiments needed for bugfixing and optimizing were all computed on a NVIDIA GeForce GTX960 installed in an old tower PC. For a few experiments also a GeForce GTX970 was used. Both machines were running on Ubuntu 15.10 Server Edition using the GPU version of TensorFlow 0.7.1. As soon as all bugs had been resolved further experiments on CIFAR-10 were run on the LRZ Compute Cloud[4]. The Compute Cloud is a service where one can set up virtual machines to run expensive computations on. Those VMs are only provisioned with a quad-core CPU and 16GB RAM and thus computations take much longer than on the GPUs, but their great advantage is that one can run arbitrarily many VMs at the same time. Thus we could conduct many experiments in parallel without additional time cost.

Due to that very basic hardware setup there were some restrictions on the extent of our experiments. On the CIFAR-10 dataset, He *et al.* experimented with networks up to 1202 layers (see Table 3.2). We could not look at the two largest models ResNet-110 and ResNet-1202 since they could not be fit into memory of one of the GPUs. We also decided not to run them on the VMs since ResNet-110 would already have taken 20 days to train which we considered to be infeasibly long. However, this is not a great downgrade to the validity of this thesis since the main conclusion of [He+15a], the absence of the degradation problem, could also be verified with the smaller models up to 56 layers.
He *et al.* also conducted experiments on the ILSVRC 2015 subset of the ImageNet dataset (over one million images of varying sizes in 1000 classes) [Rus+15]. For that large amount of data they introduced huge models with an input size of 224×224 pixels and up to 152 layers. The models were trained with a batch size of 256 for over 500k steps. Those settings require both huge memory and a lot of computational power so that reproducing the results of [He+15a] on ImageNet was not feasible for us. Nonetheless we are going to revisit a reduced version of one of the ImageNet models in Chapter 5.

4.2 Experiments and results on CIFAR-10

After we had finished implementing the CIFAR-10 ResNet models and a training and evaluation framework we proceeded as follows: First we ran a long series of trainings

[2]The Leibniz Rechenzentrum provides IT services for students and researchers of the universities in Munich. Their offers include VPN and e-mail services as well as hardware resources for long and expensive computations.
[3]`https://www.lrz.de/services/compute/linux-cluster/`
[4]`https://www.lrz.de/services/compute/cloud_en/`

on the smallest model ResNet-20. Evidently the reason for only considering ResNet-20 was that its training duration is the shortest of all networks.

Those runs were needed mainly for discovering and fixing bugs in the implementation of the models. Bugs usually expose themselves as bad convergence behaviour in the beginning of training (convergence does not immediately start or not at all) and thus final test error being worse than the resulting error of [He+15a].

During those runs we also tried different optimizers (inspired by [GW16]) and varying training durations (80k and 100k steps) with different learning rate schedules. We always made the reasonable assumption that parameter settings are independent of model depth and independent of each other, i.e. when a setting yields a better overall performance the setting will still make an improvement when other parameters are changed. See section 3.3.3 for the final parameter settings we arrived at.

As soon as we reached a test error comparable to [He+15a] on ResNet-20 we started trainings of ResNet-32, ResNet-44 and ResNet-56 in parallel on Compute Cloud VMs. Unfortunately the experiment on ResNet-56 crashed during training, so we had to restart it on the GTX960 GPU. Where the experiments have run plays a role when comparing training times, as we do in Tables 4.1 and 4.2.

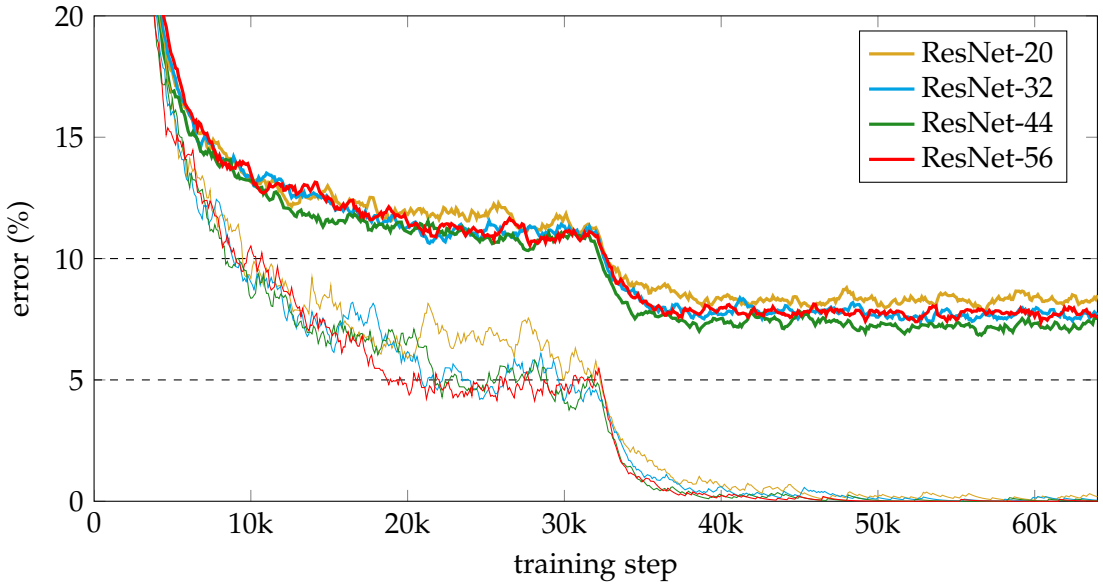


Figure 4.1: Training and evaluation results of residual neural networks on CIFAR-10. *Thin*: Training error. *Bold*: Test error.

The first observation we made is that results of our experiments can fluctuate very strongly depending on the random initialization of weights in the beginning. We observed a variance of about one percent in test error of the same model over multiple training runs. More stable results could be obtained by running each experiment multiple times and averaging over test error or always picking the best run.

See Figure 4.1 for our reproduction of Figure 6 in [He+15a]. It shows test error (bold) and training error (thin) of all trained models in percent over the 64k training steps. For better resolution of important areas in later training the error-axis is cut at 20%.
From the chart we can immediately see that test error as well as training error decrease with growing model depth. Hence we have verified the most important finding of [He+15a], the absence of the degradation problem.
A fact to notice is the test error curve for ResNet-56 lying above the curve for ResNet-44 and behaving similar to ResNet-32. This differs from the results in [He+15a] where the test error keeps falling with increasing network depth up to 110 layers. However, we do not account this to degradation since the training error is not worse than for ResNet-44. We rather blame it to the fluctuations described above or to overfitting.

model	training time	test error	test error in [He+15a]	test error in [GW16]
ResNet-20	16h*	8.28%	8.75%	8.29%
ResNet-32	5d 18h	8.03%	7.51%	7.63%
ResNet-44	8d	**7.13**%	7.17%	7.14%
ResNet-56	1d 14h*	7.40%	**6.97**%	**6.94**%

Table 4.1: Training time and test error of CIFAR-10 models after 64k training steps with comparison to test error from [He+15a] and [GW16]. *These models were trained on the GTX960 GPU, hence the significantly shorter training duration.

We also computed test error over the whole test set on each model after training has finished, i.e. after 64k training steps. Table 4.1 shows the results that confirm the outcomes from Figure 4.1. Test error keeps decreasing with growing model depth until ResNet-44 while ResNet-56 performs worse again.
Compared to [He+15a] and [GW16] our ResNet-20 and ResNet-44 implementations perform marginally better than the respective models from the reference publications while we achieved worse results with ResNet-32 and ResNet-56.

To sum things up our results on CIFAR-10 are, apart from some fluctuations, in the range of results from [He+15a] and [GW16]. Therefore we have validated our implementation of residual neural networks. We could also verify the absence of the degradation problem when using residual models.
Apart from random noise the deviation of our models from the results of the reference publications could also be attributable to slight differences in implementation. For example we use the Adam optimizer in training while both other implementations train using Nesterov accelerated gradient.

We also made experiments to compare dimension reduction strategies described in section 2.2.3. In [He+15a] He *et al.* only tried different dimension reduction options with their ImageNet models while on CIFAR-10 they fixed dimension reduction to option A.

For ImageNet they observed a slight performance improvement due to the additional residual learning parameters introduced with option B. This is what we try to reproduce for CIFAR-10.

model	training time	test error
ResNet-20, Opt. A	16h*	8.28%
ResNet-20, Opt. B	3d 13h	8.67%
ResNet-56, Opt. A	1d 14h*	**7.40**%
ResNet-56, Opt. B	1d 14h*	8.00%

Table 4.2: Test error on ResNet-20 and ResNet-56 with dimension reduction options A and B. *These models were trained on the GTX960 GPU, hence the significantly shorter training duration.

We conducted two training runs with option B on the shallowest and the deepest CIFAR-10 model, namely ResNet-20 and ResNet-56, even though the number of dimension reductions does not change with increasing depth for CIFAR-10 models (see Table 3.1). We trained both models anyway to find out if model depth has an impact on the strength of the effect.
We did not observe any problems in convergence even with the additional parameters. This is in line with the observation in [He+15a]. However the final scores collected in Table 4.2 show contrary results to [He+15a].
On both models the variant with dimension reduction option B yields a test error that is worse compared to option A to a non-neglegible extent. Test error of ResNet-56 still lies below the error of ResNet-20, thus also in this case there is no evidence of degradation. But the increase in test error comparing option B to option A grows with deeper models, from 0.39 percentage points to 0.6 percentage points. All in all the differences between option A and option B and the associated deviations from the findings in [He+15a] are so large that we consider them unlikely to be caused by random noise in the results.
Instead a reason for dimension reduction option B having no (or even the inverse) effect could be that for CIFAR-10 the number of additional parameters is simply too low. By halving image border size twice and hence also doubling the number of feature maps twice (from 16 to 32 and from 32 to 64) with 3×3 convolutions we only have $3 \times 3 \times 16 \times 32 + 3 \times 3 \times 32 \times 64 \approx 20$k additional parameters, while for the smallest ImageNet model we have $3 \times 3 \times 64 \times 128 + 3 \times 3 \times 128 \times 256 + 3 \times 3 \times 256 \times 512 \approx 1.5 \cdot 10^6$ (see Chapter 5). Another reason could be that by increasing the parameter number, 64k steps are not enough for the training to fully converge. So maybe by raising the step number and appropriately adapting the learning rate schedule one could also obtain improved results with option B on CIFAR-10.

Now that we have extensively evaluated behaviour of residual models on the standard CIFAR-10 dataset, we are going to apply residual learning to a considerably more complex dataset in the next chapter.

5 Real World Application: The Yelp Challenge

In this section we are going to present an application of deep residual learning to real world data. The problem we will address is the *Yelp Restaurant Photo Classification* challenge presented on the data science competition platform *Kaggle* [Yel15].
We are first going to describe what the challenge is about, then look at the changes made to the residual models presented in Chapter 3 to be suitable for the Yelp dataset and finally present our experimental results with the altered models.

5.1 The Yelp restaurant photo classification challenge

Yelp, Inc. is a US-american company driving the website *Yelp*[1] that provides recommendation and rating services for local restaurants and other businesses. When searching for restaurants Yelp provides the ability to filter search results based on several labels. These labels are currently assigned by human raters creating a review of a restaurant. On 21 December 2015 Yelp started a competition on Kaggle challenging participants to do the following: Build a model that can assign the correct labels to a restaurant based on photos that users might have uploaded alongside their review.

Yelp provides training data of 2k businesses that are labelled with arbitrarily many (including zero) labels out of the label set listed in Table 5.1. Each business is associated with one or more photos, resulting in a total number of roughly 230k images. The test dataset is another 230k photos describing 10k businesses but for which the true labels are kept secret. Given this, the challenge is to build a model that can assign the 9 labels from Table 5.1 to the test images with an accuracy as high as possible. The performance of a model is measured by uploading a classification result file to the Kaggle online judge which has access to the labels for test images.

Before mapping label predictions from photos to businesses we will first concentrate on correctly classifying single images as we did in previous tasks of this thesis. However, there is one major difference between this challenge and e.g. image classification on CIFAR-10. While for CIFAR-10 each image had exactly one correct label here we are facing a *multi-label classification* problem, i.e. an image can have zero or many true labels. Formally, for the multi-label case the previously given definition of data in a

[1] `http://yelp.com`

label ID	description
0	good_for_lunch
1	good_for_dinner
2	takes_reservations
3	outdoor_seating
4	restaurant_is_expensive
5	has_alcohol
6	has_table_service
7	ambience_is_classy
8	good_for_kids

Table 5.1: Labels for the Yelp restaurant photo classification challenge.

(a) Labels: 5 6 8

(b) Labels: 1 2 4 5 6

(c) Labels: 0 2 6 8

Figure 5.1: Some images from the Yelp training dataset.

classification problem (see equation 2.1) changes to

$$\mathcal{D} = \{(\boldsymbol{x}_1, Y_1), (\boldsymbol{x}_2, Y_2), \ldots, (\boldsymbol{x}_n, Y_n)\} \subseteq X \times \mathcal{P}(Y), \tag{5.1}$$

i.e. an instance $\boldsymbol{x}_i$ can be associated with an arbitrary subset Y_i of possible labels Y.

5.2 Changes to the training environment

Due to this multi-label nature and other properties of the dataset we need to change the models as well as the preprocessing, training and evaluation strategy described in Chapter 3. In many points we took inspiration from the ImageNet models and the training framework proposed in [He+15a]. However, in some aspects we had to cut model complexity (and hence expected classification accuracy) in favor of computational feasibility on the available hardware.

5.2.1 Architecture modifications

An initial test on classifying Yelp images with CIFAR-10 ResNet-20 failed with no classification performance improvement at all over the whole training duration. We concluded that an input size of 32×32 pixels is much too small for the high complexity

layer name	output size	network component
conv1	112×112	7×7, 64, stride 2
	56×56	3×3 max-pool, stride 2
conv2_n	56×56	$\begin{bmatrix} 3 \times 3, & 64 \\ 3 \times 3, & 64 \end{bmatrix} \times 2$
conv3_n	28×28	$\begin{bmatrix} 3 \times 3, & 128 \\ 3 \times 3, & 128 \end{bmatrix} \times 2$
conv4_n	14×14	$\begin{bmatrix} 3 \times 3, & 256 \\ 3 \times 3, & 256 \end{bmatrix} \times 2$
conv5_n	7×7	$\begin{bmatrix} 3 \times 3, & 512 \\ 3 \times 3, & 512 \end{bmatrix} \times 2$
	1×1	avg-pool, fc 9, sigmoid

Table 5.3: The architecture of ResNet-18. Table adapted from [He+15a].

of the Yelp dataset. Since the Yelp photos have sizes similar to ImageNet images we chose to use one of the ImageNet residual models from [He+15a].
Due to our hardware restrictions we were only able to pick the smallest ImageNet model, namely ResNet-18. It takes an input of 224×224 pixels which is first downsized to 112×112 pixels by a 7×7 convolution with stride 2 on 64 feature maps. After a 3×3 max-pooling with stride 2 the resulting $56 \times 56 \times 64$ pixels input is passed through a series of the previously introduced residual building blocks from Figure 3.2. Again the network ends with global average-pooling and a 9-way fully-connected layer. The detailled architecture of ResNet-18 is described in Table 5.3.
Since we face a multi-label classification problem with Yelp data, we need to remove softmax activation after the last layer. We replace it by a sigmoid to scale each of the 9 outputs coming from the fully-connected layer to the range between 0 and 1. Instead of taking the maximum of network output values to decide on the single label assigned to an image we need another way to pick the labels being predicted for a photo. We introduce a threshold on network outputs so that if an output's value is above this threshold the corresponding label will be assigned to the image. For sigmoids we choose the most evident threshold of 0.5.

We also made some minor changes to the architecture of residual building blocks. Because we are aiming for the best possible classification performance but have neither time nor hardware resources to make experiments on the huge ImageNet models we rely on results from other publications. We assume that if they improved performance on the problem the publication tackled this will also hold for our concrete task, the Yelp challenge.
Therefore we omitted the ReLU at the end of a residual building block because in [GW16] this reduces test error on CIFAR-10 by a small amount. We also use dimension reduction option B again since we believe that on ImageNet models this will improve

classification performance, like it did in [He+15a].

5.2.2 Advanced data preprocessing

Yelp images all have their longer sides scaled to 500 pixels, but come with varying aspect ratios. This is different to CIFAR-10 where every image has a square size of 32×32 pixels. To make Yelp photos ready for the 224×224 network input we use the data preprocessing strategy proposed for ImageNet in [He+15a].
First, we randomly resize the image so that its shorter side has a length in the range of 256 to 480 (inclusive). We scale images so that the aspect ratio is kept. Then the image is horizontally flipped with a probability of 50%. After that we take a random crop of 224×224 pixels which is then fed into the model. Note that this random scaling and cropping is also a kind of data augmentation.

Also according to [He+15a] after normalizing the input we use another data augmentation strategy. This method, introduced in [KSH12], alters the colors of an image based on *principal component analysis* (PCA) on RGB pixel values. For that we compute the 3×3 covariance matrix of RGB pixel values alongside the calculation of mean and standard deviation (which we need for normalization). Since the training dataset is so huge we only consider part of the whole training set for these computations. We have found that using only 1000 images yields a good enough estimate of those values.
Then the PCA-based color augmentation approach does the following: To each image pixel $I_{xy} = \left(I^R_{xy}, I^G_{xy}, I^B_{xy}\right)^T$ it adds the value

$$(\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3)\,(\alpha_1\lambda_1, \alpha_2\lambda_2, \alpha_3\lambda_3)^T, \tag{5.2}$$

where the $\boldsymbol{p}_i$ are the eigenvectors and the λ_i are the eigenvalues of the covariance matrix. $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3)$ is a random variable drawn from a Gaussian distribution with zero mean and standard deviation 0.1 [KSH12].
In order for the color augmentation to have the proper effect RGB pixels need to be represented differently. For CIFAR-10 pixels had the value of their color intensity, which lies between 0 and 256. Now we encode color values relative to their maximum value. This means we divide each pixel value by 256 before applying any preprocessing. This scales color values to a range between 0 and 1.

5.2.3 Changes in training and evaluation

The changes in training and evaluation are the only modifications that were really necessary to fit the proposed task and were not just for performance improvement. So they had already been implemented when we ran the initial experiments with CIFAR-10 ResNet-20.
Due to the multi-label nature of the Yelp challenge cross-entropy loss with softmax does not work properly anymore. But that does not exclude cross-entropy as a loss

function. Recall that the only requirement for cross-entropy to work as a loss function is that the outputs of the network range between 0 and 1. As mentioned before, we can achieve this by applying a sigmoid to each of the 9 output neurons. This makes the changes necessary to the loss function fairly minor: In our code we only need to replace `tf.softmax_cross_entropy_with_logits` by `tf.sigmoid_cross_entropy_with_logits`.

The terms of "training error" and "test error" are also not suitable for multi-label classification. When assigning arbitrarily many labels there are not only "right" and "wrong" predictions, but also cases in between where some labels are correctly predicted and some are not. Therefore we need another evaluation metric. We choose the so called *hamming loss* that measures classification performance via the hamming distance of predictions and true labels. Formally hamming loss is computed as

$$\text{hloss}(h \mid \mathcal{D}, \theta) = \frac{1}{|X|} \sum_{(\boldsymbol{x}_i, Y_i) \in \mathcal{D}} \frac{1}{|Y|} |h(\boldsymbol{x}_i \mid \theta) \triangle Y_i|, \tag{5.3}$$

where $\triangle$ is the symmetric difference between two sets [ZZ06]. A hamming loss of 0.5 refers to random guessing while zero loss means perfect performance. Note that hamming loss is not a loss function suitable for optimizing a model with gradient descent since it is not differentiable. For performance reasons we stick to computing training hamming loss over the current batch and test hamming loss over 10% of the test set.
Regarding test data there is another challenge imposed by the Yelp dataset. Since labels of the 200k test photos are kept secret we need to use another dataset for evaluation. To obtain test data with known labels we split the training set into *training* and *validation data*. This is a common technique in machine learning. A usual dividing point is a two-third/one-third split [Sma], so we train with roughly 160k of the 230k photos and evaluate the model on the remaining 70k images.

The last modification concerns our training hyper-parameters, namely number of steps, learning rate schedule and batch size. Since we do not have the resources to experiment with these settings we also refer to the values from ImageNet training in [He+15a]. They suggest training for 600k steps with a batch size of 256 and a dynamic learning rate schedule with an initial setting of 0.1.
Dynamic learning rate schedule means not decreasing learning rate before training error (or training hamming loss in this case) has stopped improving. To implement this we compute training hamming loss over the whole training set every 10k steps and check if it still decreases. If not we divide the learning rate by 10. We also add a possibility to manually change learning rate in case the dynamic decay will take too much time.
To have a large safety margin we set the maximum number of steps to two million and abort training when we see loss stop decreasing. A batch size of 256 is way too large for our hardware and time resources. Since a smaller batch size should not reduce classification performance to a very large extent (batch normalization layers are the only layers affected) we take a deep cut here and lower the batch size to 16. Due to the large

number of steps we only wrote checkpoints and summaries every 1000 steps. These modifications enable us to conduct the run described in the next section in a reasonable amount of time.

5.3 The experiment and its results on Yelp

Since we use an ImageNet model and a dataset with similar complexity to ImageNet we expected training to behave similar to the ImageNet runs from [He+15a]. When after 250k steps the dynamic learning rate decay had not been triggered we decided to manually lower the learning rate so that training would not take inappropriately long. We decayed learning rate again at 530k steps. After 900k steps and a total training time of 10 days and 21 hours we decided to stop training.

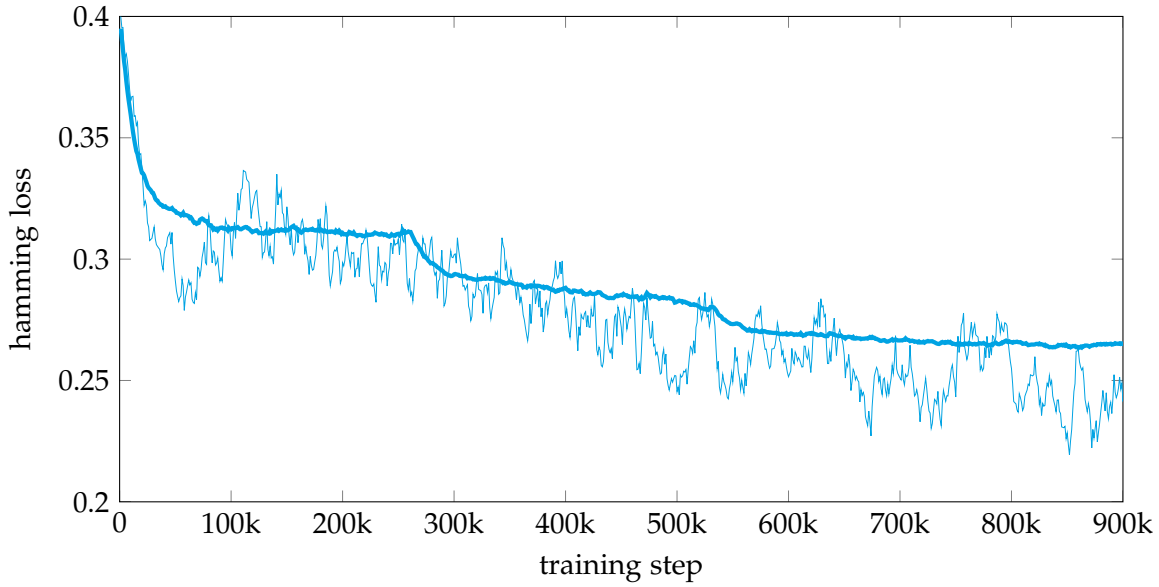


Figure 5.2: Training and evaluation result of ResNet-18 on CIFAR-10. *Thin*: Training hamming loss. *Bold*: Test hamming loss.

Figure 5.2 shows the results of this training run, plotted as hamming loss over training steps. At first sight the plot has a shape similar to the plots for CIFAR-10 in Figure 4.1. The are two facts notable on this chart in comparison to CIFAR-10. Firstly, the training hamming loss curve is very noisy. This evidently originates from the small batch size of 16 over which we computed training loss. Secondly, training and test hamming loss curves are very close together while on CIFAR-10 training error decreases to nearly zero. This is caused by the larger training set on one hand, and by the random cropping and color noise in image preprocessing on the other hand. Due to these random distortions

the model cannot fit the training set as precisely as on CIFAR-10 where images are distorted to a much smaller extent.

The most important difference to CIFAR-10, however, is that the lower bound of the loss axis lies at 0.2 and test hamming loss stagnates at roughly 0.27. This is also the fact that makes the result of this training run unsatisfactory because it means that our model only correctly predicts about three quarters of all labels (note that with random guessing one gets 50% correct). This result is confirmed by running evaluation over the whole validation set which also yields a hamming loss of 0.27.

Despite these bad results we also tried predicting labels for businesses. To achieve this we first feed every single image into the model and get its unscaled predictions, i.e. the model's score before sigmoid layer and thresholding. Then for each business we collect all images associated to it and average over the corresponding scores. This average is then put into sigmoid and thresholding to obtain the final predictions. For predicting business labels we obtain a slightly better hamming loss of 0.20.
We did the same for test data with unknown labels and submitted it to the Kaggle online judge. This judge uses a different evaluation metric, the *Mean F1-Score*. This score is computed as

$$\text{f1score} = 2\frac{pr}{p+r}, \tag{5.4}$$

where p is called *precision* and r is called *recall*. Precision and recall are terms from information retrieval, where

$$p = \frac{\text{true positives}}{\text{all predicted positives}} \quad \text{and} \quad r = \frac{\text{true positives}}{\text{all actual positives}}. \tag{5.5}$$

A good classifier maximizes both precision and recall, thus Mean F1-Score gets better the higher precision and recall are. This time a larger score means better performance. A score of 1 means perfect classification, while 0.5 stands for random guessing.
With this metric we achieve a score of 0.75. For comparison, Kaggle's the random guessing benchmark got a score of 0.45, the winning entry to the competition achieved 0.83.

We are sure that these results could be improved by tweaking training parameters. By simply training for a longer time and waiting until dynamic decay is triggered the final classification loss could be lowered. An increase of the learning rate could speed up training. Also more sophisticated loss functions for training neural networks on multi-label classfication problems exist, e.g. *BP-MLL* described in [ZZ06] and [GMW08]. Finally one could experiment with different image preprocessing strategies like in [Sze+15] or simply deeper models.

Although the results of this single run were not satisfactory we have shown that ResNet-18 can converge on this specific problem. Thus we have shown that residual neural networks are also applicable to real world multi-label classification problems on complex datasets.

6 Conclusion

In this thesis we implemented and evaluated the deep residual learning approach to image recognition presented by He *et al.* in [He+15a].

We first presented the conceptual basics of fully-connected, convolutional and residual neural networks and gave an introduction to TensorFlow. Then we applied these concepts to the concrete example of CIFAR-10 residual models from [He+15a]. In addition, we introduced and applied some advanced techniques of deep learning. We showed sample code for the implementation of models, data preprocessing, training and evaluation.
Then we conducted some experiments to validate our implementation. With the exception of ResNet-56, we were able to achieve similar performance in terms of test error compared to our reference publications [He+15a] and [GW16]. We were able to show the absence of the degradation problem that was the main finding of [He+15a].
On CIFAR-10 we could not reproduce the effect of dimension reduction via projection mapping (option B) reducing test error which He *et al.* had observed for their ImageNet models.
In the last chapter we applied residual learning to a real world image classification problem, the Yelp Restaurant Photo Classification challenge. We first described the changes we had to make to our implementation due to the higher complexity of the Yelp dataset and the multi-label nature of the problem.
We then evaluated the single training run we conducted on the Yelp challenge. We concluded that residual learning is applicable to this task since the model converged yet the results were not satisfactory in terms of classification performance. Optimizing model training to achieve better performance is left for further work.

Another starting point for further experiments would be to acquire better hardware and try to reproduce the results from [He+15a] with the deeper CIFAR-10 models ResNet-110 and ResNet-1202 as well as with the ImageNet models. One could also conduct experiments with more different optimizers as in [GW16]. This is particularly interesting in our case since TensorFlow does not provide an implementation of the optimizer used in the reference publications and thus one could try to find the algorithm that comes closest to the behaviour of the suggested optimizer.

During the making of this thesis He *et al.* published a follow-up paper on deep residual learning [He+16] where they investigate on the different dimension reduction strategies and their effect as well as on some architecture modifications inspired by [GW16]. As a next step one could try to reproduce the results from this paper, too. Other researchers

also started to work with deep residual networks [SIV16] which could also be a point to take up. Or one could go as far as totally deviating from the suggested architectures in [He+15a] and design own residual model architectures.

To sum things up image recognition with deep neural networks is still in important field in research and experience has shown that it surely will not take long until the next breakthrough in classification performance will be achieved. Since different researchers have started working with deep residual networks it is quite likely that they will play an important role in these advances.

List of Figures

List of Tables

Bibliography

[Aba+15] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[Alt15] M. Althoff. "Grundlagen der künstlichen Intelligenz - Learning." Lecture notes on the lecture "Grundlagen der künstlichen Intelligenz" at TUM in winter term 2014/2015. Jan. 2015.

[Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 0387310738.

[BS14] M. Bianchini and F. Scarselli. "On the complexity of neural network classifiers: A comparison between shallow and deep architectures." In: *Neural Networks and Learning Systems, IEEE Transactions on* 25.8 (2014), pp. 1553–1565.

[Det14] T. Dettmers. *Which GPU(s) to Get for Deep Learning: My Experience and Advice for Using GPUs in Deep Learning*. Aug. 2014. URL: http://timdettmers.com/2014/08/14/which-gpu-for-deep-learning/ (visited on 04/17/2016).

[GB10] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *International conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[GBB11] X. Glorot, A. Bordes, and Y. Bengio. "Deep sparse rectifier neural networks." In: *International Conference on Artificial Intelligence and Statistics*. 2011, pp. 315–323.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. "Deep Learning." Book in preparation for MIT Press. 2016.

[GMW08] R. Grodzicki, J. Mańdziuk, and L. Wang. "Improved multilabel classification with neural networks." In: *Parallel Problem Solving from Nature–PPSN X*. Springer, 2008, pp. 409–416.

[GW16] S. Gross and M. Wilber. *Training and investigating Residual Nets*. Feb. 2016. URL: http://torch.ch/blog/2016/02/04/resnets.html (visited on 04/01/2016).

[He+15a] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition." In: *CoRR* abs/1512.03385 (2015).

[He+15b] K. He, X. Zhang, S. Ren, and J. Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 1026–1034.

[He+16] K. He, X. Zhang, S. Ren, and J. Sun. "Identity Mappings in Deep Residual Networks." In: *arXiv preprint arXiv:1603.05027* (2016).

[IS15] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *CoRR* abs/1502.03167 (2015).

[KB14] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization." In: *CoRR* abs/1412.6980 (2014).

[Kri09] A. Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. 2009.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[Lea14] E. G. Learned-Miller. *Introduction to Supervised Learning*. Feb. 2014. URL: `https://people.cs.umass.edu/~elm/Teaching/Docs/supervised2014a.pdf` (visited on 03/17/2016).

[Lee+14] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu. "Deeply-supervised nets." In: *arXiv preprint arXiv:1409.5185* (2014).

[MHN13] A. L. Maas, A. Y. Hannun, and A. Y. Ng. "Rectifier nonlinearities improve neural network acoustic models." In: *Proc. ICML*. Vol. 30. 2013, p. 1.

[NH10] V. Nair and G. E. Hinton. "Rectified linear units improve restricted boltzmann machines." In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010, pp. 807–814.

[Nie15] M. A. Nielsen. *Neural networks and Deep Learning*. Determination Press, 2015.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors." In: *Nature* 1986/10/09/print (1986).

[Ros62] F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.

[Rud] S. Ruder. *An overview of gradient descent optimization algorithms*. URL: `http://sebastianruder.com/optimizing-gradient-descent/` (visited on 04/15/2016).

[Rus+15] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge." In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[Sed14] S. Sedhain. *What is special about rectifier neural units used in NN learning?* Mar. 2014. URL: `https://www.quora.com/What-is-special-about-rectifier-neural-units-used-in-NN-learning` (visited on 04/02/2016).

[SGS15] R. K. Srivastava, K. Greff, and J. Schmidhuber. "Highway Networks." In: *CoRR* abs/1505.00387 (2015).

[SIV16] C. Szegedy, S. Ioffe, and V. Vanhoucke. "Inception-v4, inception-resnet and the impact of residual connections on learning." In: *arXiv preprint arXiv:1602.07261* (2016).

[Sma] P. van der Smagt. "Decision Trees." Lecture notes on the lecture "Machine learning" at TUM in winter term 2015/2016.

[SZ14] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: *CoRR* abs/1409.1556 (2014).

[Sze+15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. "Going deeper with convolutions." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9.

[Yel15] Yelp, Inc. *Yelp Restaurant Photo Classification*. Dec. 21, 2015. URL: `https://www.kaggle.com/c/yelp-restaurant-photo-classification` (visited on 05/01/2015).

[Zei+13] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, et al. "On rectified linear units for speech processing." In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) 2013*. IEEE. 2013, pp. 3517–3521.

[ZZ06] M.-L. Zhang and Z.-H. Zhou. "Multi-Label Neural Networks with Applications to Functional Genomics and Text Categorization." In: *Knowledge and Data Engineering, IEEE Transactions on* 18.10 (2006), pp. 1338–1351.