Performance of Residual Networks on a Multi-Label Classification Problem

Bachelorarbeit
am Lehrstuhl für Mathematik mit Schwerpunkt Digitale Bildverarbeitung
der Fakultät für Informatik und Mathematik
an der Universität Passau

vorgelegt von
Christoph Sonntag
Erstgutachter: Prof. Dr. Tomas Sauer

Christoph Sonntag
Matrikelnummer: 77415

Abstract

Training neural networks becomes more difficult when more than just a few layers are used. It can even be shown that these deep neural networks do not only perform any better but actually do worse compared to networks with less layers. After an overview of classical and convolutional neural networks is given, a solution to allow more layers by learning a residual function is presented. Through this approach, it was possible to train a network with 34 layers that achieves better results than a shallower version with 18 layers on a dataset with multiple classes. The trained network assigns road traffic images from the view of a car's dashboard to one of nine different categories. Subsequently, experiments mapping more than one category to a single image are also presented and evaluated.

Kurzzusammenfassung

Das Training neuronaler Netze wird umso schwieriger, je mehr Schichten *(Layer)* beim Aufbau des Netzes verwendet werden. Beim Vergleich von diesen tieferen Netzen (Deep Neural Networks) mit Ausführungen, die weniger Layer verwenden, kann sogar festgestellt werden, dass sie nicht nur nicht besser, sondern meistens sogar schlechter abschneiden. Nach einem Überblick über herkömmliche Neuronale Netze und *Convolutional Neural Networks*, wird eine Lösung vorgestellt, die es durch das Lernen einer Restwert-Funktion *(Residual Function)* erlaubt eine größere Anzahl an Layern zu verwenden. Durch diesen Ansatz war es möglich – auf Basis eines Datensatzes mit mehreren Zielklassen – ein Netz mit 34 Layern zu trainieren, das bessere Ergebnisse erzielt als eine flachere Version mit lediglich 18 Layern. Das trainierte Netz ordnet Bilder aus dem Straßenverkehr, gefilmt aus der Sicht des Armaturenbretts eines Autos, einer von neun verschiedenen Kategorien zu. Im Anschluss daran werden Experimente vorgestellt und ausgewertet, bei denen einem Bild statt einer sogar mehrere Klassen zugeordnet werden.

Contents

Chapter 1

Motivation

Theoretically, even a small neural network with only one layer, any number of units and a non-polynomial activation function can reach a high accuracy score. Much more, they are even capable of approximating any given multivariate function [1, 2].

However, in order to achieve more accurate approximations, the number of parameters grows exponentially and thus reaching a point where single-layer neural networks are no longer usable in practice. For a large class of functions it can even be shown that the number of parameters needed by a shallow network are exponentially larger than the number of parameters needed by a deeper version with more layers for the same degree of approximation [3].

In modern implementations, multiple layers are used in most cases. This allows each layer to learn features of a dataset at different levels of abstraction, thus generalizing the core problem and finding a pattern in data rather than memorizing the whole set.

As with most concepts in machine learning and artificial neural networks, there exists a vague inspiration in biology. So-called receptive fields process visual stimuli in a hierarchical way as well. "Neurons in early visual areas have small receptive fields and are sensitive to basic visual features, e.g. edges and bars.", whereas "neurons in deeper layers of the hierarchy capture basic shapes, and even deeper neurons respond to full objects" [4]. This idea of how parts of the human brain process visual information at different stages with a different degree of abstraction can also be observed in neural networks with multiple layers. Simply put, early layers for example, are more likely to process colors and simple structures such as individual facial features or their skin color, whereas layers at the end of the network can relate found structures and recognize entire faces.

However, it seems that a higher number of layers incorporated in a neural network can lead to optimization problems and performs even worse in approximating certain functions in comparison to shallower ones.

Chapter 2

Fundamental Terms

In this chapter, fundamental concepts and terms in the area of neural networks are explained and illustrated with examples.

2.1 Classical Feed-forward Neural Networks

Basically, artificial neural networks, which will be simply referred to as neural networks in this thesis, are collections of linear equations, wrapped by a non-linear activation function and connected to a number of other equations. Input information is fed forward through the network by each equation's output (except the last layer) being the input of its successor.

2.1.1 Units

The most important building blocks of modern neural networks are so-called units – again name-inspired by the biological term for neurons in human brains. These units are usually grouped into layers of any size and connected to every unit in the layer before and after itself (dense layers).

Each of these units takes an input – either from the input vector or from the prior layer – and applies a linear transformation. This is followed by a non-linear activation function g responsible for the impact this unit creates. Simply put, depending on the linear transformation itself, the choice of this function is responsible whether its triggering the next unit or not. (As mentioned earlier, single layer neural networks with an arbitrary number of artificial neurons can be universal approximators for functions [2], which has been generalized by [1] for two-layer neural networks under the precondition that the activation functions are non-linear).

The input x gets transformed by a weight w and a chosen bias b, each of them initialized at the beginning (cf. Section 3.1.2) and then changed accordingly during the training process.

$$z = wx + b$$

$$a = g(z)$$

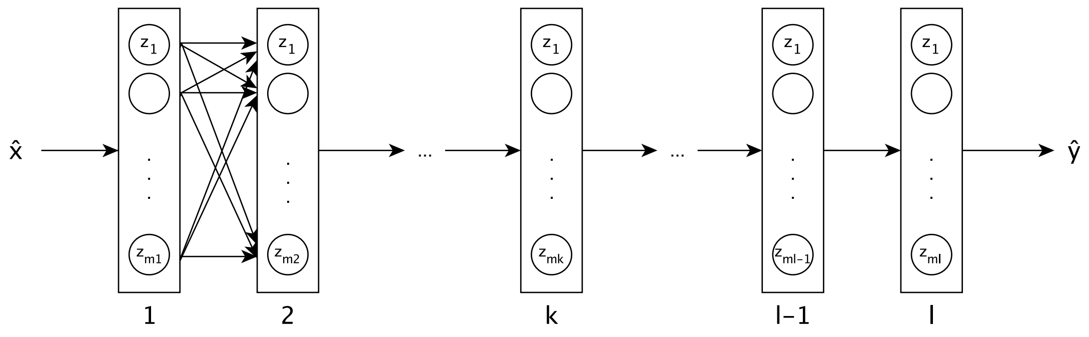


Figure 2.1: Plain feed-forward neural network with l densely connected layers and m units in each layer with k being any layer in the network. Dense connections have been simplified.

2.1.2 Notation

In order to encourage a consistent style, we agree on the following notation, when talking about feed-forward neural networks in general.

Concerning dense layers, the result of the linear equation $wx+b$ each unit outputs will usually be denoted by $z_i^{(k)}$, $i \in \{1, \ldots, m\}, k \in \{1, \ldots, l\}$ which is then transformed by a non-linearity, e.g. the Rectified Linear Unit (ReLU) $g(x) = max(0, x)$, referred to as $a_i^{(k)} = g(z_i^{(k)})$. Hereby, l usually represents the number of layers, k being any layer in the network, and m the number of units used in layer k. Therefore

$$a_i^{(k)} = g(z_i^{(k)}) = g(w_i^{(k)} a_i^{(k-1)} + b_i^{(k)})$$

as weights and biases are thought of stored in the unit itself. A hat over some letter will mean that this is not a single-value but a tensor or a matrix.

Due to the fact that this thesis deals with image classification problems in most cases, convolutional layers (cf. Section 2.2) are mainly used. In contrast to fully-connected dense layers, convolutional layers combine the idea of filters used in classical computer vision and modern neural networks. Since the convolutional operation is still a (specialized) linear operation [5], "convolving" over the image with a certain stride and size and performing the dot product, the operation itself will be denoted with an asterisk

$$z_i^{(k)} = w_i^{(k)} \star a_i^{(k-1)} + b_i^{(k)}$$

with $w_i^{(k)}$ being the tensor containing the filters.

2.1.3 Activation Functions

Choosing the activation function is crucial for the learning process and also relevant for the problem statement since it is basically setting the threshold for deciding whether a unit is activated or not.

In early neural networks the sigmoid function $sig(x)$ or $tanh(x)$ has been used in most cases, but it has been shown that these tend to let units die more easily (Section 3.1.1) during the learning process. Therefore, the Rectified Linear Unit (ReLU) function is used in most modern implementations [6].

Sigmoid

The Sigmoid function (Figure 2.2) "squashes" its input values into a range of $(0, 1)$, modeling concepts where units either fire or not fire due to its steep slope at $x = 0$.

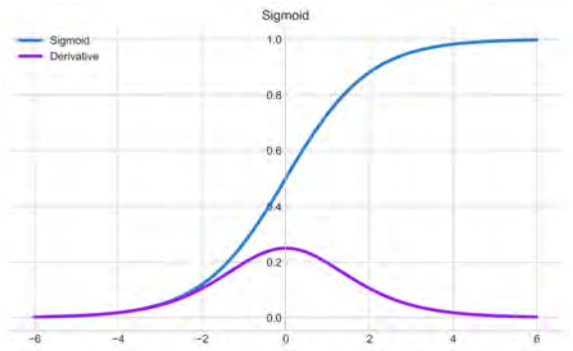


Figure 2.2: The Sigmoid function $sig(x) = \frac{1}{1+e^{-x}}$ and its derivative $\frac{d}{dx}(\frac{1}{1+e^{-x}}) = \frac{e^{-x}}{(1+e^{-x})^2}$

Hyperbolic Tangens

In contrast to the Sigmoid function the Hyperbolic Tangens (Figure 2.3) is centered around zero with values in $(-1, 1)$. It is defined as $g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

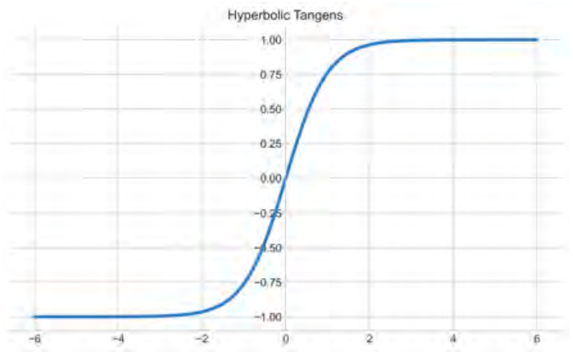


Figure 2.3: The Hyperbolic Tangens

Rectified Linear Unit (ReLU)

Defined as $ReLU(x) = max(0, x)$, ReLU (Figure 2.4) is basically the identity function for values bigger than zero and cuts off all information lower by setting them to zero.

Softmax

Unlike the other activation functions, Softmax accepts multiple inputs $\hat{x}$ and maps them to $(0, 1]$ so that the output vector sums up to one – representing a probability distribution. This is often used in classification problems with n different classes

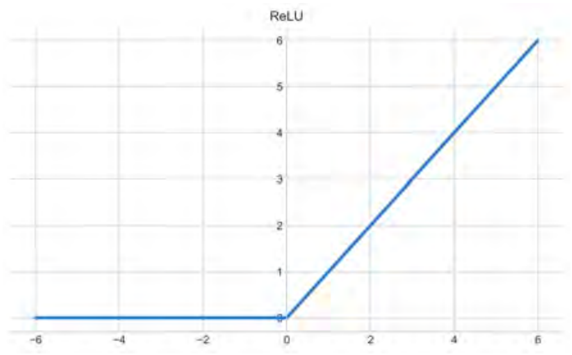


Figure 2.4: The ReLU activation function

where the last layer in the network uses Softmax in order to assign probabilities to each class. [5]

$$softmax(\hat{x})_i = \frac{e^{\hat{x}_i}}{\sum_{j=1}^{n} e^{x_j}} \text{ for } i \in \{1, \ldots, n\}$$

2.1.4 Training

After setting up and initializing the network (weights, biases) with parameters according to Section 3.1.2, the input values get fed forward through the network and the output is produced. In order to measure the quality of the output and how much it differs from the expected values (according to the training dataset), a certain metric is needed (often called error or loss function). This procedure is mostly referred to as learning, namely learning how to tweak parameters in order to achieve better results. An improvement in approximating the target function should directly affect the output of the error, so that minimizing the error function also leads to better results in the network.

Error functions

The chosen error function typically depends on the underlying problem. When dealing with regression problems, the most often used loss function is the Mean Square Error (MSE) [6]

$$MSE = \frac{\sum_{i=1}^{n}(y_i - y_i')^2}{n}$$

with n being the number of training examples, y_i a single labeled data point in the training set and y_i' the networks prediction for this point. Despite the high popularity, MSE is prone to amplifying bad predictions where results are far away from the actual labels due to the squared distance. In comparison, the Mean Absolute Error (MAE) is a small variant, which – as the name suggests – uses the absolute distance between prediction and the actual label.

$$MAE = \frac{\sum_{i=1}^{n} |y_i - y_i'|}{n}$$

When dealing with classification problems (compare Section 2.1.3), and binary classification problems in particular, a often used error function is the Binary-Cross-Entropy Loss (BCEL). It is based on the idea of cross-entropy describing the difference between the true underlying probability distribution P_{data} – or at least the empirical distribution based on the training samples – and the results P, predicted by the network. Having two classes with y_i either being one or zero, the Binary-Cross-Entropy is defined as

$$BCEL = -\frac{1}{N}\sum_{i=1}^{N} y_i \dot{\log}(y_i') + (1 - y_i)\dot{\log}(1 - y_i').$$

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is the most used gradient-based algorithm for optimizing a network in order to minimize the chosen loss function described in Section 2.1.4 [5]. As the name suggests, changes on weights and biases in the network are made based on the gradients of the loss function in respect to each single parameter.

The idea of SGD is based on a simple way of interpreting the gradient of a function. Consider a network with one single input and output. Then $y = L(x)$ models the loss function of this small network. L' or $\frac{dy}{dx}$ is called the derivative of L and describes the slope of L at the point x. This gives an indication on how much impact small changes of x have on the output y. However, as the networks grows and possibly makes use of multiple inputs, partial derivatives must be used in order to estimate this impact. In analogy to the single input example, $\frac{\partial}{\partial x_i}L(x)$ measures how L changes at point x when tweaking the parameter x_i. The gradient $\nabla L(x)$ at point x in this example is – by definition – a vector whose components are the partial derivatives of L in respect to the parameters at this point x.

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial x_1}(x) \\ \vdots \\ \frac{\partial L}{\partial x_n}(x) \end{bmatrix}$$

The gradient vector and its components are generally interpreted as the "direction and rate of fastest increase" [7, 5]. Therefore the negative gradient of an inspected parameter points at the direction of fastest **decrease**, which means that adding the negative gradient vector at x ultimately minimizes the loss function L.

At some points it makes sense to scale the gradient vector with some scalar ϵ, called the learning rate, determining the size of the step L makes at point x [5]. This can help in learning faster, but also bears the risk of running over a possible minimum if ϵ is chosen too big. The choice of a very small ϵ on the other hand, can slow down training, because steps towards a minimum are becoming small as well.

The described method of calculating partial derivatives for each parameter of a neural network – which can easily be multiple millions – is a computationally expensive task. Therefore, the real gradient is replaced by a stochastic approximation, calculated from a randomly selected subset of the dataset in most applications (SGD) [8].

2.2 Convolutional Neural Networks

As for now, the described networks only handled one-dimensional input data. Two-dimensional data like images for example would have been flattened by stacking each column of pixels on top of each other, obtaining only one dimension. However, with this approach valuable spatial information gets lost because neighboring pixels have typically a higher probability of being related than pixels that are far away from each other.

Besides that, processing images with multiple dimensions (including the depth for colors) using only dense layers as described in Section 2.1 lets the number of parameters in the network grow fast, making training this type of network unfeasible [6].

2.2.1 Filters and the Convolutional Operation

Similar to classical computer vision tasks, filter kernels are used in order to highlight and detect certain features like edges. The difference here is that kernels do not have to be defined by hand, but will be learned during training as parameters. By using the process of convolution, a filter can be applied to a given image.

In its most general form, the convolution is a mathematical operation (cross-correlation) on two functions $f, g : \mathbb{R}^n \mapsto \mathbb{R}$, producing a third one, which measures the overlap between them, when one function is "flipped" and shifted by x [6].

$$(f \star g)(x) = \int_{\mathbb{R}^n} f(z)g(x-z)dz.$$

In discrete terms this can be written as

$$(f \star g)(x) = \sum_n f(n)g(x-n).$$

By adding indices for the second dimension, this operation can be used to apply filters to input images. The idea behind this is that it is assumed that areas close to each other are more likely to be related than areas far away from each other. Hence, what happens is that the filter is applied to an area *(convolutional window)* of the image and then slides to the next part of the image. This procedure is repeated until the whole image has been scanned.

Let $f \in \mathbb{R}^{x \times y}$ be a two-dimensional greyscale image with dimensions $x \times y$ and ω a filter of finite size. We also suppose that ω consists of an odd number of $2N+1$ elements (typically much smaller than the actual image), indexed from $-N$ to N. Therefore, $\omega \in \mathbb{R}^{(2N+1)\times(2N+1)}$ and

$$g(x,y) = w \star f(x,y) = \sum_{i=-N}^{N} \sum_{j=-N}^{N} \omega(j,i) f(x-j, y-i) \tag{2.1}$$

is the image after the filter has been applied using the convolution [5]. As seen in Figure 2.5, the filter size is not limited to an odd number of elements though. Figure 2.5 also shows that the output size is inevitably smaller than its input. Since the filter moves within the edges of the input image, the output shape can be determined by

$$(x - x' + 1) \times (y - y' + 1)$$

assuming that $\omega \in \mathbb{R}^{x' \times y'}$.

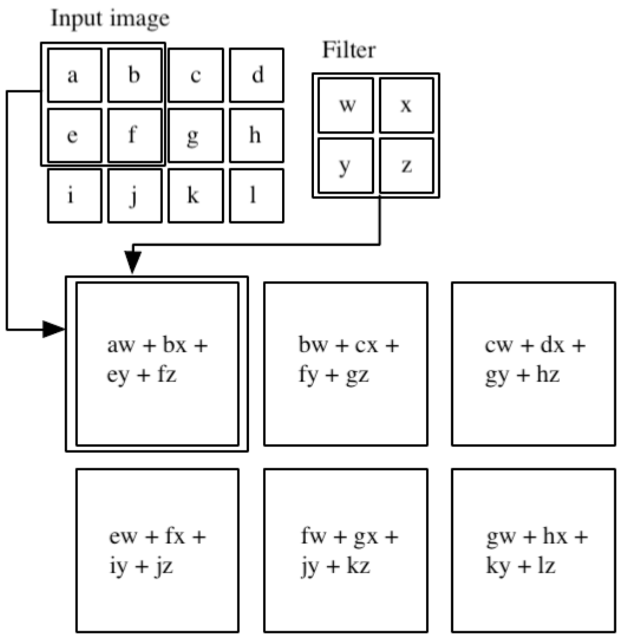


Figure 2.5: Example of a convolution with a filter size of 2×2 and a stride of 1 with valid padding, meaning that the filter only slides over areas that are entirely within the input image. This halves the size of the output in comparison to the input image.

2.2.2 Padding and Stride

However, it is not always desirable to reduce the size of the input data through convolution. Although input data is only reduced in small steps because filter sizes are generally not bigger than 7, there is a risk that information is compressed too much – especially in deep neural networks with many convolutional layers. One solution to solve this issue is to add extra columns and rows around the actual input, filled with zeros. This adds no extra information to the input but extends the data before the convolution so that the shape of the output is not smaller than the actual input shape *(valid padding).* If the output shape should be the same as the input shape *(same padding).*

In some cases, however, down-sampling data may be desired nonetheless and can be achieved by increasing the stride. Until now, the convolutional window has been moved across the input with one element a time for each row and column. If this step size is increased, the sliding window skips some overlapping values and the output shape decreases. This is a common practice for reducing the resolution of the output either for computational efficiency or because down-sampling is desired [6].

2.2.3 Pooling Layer

Another way of reducing resolution and aggregating information is using a pooling layer. Similar to the convolution, a pooling window slides over the input data as well. This time, however, it is not a filter which is applied but a function, "typically calculating either the maximum or the average value of the elements in the pooling window" [6]. It can be said that pooling works similar to what is shown in Figure 2.5 except that pooling does not have any parameters. Instead, all values in the current pooling window are aggregated to a single value by *maximum pooling* or *average pooling.*

This is helpful because neural networks are ultimately drawing information from raw data. This could be detecting edges in images for example. Imagine an image A and another image B shifted to the right by one pixel. In this use-case, the raw data is vastly different but the representation (edges in the image) is mostly the same. By using pooling layers, these differences can be balanced. Information in a certain area of the image is aggregated so that the a pixel's actual location does not matter that much on a small scale [6].

Chapter 3

Deep Neural Networks

As described in Chapter 1, deep neural networks with more layers are generally more efficient than shallow ones in terms of parameters and therefore the time needed for learning. This is, among other reasons specific for visual learning described up to this point, the main motivation for experimenting with deeper models.

3.1 Limitations and Arising Problems

In contrast to our expectations, even with allowing to scale units horizontally across multiple layers with bounded space, deep neural networks are encountering optimizations problems [9, 10].

3.1.1 Vanishing Gradients

As more layers are added to a network, gradients in early layers are vanishing and becoming arbitrarily small [11, 10]. This results in the networks loss function converging much slower and even having gradients near to zero and therefore contributing no change towards a local minimum.

One reason for this can be the use of the Sigmoid function as activation for layers [12], which makes the "logistic sigmoid activation (...) unsuited for deep networks with random initialization because of its mean value, which can drive especially the top hidden layer into saturation." [13].

The Sigmoid function 2.1.3 by its nature squeezes its input in a range of $(0, 1)$, asymptotically approaching both ends with a very steep slope. This means there is only a very small range where inputs get projected to values between zero and one and the rest of $\mathbb{R}$ having values very close to them. This becomes clearer when we take a look at the derivative of the Sigmoid function at $x = 0$. Then $\frac{d}{dx}\frac{1}{1+e^{-x}}(0) = \frac{e^{-x}}{(1+e^{-x})^2}(0) = \frac{1}{4}$. Therefore, the following applies to the output of units using the sigmoid function as its activation: $a^{(k)} \in (0, \frac{1}{4}]$.

Let $\mathcal{L}$ be the function measuring the loss using SGD. Then

$$\frac{\partial \mathcal{L}}{\partial \hat{w}^{(k)}} = \frac{\partial \mathcal{L}}{\partial \hat{a}^{(l)}} \cdot \frac{\partial \hat{a}^{(l)}}{\partial \hat{z}^{(l)}} \cdot \frac{\partial \hat{z}^{(l)}}{\partial \hat{a}^{(l-1)}} \cdot \frac{\partial \hat{a}^{(l-1)}}{\partial \hat{z}^{(l-1)}} \cdot \dots \cdot \frac{\partial \hat{z}^{(k)}}{\partial \hat{w}^{(k)}}$$

is the derivative of $\mathcal{L}$ with respect to the weight matrix in layer k applying the chain rule. Together with any $i \in \{1, \dots n\}$ and

$$\hat{a}^{(i)} = sigmoid(\hat{z}^{(i)})$$
$$\hat{z}^{(i)} = \hat{a}^{(i-1)} \cdot \hat{w}^{(i)} + \hat{b}^{(i)}$$

it can be easily seen that each derivative contains the derivative of the sigmoid function as well, which leads to values in $(0, \frac{1}{4}]$ being multiplied together and resulting in values closer to zero.

$$\frac{\partial \hat{a}^{(i)}}{\partial \hat{z}^{(i)}} \cdot \frac{\partial \hat{z}^{(i)}}{\partial \hat{a}^{(i-1)}} = \frac{\partial sigmoid(\hat{a}^{(i-1)} \cdot \hat{w}^{(i)} + \hat{b}^{(i)})}{\partial \hat{a}^{(i-1)} \cdot \hat{w}^{(i)} + \hat{b}^{(i)}} \cdot \frac{\partial \hat{a}^{(i-1)} \cdot \hat{w}^{(i)} + \hat{b}^{(i)}}{\partial sigmoid(\hat{a}^{(i-2)} \cdot \hat{w}^{(i-1)} + \hat{b}^{(i-1)})}$$

Another common practice that vanishes gradients as well is using the Gaussian distribution with $\mu = 0$ and $\sigma^2 = 1$ for weight initialization (cf. Section 3.1.2). This also results in weights near to zero.

So even with only a few layers in a deep neural network, the gradient can become very small using SGD with sigmoidal units and a Gaussian weight initialization. The more layers the network uses, the smaller the gradients become and results in not finding a proper minimum of the loss function due to exponentially small steps of the gradient towards a local minimum.

3.1.2 Troubleshooting

However, as the main problem training deep neural networks is based on the definition of the Sigmoid function and its derivative, using a different activation function together with clever weight initialization [13] as well as intermediate normalization layers [11] can help address these issues.

ReLU

The Rectified Linear Unit (described in 3.1.2) is a piece-wise defined function, zeroing all inputs smaller than or equal to zero and returning the identity else. The derivative – in contrast to the Sigmoid function – is therefore

$$\frac{d}{dx}ReLU(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0. \end{cases}$$

This means gradients will not vanish in most cases, because the chain of multiplications does not strictly consist of values < 1. One might argue that there is a possible risk of units "dying" out when the weight in combination with the bias is negative. Then the ReLU's output is zero and might hurt optimization by blocking gradient back-propagation. There are, however, experimental results suggesting that these zero activations can actually help supervised training as long as there is at least one path allowing the gradient propagating through [14].

Batch Normalization

Despite the fact, that using ReLU as the activation function g, initializing the parameters (weight matrices, etc.) carefully and choosing small learning rates helps to reduce the chance of vanishing or exploding gradients [11], another method proved success by stabilizing the distribution of non-linearity inputs.

Furthermore, it has been shown that networks converge much faster if the input data of layers is normalized to zero mean and a variance of one (unit-variance) [11]. As the name suggests, normalization is not applied to a whole layer at once but for mini-batches. A certain size m is set for a single batch with values $\mathcal{B} = \{x_1, \ldots, x_m\}$, where mean $\mu_\mathcal{B}$ and variance $\sigma^2_\mathcal{B}$ are calculated and each input $x_i \in \mathcal{B}$ is normalized to $\hat{x_i}$ [11].

$$\mu_\mathcal{B} = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{(mean)}$$

$$\sigma^2_\mathcal{B} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2 \qquad \text{(variance)}$$

$$\hat{x_i} = \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \underbrace{\epsilon}_{\substack{\text{added for}\\ \text{numerical}\\ \text{stability}}}}} \qquad \text{(normalization)}$$

However, just normalizing each input might "change what the layer can represent" [11]. When just normalizing inputs and for example applying the sigmoid function as its activation, the linear essence of the input is preserved, which somehow contradicts the application of such a non-linearity. Therefore, γ and β are introduced as learnable parameters, responsible for scaling and shifting the normalized input to

$$y_i = \gamma\hat{x_i} + \beta$$

Using Batch Normalization as a step before the non-linearity, it ultimately computes the following function

$$y_i = BN_{\gamma,\beta}(x_i) = \gamma\frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}} + \beta$$

and ensures zero mean and unit variance, thereby mostly eliminating the vanishing gradient problem.

Weight Initialization

Another crucial part of a network's architecture is how units of the network are initialized at setup. This can possibly be a cause of whether units die early in the learning process (become arbitrarily small or big and therefore ineffective, cf. Section 3.1.1) or perform well over the whole training phase [13].

One simple method for weight initialization that comes to mind is to initialize weights with some random distribution. However, if the random values are either too small or too big, gradients can easily vanish again, depending on the used activation function. When using differentiable activation functions like sigmoid for example, the gradient's slope easily becomes too small if randomly produced values are very big or very small due to high variance. If variance is lower on the other hand, gradients will move in a very narrow range and training will be slowed.

This does not happen for networks using ReLU activations 3.1.2 because the derivative can only be zero or one. Hence, the variance of the probability distribution is not quite important because there will always be positive and negative samples and therefore gradients with value zero or one.

But how is it possible to ensure that randomly chosen initialization values are in a somewhat "perfect" range in all cases?

Xavier Initialization. For activation functions differentiable at $x = 0$ (especially tanh 2.1.3), Kumar et al. [15] suggests to use a distribution with output variance of one in order to ensure faster convergence. From that they derive the Xavier Initialization [13] for tanh with

$$\rho^2 = \frac{1}{d^k}$$

where d^k is the number of input neurons in layer k. This can for example be achieved by multiplying standard normal distribution with

$$\sqrt{\frac{1}{d^k}}$$

He Initialization. When using the ReLU activation function, which is non-differentiable at $x = 0$, the variance of the distribution used for random initialization should be

$$\rho^2 = \frac{2}{d^k}$$

with

$$\sqrt{\frac{2}{d^k}}$$

as a factor [16].

3.1.3 Degradation Problem

Even though above methods helped to allow neural networks with a higher number of layers it seems as if accuracy saturates at first but then degrades rapidly with increasing depth of a network [10] (Figure 3.1), remaining at a higher *training* error than less complex networks.

This seems counter-intuitive at first, because one would expect a model with more parameters (due to more layers) to perform at least as good as if these extra layers would not have been added. Consider a shallow neural network with n layers and another more complex one with m layers where $m > n$. In order to learn the same function on both we could simply replace the first n layers of the deeper NN with the exact same layers of the shallower one and make the *residual* $m - n$ layers learn the identity function.

These $m - n$ layers should not add further complexity to the network as well as not increase the training error, since they will just compute the identity. If the now deeper network finds a more complex function to learn, it can be expected that the additional layers would be able to find this solution. However, when comparing the

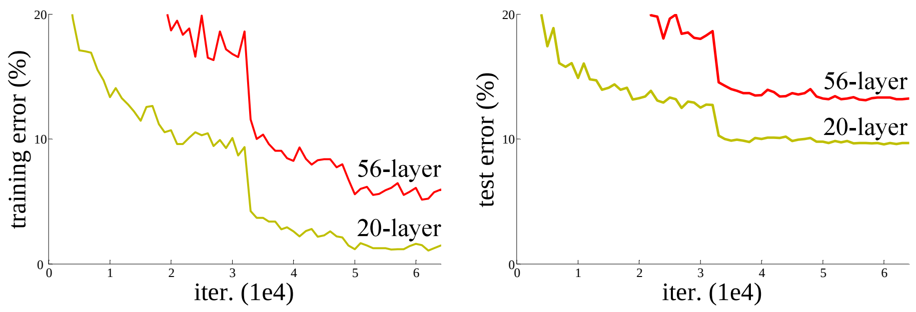


Figure 3.1: Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error [10]

.

training error of this constructed network to shallower ones it seems that this is not the case (Figure 3.1).

In addition to that, Batch Normalization (BN, cf. 3.1.2) has been used in the experiments made by He et al. [10] and by monitoring the gradients, it could be argued that these observations are *unlikely* a result of vanishing gradients. The authors conclude that it is somehow hard for deeper networks to learn the identity function and that it is possibly easier approaching a zero mapping ($f(x) = 0$ instead of $f(x) = x$). In addition to that they "...conjecture that the deep plain nets may have exponentially low convergence rates, which impact the reducing of the training error." [10].

3.2 Residual Neural Networks

In order to address this problem described in Section 3.1.3 and allow deeper layers to be added to the network, He et al. [10] introduce skip- (sometimes referred to as shortcut-) connections to the network.

An analogy for this idea roughly exists in biology [17] where layers in the cerebral cortex in the brain of mammals receive input from prior layers, skipping the intermediate ones.

3.2.1 Identity Block

These shortcut connections in Residual Networks (ResNets) simply take the output of some layer, skip one or more layers (the residual block) and add the output unchanged – as an identity mapping – to the linear output of the desired layer before the activation function is applied. This passes information from early layers much deeper into the network and allows the network to easily learn the identity function by setting $F(x) = 0$, thus driving all weights and biases in the stacked residual layers between the shortcut connection to zero and leaving x as the identity. If, however, the identity mapping is not the optimal solution for this block, weights and biases are learned to adjust the value of the residual $F(x)$.

Additionally, because shortcut connections neither add extra parameter nor computational complexity, the network can still be trained end-to-end by Stochastic Gradient Descent (SGD) using backpropagation [10]. Furthermore, gradients are

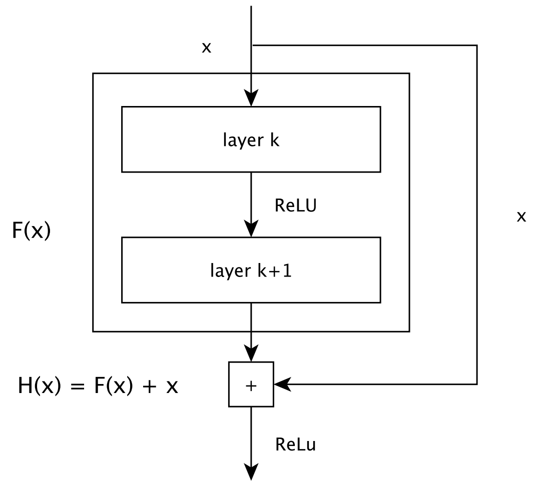


Figure 3.2: Detail from a larger Residual Network with a shortcut connection reaching from $x = a^{(k-1)}$ of layer k to the linear output of layer $k+1$ producing $F(x)+x$, where $F(x) = z^{(k+1)}$ (comp. Notation in Section 2.1.2)

much less likely to vanish, because shortcut connections can be used to preserve gradient, due to the identity.

Let $\mathcal{L}$ be the loss function and x the input to a residual block as shown in Figure 3.2. When finding the partial derivative for the gradient in respect to x, the identity shortcut resolves to one, thus not decreasing the gradient any further.

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial H}\frac{\partial H}{\partial x} = \frac{\partial \mathcal{L}}{\partial H}(\frac{\partial F}{\partial x} + \underbrace{1}_{\text{derivative of x}}) = \frac{\partial \mathcal{L}}{\partial H}\frac{\partial F}{\partial x} + \frac{\partial \mathcal{L}}{\partial H}$$

3.2.2 Convolutional Block

Due to the addition of the identity, both x and $F(x)$ must have the same dimensions, which cannot always be guaranteed, especially in computer vision when using convolutional or pooling layers in between. Either only convolutions preserving the dimension are applied in the residual block or methods are used to adjust dimensionality if it increases in the residual block.

Padding

The simplest method to achieve the same dimension is to fill the required number of entries with zeros until the dimension is properly adjusted. This introduces no extra parameters [10].

Linear transformation

Another way of adjusting dimensions is introducing a matrix W_s that will be learned during backpropagation along with the other parameters [10]. The residual block then turns into

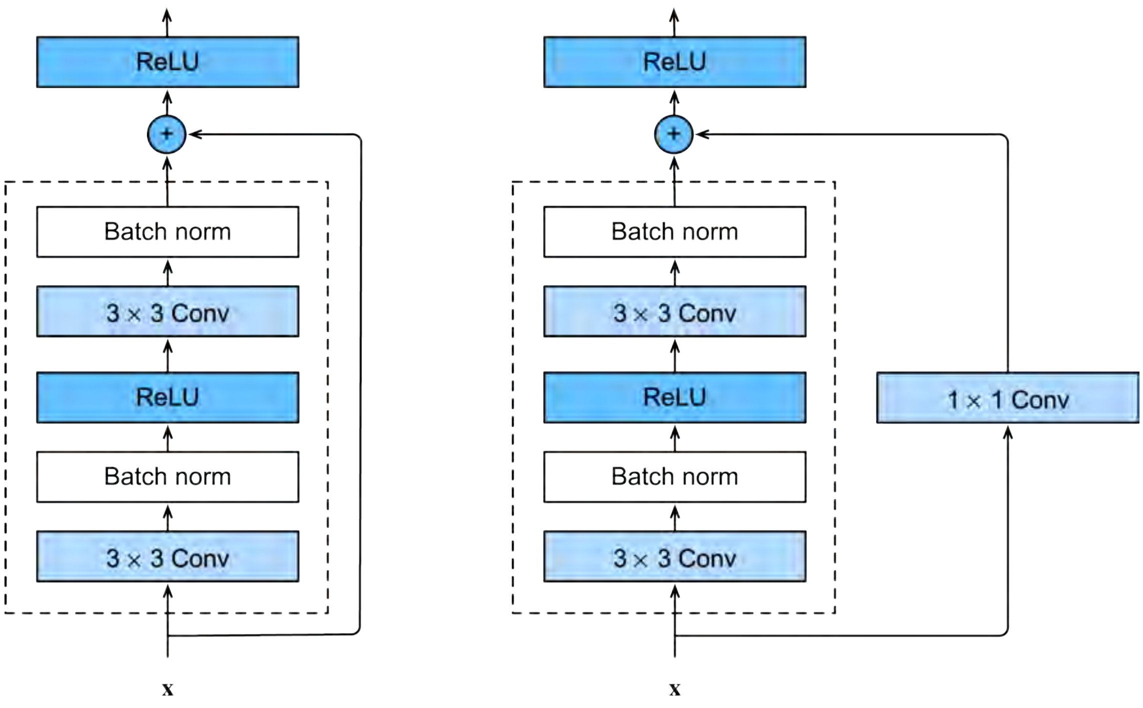


Figure 3.3: ResNet blocks with identity and convolutional type of skip connection [6].

$$H(x) = F(x) + W_s x$$

where

$$W_s \in \mathbb{R}^{m \times n} \text{ for } x \in \mathbb{R}^{n \times q} \text{ and } F(x) \in \mathbb{R}^{m \times p},\ m, n, p, q \in \mathbb{N}$$

3.2.3 Architecture

In order to make ResNets comparable to plain networks regarding their performance, He et al. [10] utilize the exact same architecture, despite the use of shortcut connections in ResNets. They consist of stacked convolutional layers with mostly 3x3 sized filters. They apply two simple design rules [10]

- For the same output feature map size, the layers have the same number of filters
- If the feature map size is halved, the number of filters is doubled so as to preserve the time complexity per layer

After the last convolutional layer (or the end of an identity block), global average pooling is applied and the input runs through a 1000-way fully connected layer. Because ResNets performance has been originally measured with the ImageNet dataset [18], softmax has been used as this dataset is typically being used on classification problems. Downsampling is performed by using a stride of two in convolutional layers [10].

The authors use multiple methods to augment the already existing 14M samples included in the ImageNet [18] dataset. Firstly, the shorter side is randomly sampled in $[256, 480]$ and cropped to a 224x224 image. Then, random images are flipped horizontally and per-pixel mean is subtracted from all images. After each convolution operation, Batch Normalization (BN) is applied before the activation function (cf. Figure 3.3). This step is mainly responsible for the absence of vanishing gradients, as BN "ensures forward propagated signals to have non-zero variances" [10]. They also "verify that the backward propagated gradients exhibit healthy norms", so that "neither forward nor backward signals vanish" [10]. Weights are initialized according to [16] and the network is trained by SGD (cf. Section 2.1.4) using a batch size of 256 and a learning rate of 0.1, divided by ten if the loss function plateaus. The authors train their ResNets with up to 60×10^4 iterations. They use a training set consisting of 1.28 million images. Making use of the mentioned 256 image per batch, they need around $\frac{1280000}{256} = 5000$ iterations for the network to see the whole dataset. This means that the network sees the entire dataset a total of 120 times (epochs).

Chapter 4

Experiments using Residual Networks

In this chapter a method for assigning multiple labels to an image using Residual Networks will be examined. The used architectures will be introduced and its performance on this task and dataset will be compared to ”plain networks” as referred to by He et al. [10].

4.1 The CULane Dataset

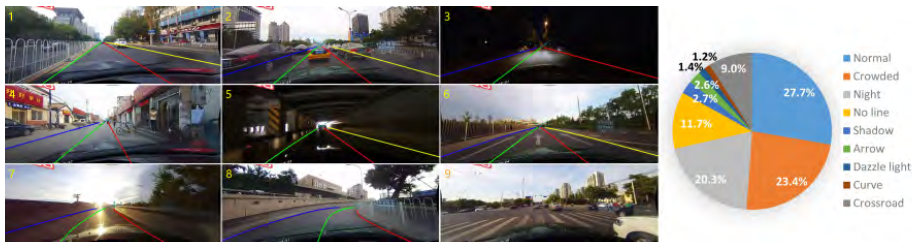


Figure 4.1: Example images and distribution of available classes in the CULane dataset [19]

Originally developed for traffic lane detection, the authors of [19] made the dataset available for academic purposes. It consists of 133.235 frames, extracted from more than 55 hours of video material, collected by six different vehicles in Beijing [20]. The images are shot from the view of a car’s dashboard and should be annotated with multiple labels from the following nine different categories:

- Normal, 27.7%, 8676 samples,
- Crowded, 23.4%, 7329 samples,
- Night, 20.3%, 6358 samples,
- No line, 11.7%, 3664 samples,
- Shadow, 2.7%, 845 samples,
- Arrow, 2.6%, 814 samples,

- Dazzle light, 1.4%, 438 samples,
- Curve, 1.2%, 375 samples,
- Crossroad, 9.0%, 2819 samples.

However, only 31.323 images have been labeled with classes distributed unequally over the dataset (cf. Figure 4.1 for exact distribution).

normal	crowd	hlight	shadow	noline	arrow	curve	cross	night	filename
1.	0.	0.	0.	0.	0.	0.	0.	0.	/06051417_0693.MP4/01...
0.	0.	1.	0.	0.	0.	0.	0.	0.	/05251426_0416.MP4/04...
0.	0.	0.	0.	0.	0.	0.	0.	1.	/05252231_0536.MP4/00...
1.	0.	0.	0.	0.	0.	0.	0.	0.	/05250455_0302.MP4/04...
0.	1.	0.	0.	0.	0.	0.	0.	0.	/05251502_0428.MP4/04...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.1: Example of the formatted CULane dataset containing nine classes and the filepath to the image.

4.2 Multi-Label Classification

Due to the fact that the performance of ResNets on predicting multiple labels should be examined in this thesis, the core problem is a multi-label image classification task. As opposed to multi-class classification, where each image vector is assigned only one from a set of multiple classes, the prediction in this case may include more than one label. However, as the given dataset was originally intended for traffic lane detection and only their test set (26% of all collected samples) has been divided into nine categories (Figure 4.1), each image is only assigned to one class.

Nonetheless it is possible to use this dataset for multi-label classification, as thresholds for considering a prediction as sufficiently good must be set anyway. In this case, instead of the best prediction (predicted class), the best k predictions are selected as labels (cf. Section 4.5.3).

4.3 Metrics

In classical binary classification problems, the quality of a solution can be evaluated by comparing true positive (tp), false positive (fp) and false negative (fn) results. Specifically precision and recall are two important measures defined as [21]

$$precision = \frac{tp}{tp + fp}$$

$$recall = \frac{tp}{tp + fn}$$

and describe how good a model is at predicting the positive class in general (precision) and how good it is at predicting the positive class if the predicted outcome is positive as well (recall). As described in Section 4.1, the labels on the CULane dataset are quite imbalanced, with some classes accounting for less than 3%. A common practice for this issue is to use the F_1 (harmonic mean of precision and recall) or the more generalized F_β score [22] as metrics. The former one is defined as

$$F_1 = \frac{2 \times (precision \times recall)}{precision + recall}$$

whereas the F_β score generalizes this ratio by introducing β, as a way to give either the precision or the recall more importance.

$$F_\beta = \frac{(1 + \beta^2) \times (precision \times recall)}{\beta^2 \times precision + recall}$$

As for this multi-labeling problem β will be set to $\beta = 2$ which means that recall will be valued twice as important as precision. More attention is put on minimizing false negatives, which is desired for these experiments – assuming that having "more information" in the sense of detecting more false negatives as true is better than having less. The result will be a value in $[0, 1]$.

However, these metrics only make sense when dealing with a binary classification problem. Since there exist more than two classes in this case, the F_β score is calculated for each label in relation to all other labels and is than averaged (cf. Listing 4.1).

In order to make the results of these experiments more comparable to He et al. [10], top-1, top-2 and top-3 rates will be used to measure the performance of the networks as well. The top-k rate denotes that out of all probabilities for a given input image, the correct label is within the top-k predictions. Two constraints must be made:

- Due to the fact that CULane instead of ImageNet is used as a data source, top-k error rates with $k > 3$ are not purposeful because only nine classes exist (ImageNet consists of 1000 classes).
- In addition to that, top-k rates are not evaluated per iteration due to increased time and technical efforts.

As shown in Figure 4.7, the averaged difference (in %) between the prediction for top-1 and top-2 and between top-2 and top-3 are marked in parenthesis below the top-k values as well. This highlights the frequently stark difference between the two involved ratings and is among others an effect of the only single-class labeled CULane dataset (more in Section 4.5).

Listing 4.1: Python implemented version of the F_β score for multi-label problems using Keras. Epsilon is used for numerical stability.

```python
from keras import backend as b


def fbeta(true, pred, beta=2):
    pred = b.clip(pred, 0, 1)
```

```
tp = backend.sum(b.round(.clip(true * pred, 0, 1)), axis=1)
fp = backend.sum(b.round(b.clip(pred - true, 0, 1)), axis=1)
fn = backend.sum(b.round(b.clip(true - pred, 0, 1)), axis=1)

precision = tp / (tp + fp + b.epsilon())
recall = tp / (tp + fn + b.epsilon())

return b.mean(
            (1 + (beta ** 2)) * (precision * recall) /
            ((beta ** 2) * precision + recall + b.epsilon())
    )
```

4.4 Architecture

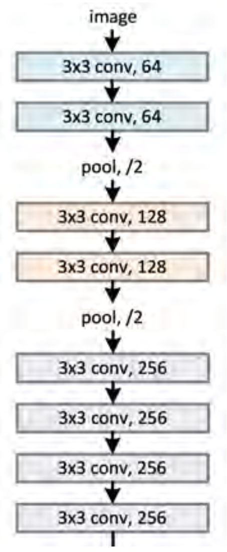


Figure 4.2: Detail of the VGG-18 network scheme.

At first, a baseline model has been developed by using an architecture similar to VGG-nets [23] with 18 layers in total (cf. Figure 4.2). Each pair of convolutional layers with 3x3 filters is succeeded by a max pooling layer. The number of filters doubles after each of these blocks in order to preserve time complexity per layer [10]. The convolutional layers use the ReLU activation function and same padding in order to guarantee the same dimensions of the feature maps. The stacks of convolutional layers are then followed by two fully-connected layers with 4096 units and another one with nine units and softmax (as described in Section 2.1.3) as its activation function in order to output predictions for each label.

In order to make the experiments as comparable as possible, the architecture regarding ResNets described in Section 3.2.3 is mostly maintained. A full overview can be seen in Table 4.2.

Scale Augmentation (changing scale on images in order to increase the number of samples) is not utilized since images in the CULane dataset are made using dashboard-mounted cameras in vehicles, which typically do not change scale during recording. However, images are horizontally flipped in order to increase the amount

layer name	**plain-18**	**plain-34**	**ResNet-18**	**ResNet-34**
conv 1	3x3, 64	3x3, 64	7x7, 64, stride 2	
	3x3, 64	3x3, 64 (3x)	3x3 max pool, stride 2	
conv 2.x	3x3, 128	3x3, 128	3x3, 64	3x3, 64
	3x3, 128	3x3, 128 (4x)	3x3, 64 (2x)	3x3, 64 (3x)
conv 3.x	3x3, 256	3x3, 256	3x3, 128	3x3, 128
	3x3, 256 (2x)	3x3, 256 (3x)	3x3, 128 (2x)	3x3, 128 (4x)
conv 4.x	3x3, 512	3x3, 256	3x3, 256	3x3, 256
	3x3, 512 (2x)	3x3, 256 (3x)	3x3, 256 (2x)	3x3, 256 (6x)
conv 5.x	3x3, 512	3x3, 512	3x3, 512	3x3, 512
	3x3, 512 (2x)	3x3, 512 (3x)	3x3, 512 (2x)	3x3, 512 (3x)
	average pool			
	9d fully connected, softmax			

Table 4.2: Architecture of the implemented plain and residual Networks.

of available training data, which is in line with the above mentioned statement because this only changes the semantic meaning of the picture.

Input images are resized from 1640 $\times$ 590 Pixels to 328 x 118 Pixels in order to reduce the amount of data the network needs to process and thereby cutting used computing power as well. A loss of information by this practice is not expected.

Due to lower hardware capabilities than the authors of He et al. [10] have access to, mini-batch size is reduced by the factor of eight which minimizes the used memory of our Graphical Processing Unit (GPU). Although this helps to train faster, the estimated gradients become more noisy because the network only sees an even smaller fraction of the whole training set. That is why learning rate is lessened to 1×10^{-4} in order to not being pushed out of a "valley" in the loss function by noisy gradients.

The CULane dataset contains a total of 31323 labeled samples, which means that about $\frac{31323\dot{0}.75}{32} \approx 734$ iterations are required for the network to perceive the whole dataset (25% are used for validation). In order to reach the same number of iterations as in [10], about 817 epochs are necessary. However, the available hardware with a varying training time between 428s and 2561s per epoch, meaning a total time of 97h – 581h (24 days), sets limits on how much iterations could be done.

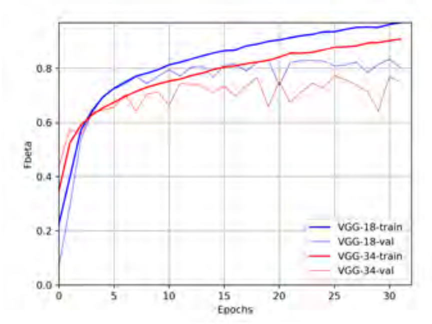
Depending on the hardware used for training, which ranges from a powerful server kindly provided by the Institute for Software Systems in Technical Applications of Computer Science (FORWISS) Passau (NVIDIA TITAN V with approx. 10GB of usable memory GPU memory) to the services made available by Google Colaboratory and Kaggle (limited to 9 hours of training), only between 1.2×10^4 and 47×10^4 iterations could be achieved (in comparison with up to 60×10^4 in [10]).

4.5 Evaluation and Problem Analysis

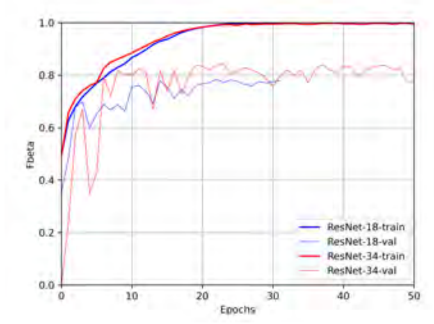
4.5.1 Results

Plain Networks

It was possible to show the degradation problem (cf. Section 3.1.3) even with the mentioned difficulties (less iterations, smaller dataset). The results in Table 4.3 suggest that the deeper 34-layer plain network has a lower Top-1 accuracy rate than the shallower 18-layer plain network. Figure 4.3 (a) shows the training process over 32 epochs (2.35×10^4 iterations), where it can be seen that the deeper 35-layer network has a lower F_β score on the training set throughout the whole training process despite the fact that the solution space of the 18-layer plain network is a subspace of that of the 34-layer network [10].



(a) Plain Networks of 18 and 34 layers.



(b) Residual Networks of 18 and 34 layers.

Figure 4.3: Training on CULane with plain and ResNets measured by the F_β score.

Residual Networks

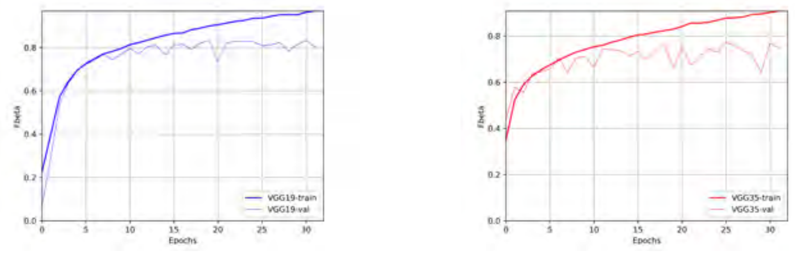
In comparison to the plain networks it seems as if the described behavior changed in so far as the accuracy rate switched. The deeper 34-layer ResNet proves to have a higher F_β validation score over the whole training as its shallower 18-layer counterpart (cf. Figure 4.3 (b)). This shows that the observation, that adding more layers to a network possibly leads to optimization problems, does not hold in this case. In addition to that, the deeper Residual Network seems to have a higher top-1 score than the shallower one as well. It even outperforms the deep 34-layer plain network whereas the 18-layer plain network still earns the highest rates.

	plain	ResNet
18 layers	86,78	79,88
34 layers	80,87	82,88

Table 4.3: Accuracy rate (%) showing the percentage of true top-1 predictions on CULane validation set.

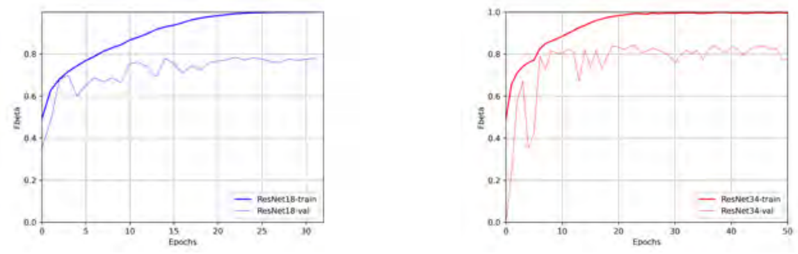
It can be said that the degradation problem (Section 3.1.3) is well addressed in these experiments, showing less performance on the deep plain network and higher performance on the deep ResNet when compared to their shallower reference model. However, these experiments on this specific CULane dataset also experience the best performance on the shallow 18-layer plain network.

4.5.2 Generalization Problems



(a) VGG-18 (b) VGG-34

Figure 4.4: Performance measuring using the F_β score.



(a) ResNet-18 (b) ResNet-34

Figure 4.5: Performance measuring using the F_β score.

As can be seen in Figures 4.4 and 4.5 the graph of the train and validation F_β score clearly head into two different directions, suggesting that both models experience relatively strong overfitting as early as from the seventh epoch on. Both plain and residual networks also nearly reach a perfect F_β score after 30 epochs. This indicates that they are pretty good on predicting the training set but perform worse on unseen data. This observation is reinforced by identifying the stark difference of the F_β score between the training and validation set. It seems as if both model types rather memorize the given data than identifying a general pattern in it. This has several reasons.

dataset size. Although a relatively big part of the training dataset has been used as test data (25%, shuffled before split) – concerning a total of only 31.323 samples – it would have been better to use a separate validation set as well (despite this approach is controversial [6]). In this thesis, validation and test dataset are used synonymously. There has been no separate test set. Therefore, the metrics incorporating test data used in this thesis are actually using the validation data. However, there is generally not enough data to support the problem statement in a way that it is possible to gain meaningful results.

Imbalanced data. This is only reinforced by the fact that the nine labeled classes are highly imbalanced (cf. Figure 4.1). It is intuitively clear that a feature only supported by less than 700 images performs less good than another with several thousands. Compare "Crowded" with 23.% and "Curve" with only 1.2% in the whole dataset.

Abstract features. In addition to that, the problem statement (assigning multiple labels to given images) in combination with the CULane dataset is rather abstract, which supports the problem of imbalanced data. The labeled classes represent features that intuitively are rather different in an abstract way as well. "Curve" or "Arrow" both are objects with a defined shape, which can easily be mathematically modeled, whereas "Crowd" or "Dazzle light" are descriptions of a strongly contrasting type.

4.5.3 Multi-Labeling Threshold

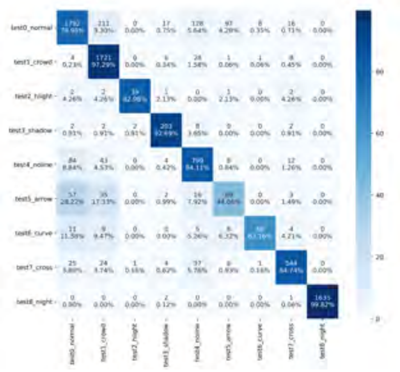
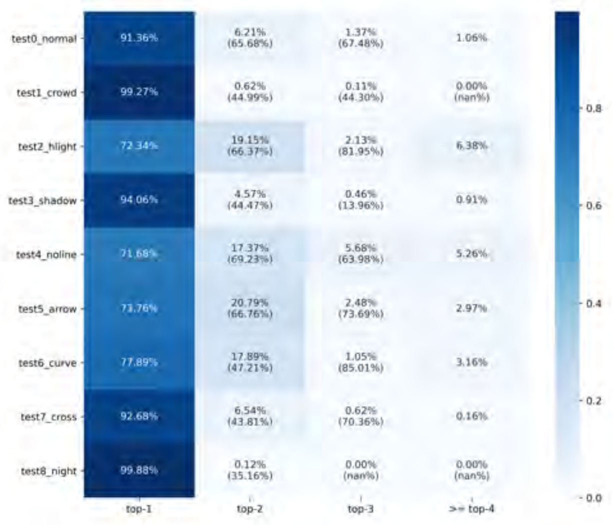


Figure 4.6: Confusion Matrix showing percentage of actual and predicted numbers per class.

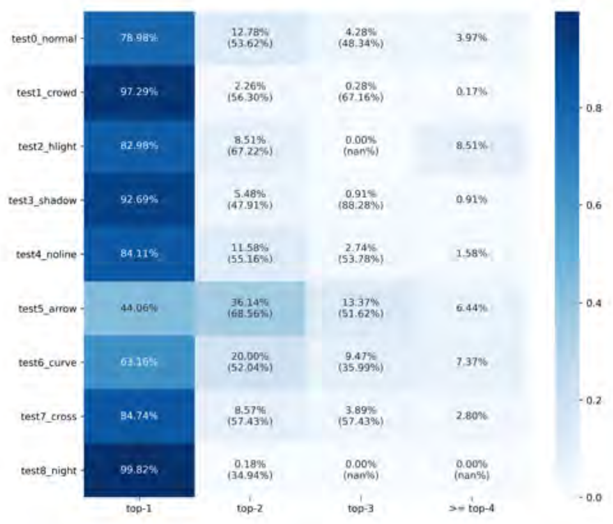
Until now the initial task has been treated as a multi-class labeling problem, where a given input is assigned one out of n possible classes. In order to assign multiple labels, the predictions must be examined for their quality. A threshold is required up to which predictions can be considered sufficiently good. Otherwise all labels are correct to some extent, because predictions for each label must sum up to one and are therefore greater than or equal to zero.

As described in Section 4.3, the top-k predictions for $k \in \{1, 2, 3\}$ have been analyzed. The results of all remaining labels have been combined into a fourth rating named "≥ 4".

Figures 4.7 and 4.8 show the results known from Table 4.3 broken down into the nine different classes. Similar to the confusion matrix (as an example plotted for the deep plain network in Figure 4.6), this type of plot shows the percentage of correct predictions sorted into ranks (more in Section 4.3). The values below these denote the delta between two consecutive percentages. With this additional information it can be clearly seen that CULane training data samples are only assigned to single classes and not multiple ones. The differences arrange around 50% in most cases, which means that the highest prediction differs from the second highest prediction by 50%. This makes discussions about the choosing multiple labels obsolete.

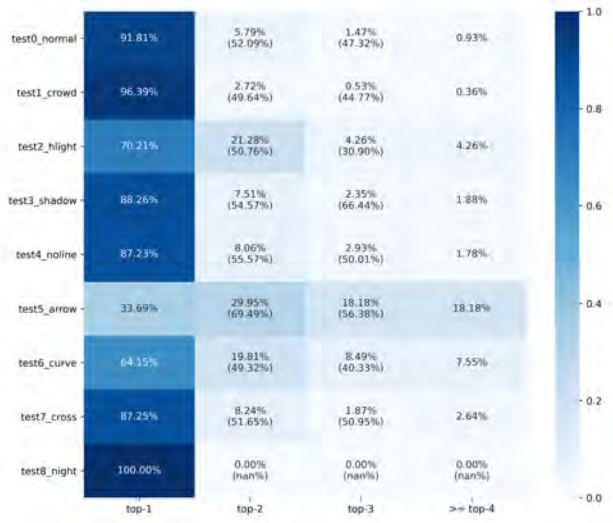


(a) VGG-18

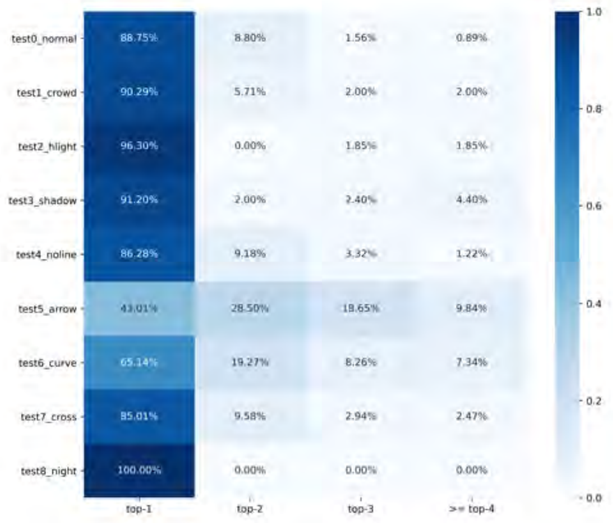


(b) VGG-34

Figure 4.7: Top-1, Top-2 and Top-3 accuracy rate of 18 and 34 layer plain network. The fourth column denotes that the correct prediction occurred in a place ≥ 4.



(a) ResNet-18



(b) ResNet-34

Figure 4.8: Top-1, Top-2 and Top-3 accuracy rate of 18 and 34 layer residual network. The fourth column denotes that the correct prediction occurred in a place ≥ 4.

Chapter 5

Conclusion and Future Work

This thesis shows that adding short-cut connections as He et al. [10] use it in their work, helps reducing the impact of degradation in deeper networks and even in out-performing shallow networks in some tasks. As mentioned at the beginning, deeper networks allow each layers to work on a different abstraction level of a feature and use less units for a certain degree of approximation.

Regarding the application on multi-labeling classification in combination with the CULane dataset, neither plain nor residual networks performed sufficiently good on this task. As described in Chapter 4.5.2, this is mainly due to external and non-influenceable reasons like the number of labeled samples or the labels itself being on different abstract levels. A conclusion on performance problems of ResNets in general cannot be drawn from this.

This however suggests that for datasets with a limited number of labeled examples, it sometimes is better to develop less complex models than used in this thesis. Even the smaller 18 layer plain network proofs to have too many trainable parameters, than what can be met by the complexity of examples in the CULane dataset.

Another area of artificial neural networks worth investigating for this multi-labeling problem is transfer learning. Layers at the end of a model, that has already been trained, are replaced by untrained layers and adapted to learn features of a new problem. However, it might be difficult to find a suitable dataset that sufficiently supports the features of CULane in a way that new layers can gain enough advantage.

List of Figures

List of Tables

Bibliography

[1] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of Control, Signals, and Systems*, vol. 2, pp. 303–314, Dec. 1989.

[2] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, pp. 359–366, Jan. 1989.

[3] S. Liang and R. Srikant, “Why deep neural networks for function approximation?,” 2017.

[4] A. Veit, M. J. Wilber, and S. J. Belongie, “Residual networks are exponential ensembles of relatively shallow networks,” *CoRR*, vol. abs/1605.06431, 2016.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[6] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning.* 2020. `https://d2l.ai`.

[7] D. Bachman, *Advanced calculus demystified.* McGrawHill, 2007.

[8] S. Sra, S. Nowozin, and S. Wright, *Optimization for Machine Learning.* Neural information processing series, MIT Press, 2012.

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *J. Mach. Learn. Res.*, vol. 15, p. 1929–1958, Jan. 2014.

[10] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015.

[11] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” 2015.

[12] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–44, 05 2015.

[13] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 249–256, 01 2010.

[14] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 315–323, JMLR Workshop and Conference Proceedings, 11–13 Apr 2011.

[15] S. K. Kumar, "On weight initialization in deep neural networks," *CoRR*, vol. abs/1704.08863, 2017.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015.

[17] Thomson, "Neocortical layer 6, a review," *Frontiers in Neuroanatomy*, 2010.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[19] X. Pan, J. Shi, P. Luo, X. Wang, and X. Tang, "Spatial as deep: Spatial cnn for traffic scene understanding," in *AAAI Conference on Artificial Intelligence (AAAI)*, February 2018.

[20] X. Pan, J. Shi, P. Luo, X. Wang, and X. Tang, "Culane dataset," February 2018. `https://xingangpan.github.io/projects/CULane.html`.

[21] D. L. Olson and D. Delen, *Advanced Data Mining Techniques*. Springer, 2008.

[22] N. Chinchor, "MUC-4 evaluation metrics," in *Fourth Message Uunderstanding Conference (MUC-4): Proceedings of a Conference Held in McLean, Virginia, June 16-18, 1992*, 1992.

[23] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2015.

Eigenständigkeitserklärung

Hiermit bestätige ich, Christoph Sonntag, dass ich die vorliegende Arbeit selbstständig und ohne unzulässige Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich und sinngemäß übernommenen Passagen aus anderen Werken kenntlich gemacht habe. Die Arbeit ist weder von mir noch von einer anderen Person an der Universität Passau oder an einer anderen Hochschule zur Erlangung eines akademischen Grades bereits eingereicht worden.

Ort, Datum Unterschrift