Dense Face Detection with Convolutional Neural Networks

Nikita Zozoulenko
nikita.zozoulenko@gmail.com
Katedralskolan Linköping
Supervisor: Rickard Engström

March 11, 2018

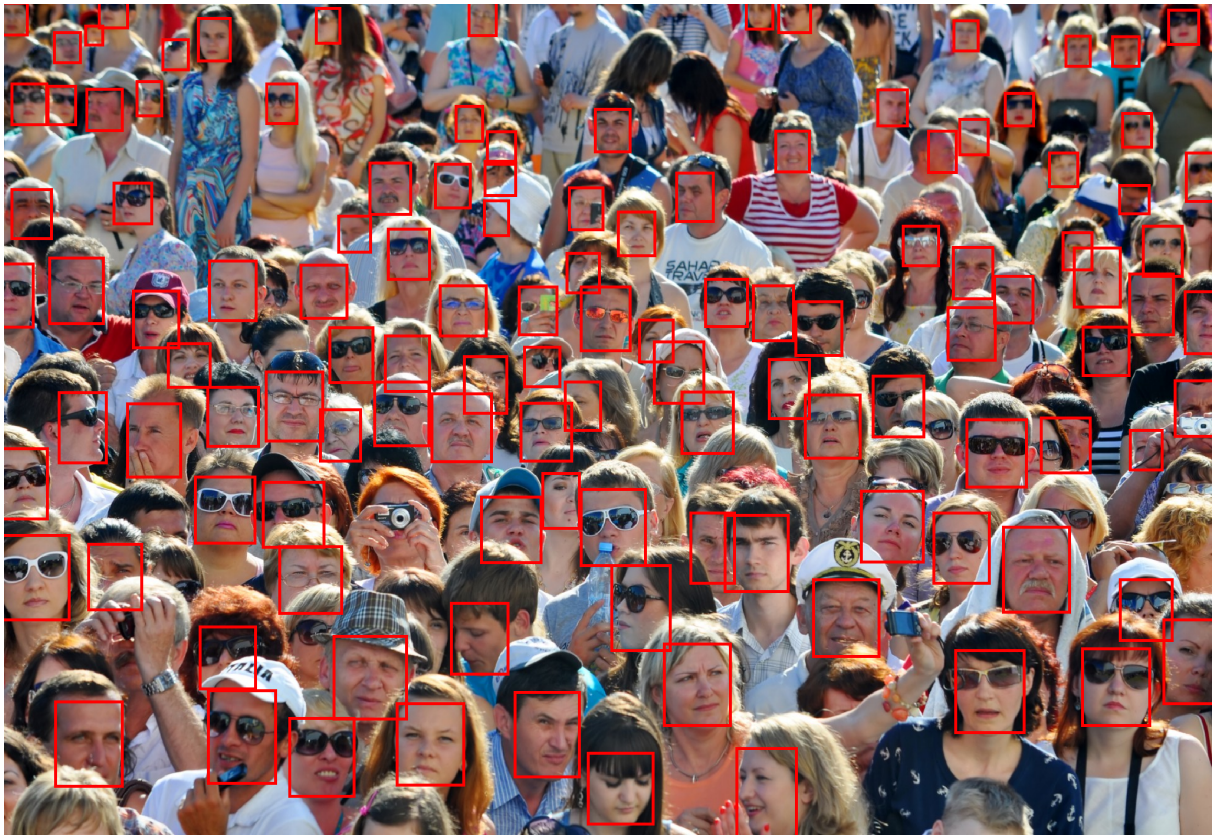


Figure 1: The proposed convolutional neural-network-based model FaceResNet is capable of dynamically detecting over 200 faces in a crowded scene for various scales, lighting and occlusions. A detection is shown as a red boudning box.

Abstract

Convolutional neural networks are currently the state of the art model for tasks such as image classification, image recognition and object detection for computer vision systems. In this paper the mathematical model behind the standard feed-forward neural network and the convolutional neural network is derived. The performance of two models, a convolutional neural network and a feed-forward neural network, are then compared on the task of classifying handwritten digits. A convolutional neural-network-based model is then constructed to make a cutting edge face detector for real time video, being able to successfully detect over 200 faces in a crowded scene for various scales, lighting and occlusions.

Contents

1 Introduction

1.1 Background

An Artificial neural network is a biologically inspired machine learning model that tries to replicate the way the brain in mammals functions. There are many different kinds of Artificial neural networks which vary in structure and architecture. The basic principle behind neural networks is that they are made up of a number of layers of neurons. The layers are built sequentially such that the output of one layer is the input to the next layer. The number of layers a neural network is constructed out of is called its depth. How the neurons of the previous layer are connected to the neurons of the next layer depends on the specific type of artificial neural network. [1]

Similar to other machine learning models, an artificial neural network wants to predict a predetermined number of values, given an input of a number of data points. For instance, a neural network can be given 784 pixel values from a 28×28 sized image of handwritten digits as an input. The model can then be used to generate 10 probabilities that the image is of 10 different digits (0 to 9). The neural network learns to predict accurate predictions through a process called training. Data pre-labeled by humans is used to supervise the training. The model is given input data and is asked to predict values as close to the human label as possible. The model does this by optimizing a number of learnable parameters at every layer in the neural network. The human label is called the ground truth of the data. [1]

Artificial neural networks, or more specifically convolutional neural networks, were popularized 2012 when the model was used to win the annual ImageNet Large-Scale Visual Recognition Challenge, beating all current machine learning models. Today they have achieved state of the art results in areas such as self-driving cars, image classification, object localization, automatic image annotation, semantic segmentation of objects in images and natural language processing. [1]

1.2 Purpose

When I first started out in the field of deep learning I found that there was a lack of fully derived explanations of the underlying mathematics behind artificial neural networks. Sources had either only explained one simple forward pass through the network, or had only derived the most simple

case, ignoring the more complicated general cases. The aim of this paper is to present a clear derivation of the general case for feed-forward neural networks and convolutional neural networks. Furthermore, the aim is to apply the derived model of the convolutional neural network to two problems: Classifying handwritten digits and to detect and pinpoint the location of a variable number of human faces in an image.

1.3 Problem statement

What is a feed-forward neural network? What is a convolutional neural network? What is forward and backpropgation and how is it derived in a feed-forward neural network and in a convolutional neural network? Can an artificial neural network be trained to classify handwritten digits and detect a variable number of faces in images?

2 Method

A major part of the paper and the derivations of the models are based on the material from Stanford's course "CS231n: Convolutional neural networks for Visual Recognition". Some advanced concepts were directly based off the papers they were first introduced in (e.g. Batch Normalization) and expanded to fit the general case of an arbitrary input to the model. The general case for the mathematical model of the feed-forward neural network and the convolutional neural network were derived by hand with the help of the simple cases. The models were implemented in C++ with the linear algebra library Eigen and in Python in pure NumPy, a library for scientific computing. The derived partial derivatives required by the models were compared to their numerical approximations using the formal definition of a derivative. [1] [2] [3] [4]

When my own implementation of the artificial neural networks became too computationally expensive and inefficient for the two problem cases of classifying handwritten digits and dense face detection, the models were re-implemented in Google's machine learning library TensorFlow and Facebook's GPU-accelerated tensor and dynamic neural network library PyTorch. [5] [6]

3 Feed-forward neural networks

A feed-forward neural network is the most elementary version of an artificial neural network. As all varying kinds of artificial neural networks, a feed-forward neural network is made out of a number of layers of neurons. The unique property of a feed-forward neural network is that all the neurons in a layer are connected to every neuron in the next layer (see figure 2). [1]

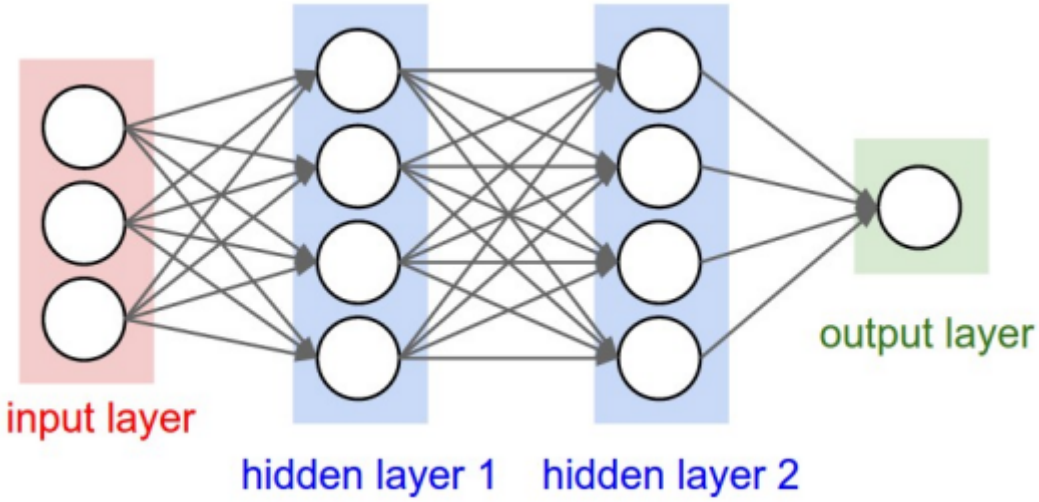


Figure 2: An illustration of a simple feed-forward neural network. It consists of 4 layers: 1 input layer (red), 2 hidden layers (blue) and 1 output layer (green). A circle represents a neuron. Every neuron in a layer is connected to all the neurons in the following layer, shown by the grey lines between the neurons. [7]

A neuron is represented by a floating point decimal number. The value of a neuron is called its activation. Given an input of n data points the model wants to predict m different values. n and m are set as the size and number of neurons in the input and output layer respectively. The values of the input neurons are transferred to the next layer depending on the strength of the connection between every pair of neurons in two adjacent layers. This is done for every layer until the signal has reached the output layer. The process of propagating the value of the neurons from the input layer to the output layer is called forward propagation. [1]

The network is trained to predict correct values by optimizing the fixed connections, also called weights, between every pair of neurons in 2 consecutive layers in the neural network. [1]

3.1 Tensors, indexing and notation

A tensor is the generalization of vectors and matrices. Scalars are tensors of order 0. A tensor of order 1 is a row vector $x \in \mathbb{R}^N$ with N elements. It can also be seen as a one-dimensional array. Matrices M are tensors of order 2 such that $M \in \mathbb{R}^{R \times N}$ and can be viewed as vectors with R elements where every element is another vector with N scalar elements. Matrices can also be seen as a two-dimensional array with RN elements. A tensor of order n is an n-dimensional array and is indexed by an n-tuple. For instance, a tensor $X \in \mathbb{R}^{R \times C \times H \times W}$ is indexed by the four-tuple (r, c, h, w) where $1 \leq r \leq R$, $1 \leq c \leq C$, $1 \leq h \leq H$ and $1 \leq w \leq W$. [1]

3.2 Forward propagation

Figure 2 only illustrates a single set of input neurons, called a training example, being forward propagated. In practice a mini-batch of R training examples is constructed and forward propagated at the same time. [1] [8]

Let L denote the number of layers in the neural network and l, $1 \leq l \leq L$, one specific layer in the neural network. Let $N^{(l)}$ denote the number of neurons in layer l. The activation (values) of layer l can be expressed as a tensor of order 2: $X^{(l)} \in \mathbb{R}^{R \times N^{(l)}}$ indexed by the two-tuple (r, i) where $1 \leq r \leq R$ and $1 \leq i \leq N^{(l)}$. In addition to the neurons of layer l, there is a bias neuron $b^{(l)} \in \mathbb{R}$ (see figure 3). It is called a bias neuron because its value is independent to what input the neural network is given. [1] [8]

The weights representing the strength of the connections between the neurons of layer l and $l+1$ are also expressed as a tensor of order 2. Let $W^{(l)} \in \mathbb{R}^{N^{(l+1)} \times N^{(l)}}$ such that the element $W^{(l)}_{j,i}$ is the strength of the connection between neuron $X^{(l)}_{ri}$ and $X^{(l+1)}_{rj}$ for arbitrary example r in the mini-batch. [1] [8]

The calculation of the value of a neuron in layer $l+1$ includes calculating the sum of every neuron in layer l multiplied with its corresponding weight in $W^{(l)}$, plus the value of the bias $b^{(l)}$. The sum is then put in a so called activation function f. The value of the activation function is the neuron's activation in layer $l+1$. [1] [8]

Let $Z^{(l)} \in \mathbb{R}^{R \times N^{(l)}}$ be the value of each neuron in layer l before being put into the activation function. Given an input $X^{(1)}$ the forward propagation is expressed recursively as: [1] [8]

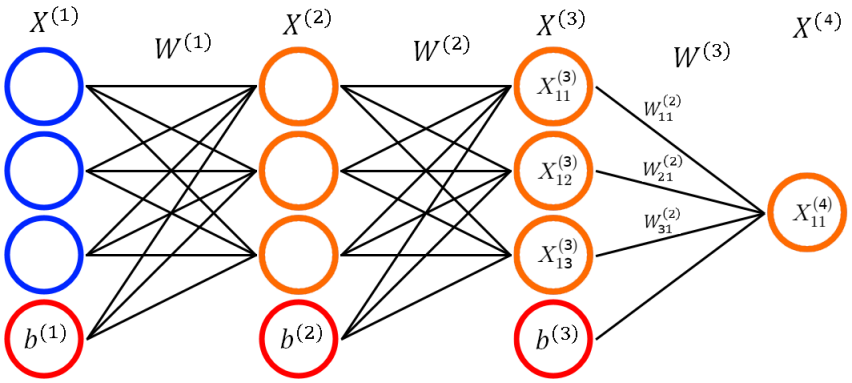


Figure 3: An example of a 4 layer feed-forward neural network. The input neurons are marked with blue. The bias neurons are marked with red. Black lines between neurons symbolyze the weights between every pair of neurons between two layers.

$$Z_{rj}^{(l+1)} = b^{(l)} + \sum_{i=1}^{N^{(l)}} X_{r,i}^{(l)} W_{i,j}^{(l)} \tag{1}$$

$$X_{rj}^{(l+1)} = f(Z_{rj}^{(l+1)}) \tag{2}$$

The tensor of activations and weights are constructed in such a way that one forward pass through a single layer can be computed with a single dot product and an addition of the bias term added to every element. The activation function f is then applied element-wise on each neuron: [1] [8]

$$Z_{rj}^{(l+1)} = \left[X^{(l)} W^{(l)}\right]_{rj} + b^{(l)} \tag{3}$$

$$X_{rj}^{(l+1)} = f(Z_{rj}^{(l+1)}) \tag{4}$$

Common activation functions are Rectified Linear Units (ReLU), sigmoid (σ) and hyperbolic tangent (tanh) and are defined by equations (5) - (7) (see figure 4). A non-linear activation function is chosen to enable the network to make use of non-linearities when learning to predict values. Without non-linear activation functions, the whole model is equivalent to one large linear transformation of the input data. [1]

$$\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{5}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

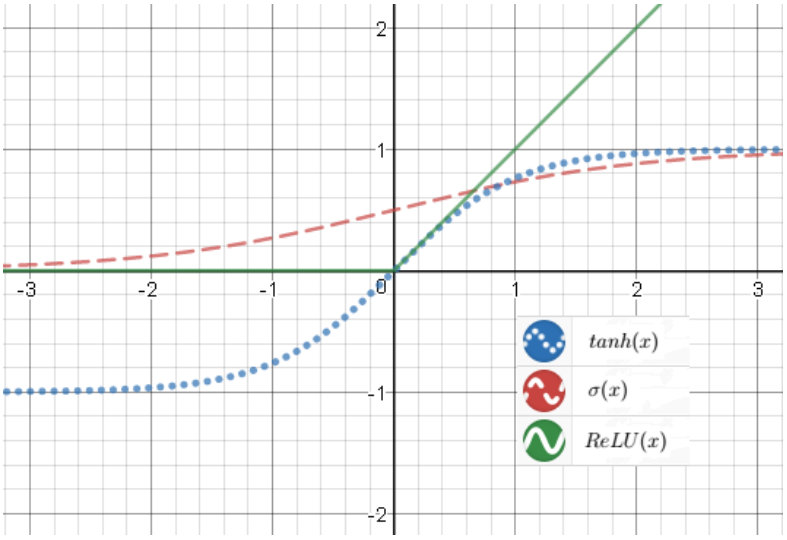


Figure 4: A graph of ReLU, σ och tanh.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{7}$$

3.3 Loss function

Given an input X and a ground truth y the model wants to predict values $\hat{y}$ which resembles the ground truth as closely as possible. It is achieved by defining a multivariate loss function $L(\theta; X, y)$ with the network's parameters θ (all the weights and biases) with respect to a single training example (X, y). The loss function describes the quality of the neural network's prediction $\hat{y}$ such that a lower loss represents a more accurate prediction. $\hat{y}$ is the activations at the output layer and y is the human labeled ground truth with the same dimensionality as $\hat{y}$. $L(\theta)$ is used to the loss function with respect to one mini-batch of training data. One way of defining the loss function is to use the mean squared error defined by the following equation: [1]

$$L(\theta) = \frac{1}{RN} \sum_{r=1}^{R} \sum_{i=1}^{N} (\hat{y}_{r,i} - y_{r,i})^2 \tag{8}$$

Here R is the batch size and N is the number of neurons in the last layer.

The process of minimizing the loss function is called training. The model iterates through supervised training data and calculates the loss with respect to a mini-batch of training examples. Since the given input X and ground

truth y stays fixed, the network learns by optimizing its weights $W^{(l)}$ and biases $b^{(l)}$ for every layer l to decrease the loss. [1] [8]

3.4 Gradient Descent

The gradient $\nabla L(\theta)$ is a vector of partial derivatives with respect to the parameters θ of the function L defined by the following equations: [9] [10]

$$\nabla L(\theta) : \mathbb{R}^n \to \mathbb{R}^n \tag{9}$$

$$\nabla L(\theta) = \left(\frac{\partial L(\theta)}{\partial \theta_1}, \ \frac{\partial L(\theta)}{\partial \theta_2}, \ \cdots, \ \frac{\partial L(\theta)}{\partial \theta_n} \right) \tag{10}$$

The gradient $\nabla L(\theta)$ shows the direction of steepest ascent in the point (θ_1, θ_2, ..., θ_n) in the n-dimensional vector space $\mathbb{R}^n$. For a function $f(x)$ of a single variable x, the gradient is simply the derivative of the function with respect to x and is the slope of the tangent line to f at x. For a function $f(x, y)$ of two variables x and y, the gradient is the two-dimensional vector of the slope in the x dimension and y dimension respectively. The loss function used in neural networks can be a function of millions of parameters, depending on the depth and size of the neural network. [9] [10]

Gradient descent is the method of iteratively changing the values of the parameters θ proportionally to the negative gradient $-\nabla L(\theta)$ to minimize the function $L(\theta)$ (see figure 5). The most basic algorithm of gradient descent is called Stochastic Gradient Descent (SGD) and uses the hyperparameter α, called learning rate, to control the magnitude of the gradient. It is called Stochastic Gradient Descent because, at each iteration, the mini-batch responsible for the losses is randomly chosen and sampled from the whole batch of training data. Stochastic Gradient Descent is defined by the following equations: [9] [10] [8]

$$\frac{\partial L(\theta)}{\partial \theta_i} = \nabla_{\theta_i} L(\theta) \tag{11}$$

$$\theta_i \to \theta_i - \alpha \frac{\partial L(\theta)}{\partial \theta_i} \tag{12}$$

3.5 Backpropagation

Backpropagation is the process of calculating the partial derivatives of the loss function with respect to the model's parameters, or in other terms, the

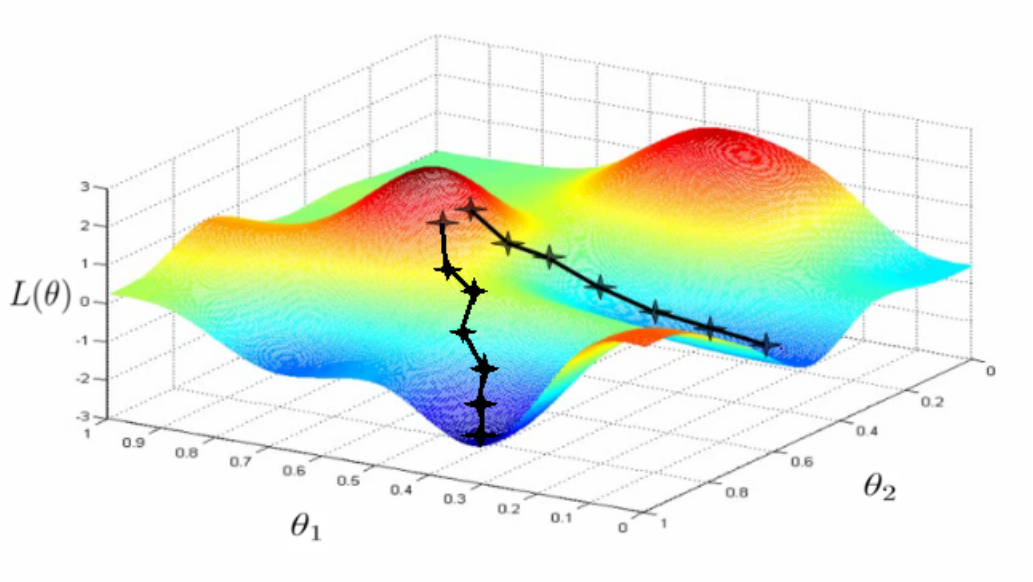


Figure 5: An illustration of Gradient Descent on a function of two variables. Red regions symbolizes a high function value while blue regions symbolizes a low function value. The parameters were initialized near the global maximum and their values are altered iteratively to move in the direction of the negative gradient: the direction of steepest descent, to find a local minimum. [11]

process of computing the gradient. [8] [9]

The partial derivatives can be approximated numerically with the formal definition of a derivative defined by equation (13): [8] [9]

$$\frac{\partial L(\theta)}{\partial \theta_i} = \lim_{h \to 0} \frac{L(\theta_1, ..., \theta_i + h, ..., \theta_n) - L(\theta)}{h} \tag{13}$$

This would not be a problem for the small neural network in figure 3 with a total of 24 parameters, but would be extremely inefficient for deep neural networks with millions of parameters. Instead, the chain rule is applied to calculate the precise value of the partial derivatives. [1]

Let $\delta^{(l)}$ denote the so called delta-error at layer l, defined by equation (14):

$$\delta^{(l)} = \frac{\partial L(\theta)}{\partial X^{(l)}} \tag{14}$$

The delta-error is the partial derivative of the loss with respect to a specific neuron in the model and is needed to efficiently compute the gradient with the help of the chain rule. The delta-error can be interpreted as how much a deviation in the value of a neuron affects the loss function. A change in value of a neuron with a big delta-error results in a greater change of the total loss compared to a neuron with a small delta-error. Because the activation of

layer $l+1$ is a function of the previous layer l, the delta-error can be computed recursively from the output layer and propagated backwards to the first layer in the network. By applying the chain rule the delta-error $\frac{\partial L(\theta)}{\partial X^{(l)}}$ at layer l can be broken up into three partial derivatives $\frac{\partial L(\theta)}{\partial X^{(l+1)}}$, $\frac{\partial X^{(l+1)}}{\partial Z^{(l+1)}}$ and $\frac{\partial Z^{(l+1)}}{\partial X^{(l)}}$. Since a single neuron in layer l is connected to every neuron in layer $l+1$ you have to sum over every neuron in layer $l+1$. The first partial derivative is the delta-error of the next layer and the other two partial derivatives can be derived from equations (1) and (2) and are easily differentiable since (1) is a linear equation: [1] [8]

$$\begin{aligned}
\delta_{r,i}^{(l)} &= \frac{\partial L(\theta)}{\partial X_{r,i}^{(l)}} \\
&= \sum_{j=1}^{N^{(l+1)}} \frac{\partial L(\theta)}{\partial X_{r,j}^{(l+1)}} \frac{\partial X_{r,j}^{(l+1)}}{\partial Z_{r,j}^{(l+1)}} \frac{\partial Z_{r,j}^{(l+1)}}{\partial X_{r,i}^{(l)}} \\
&= \sum_{j=1}^{N^{(l+1)}} \delta_{r,j}^{(l+1)} f'(Z_{r,i}^{(l+1)})\ W_{j,i}^{(l)}
\end{aligned} \tag{15}$$

The delta-error of the last layer L depends on the type of loss function used. For the mean squared error the delta-error of the output layer is defined by the following equation: [1] [8]

$$\begin{aligned}
\delta^{(L)} &= \frac{\partial L(\theta)}{\partial \hat{y}} \\
&= \frac{2}{RN^{(L)}}(\hat{y} - y)
\end{aligned} \tag{16}$$

With the delta-error defined at every layer in the network the partial derivative of the loss function with respect to the networks parameters can be calculated. Just like equation (15), the chain rule is applied to break up $\frac{\partial L(\theta)}{\partial X^{(l)}}$ into three partial derivatives, $\frac{\partial L(\theta)}{\partial X^{(l+1)}}$, $\frac{\partial X^{(l+1)}}{\partial Z^{(l+1)}}$ and $\frac{\partial Z^{(l+1)}}{\partial W^{(l)}}$. All the examples in the mini-batch are summed over since the weights affect every single example. $\frac{\partial L(\theta)}{\partial X^{(l+1)}}$ is the delta-error and the other two partial derivatives are derived from equations (1) and (2): [1] [8]

$$\begin{aligned}\frac{\partial L(\theta)}{\partial W_{j,i}^{(l)}} &= \sum_{r=1}^{R} \frac{\partial L(\theta)}{\partial X_{r,i}^{(l+1)}} \frac{\partial X_{r,i}^{(l+1)}}{\partial Z_{r,i}^{(l+1)}} \frac{\partial Z_{r,i}^{(l+1)}}{\partial W_{i,j}^{(l)}} \\ &= \sum_{r=1}^{R} \delta_{r,i}^{(l+1)} f'(Z_{r,i}^{(l+1)})\ X_{r,j}^{(l)}\end{aligned} \tag{17}$$

The partial derivative of the loss function with respect to the biases are found in a similar way to equations (15) and (17): [1] [8]

$$\begin{aligned}\frac{\partial L(\theta)}{\partial b^{(l)}} &= \sum_{r=1}^{R} \frac{\partial L(\theta)}{\partial X_{r,i}^{(l+1)}} \frac{\partial X_{r,i}^{(l+1)}}{\partial Z_{r,i}^{(l+1)}} \frac{\partial Z_{r,i}^{(l+1)}}{\partial b_{i,j}^{(l)}} \\ &= \sum_{r=1}^{R} \delta_{r,i}^{(l+1)} f'(Z_{r,i}^{(l+1)})\end{aligned} \tag{18}$$

3.6 Training neural networks

The model is trained by dividing the training data into mini-batches of size R. The mini-batch is then forward propagated and the loss is calculated. The loss is then used to backpropagate the models error layer by layer. Backpropagation starts at the output layer propagates through the whole network backwards until it reaches the input layer. For every layer in the network the delta-error from the proceeding layer is used to calculate the new delta-error. It is then used to calculate the layers contribution to the total gradient by computing the partial derivatives. When the gradient has been fully computed one iteration of the gradient descent algorithm is applied to update the weights and biases of the neural network. This process is repeated until all the networks parameters have converged.[1]

Two implementations of a feed-forward neural network can be found on github in Python and C++ in the repositories *neural-network-python* and *neural-network-cpp* respectively: `https://github.com/nikitazozoulenko`.

4 Convolutional neural networks

When humans identify objects by sight we look for specific high level features that object has. A cat for example has one head, four legs and a body. These high level features are in turn made up of a combination of low level

features: A head consists of two eyes and a mouth which consists of elementary geometric shapes which are composed of a combination of basic lines and edges. In addition to specific features, cats also have a furry texture. Convolutional neural networks (CNN) were designed to specifically excel at computer vision tasks. What a convolutional neural network does is that it learns these hierarchical structures by teaching itself a number of filters to apply to the image (see figure 6) through an operation called a convolution. These filters are stacked on top of each other in terms of layers and allows the networks to learn higher level features with the network architecture going deeper. For instance, the first layer might learn how to detect lines and edges, the middle layers might learn to recognize small body parts such as eyes and ears and the last layers can learn how to detect cats, humans or other arbitrary objects. [1]

the networks to learn higher level features with the network architecture going deeper

Figure 6: The result of a filter for vertical and horizontal edge detection applied on a picture of a cat.

In feed-forward neural networks every neuron in a layer is connected to all the neuron in the next layer. If you would use feed-forward neural networks for a computer vision problem you would reshape the given input image to a single CHW-dimensional vector where W and H are the images width and height in pixels and C is the number of color channels in the image. Because every neuron in the previous layer is connected to all the neurons in the proceeding layer most of the spatial information in the image is lost. Convolutional neural networks are different in the way how they operate on spatially local data. The neurons in layer l are only connected to the neurons in layer $l + 1$ that are in the close spatial vicinity of the neurons in layer l. In practice this has yielded more accurate predictions over their feed-forward counterparts and has lead to the model becoming the state of

the art in computer vision. [1] [10] [12]

4.1 Model structure, parameters and notation

Convolutional neural networks are composed of a number of layers stacked on top of each other, similar to feed-forward neural networks. The difference between these two models is how one layer is connected to the next layer. While feed-forward neural networks have only one type of layer, convolutional neural networks have a wide variety of different layers. The five elementary layers and operations of the convolutional neural network are the convolutional layer, the maxpooling layer, the softmax layer, the activation function layer and batch normalization, which all behave differently. A convolutional neural network can also use fully connected layers where every neuron in a layer l is connected to all neurons in layer $l+1$, which behave exactly the same as feed-forward neural networks. [1] [10] [12]

At every layer l there are parameters $\theta^{(l)}$ and activations (neurons) $X^{(l)}$. The last layer is characterized by $\hat{y}$ and $X^{(L)}$ where L is the number of layers in the network. Given an input $X^{(1)}$ the model predicts values $\hat{y}$. The input is forward propagated recursively by using the activations $X^{(l)}$ and parameters $\theta^{(l)}$ from layer l to compute the activations in layer $l+1$, defined by equation (19). Simiar to feed-forward neural network, the prediction is then put into a loss function $L(\theta)$, which is used by the backpropagation algorithm along with Stochastic Gradient Descent to train the network. [1] [10]

$$X^{(1)} \xrightarrow{\theta^{(1)}} X^{(2)} \xrightarrow{\theta^{(2)}} \cdots \xrightarrow{\theta^{(L-2)}} X^{(L-1)} \xrightarrow{\theta^{(L-1)}} X^{(L)} = \hat{y} \tag{19}$$

To be able to fully utilize the spatial information from a given input the model represent the activation at a given layer l as a tensor of order 4: $X^{(l)} \in \mathbb{R}^{R \times C \times H \times W}$. A layer takes in a batch of three-dimensional volumes of neurons and produces a new batch of three-dimensional volumes of neurons at the next layer. R stands for the batch size and C, W and H are the depth, width and and height of the volume of neurons. To feed an image into the model, the image is made into a tensor of size $3 \times H \times W$ where the values of the neurons are the value of the pixels in the image, for all 3 red, green and blue color channels in the image. The image tensors are then stacked into a single tensor of size $R \times 3 \times H \times W$ to form a mini-batch. [1]

A $H \times W$ slice of the activations is called a feature map or a channel.

Convolutional neural networks are usually illustrated as three-dimensional volumes of activations (see figure 7) or as stacked feature maps (see figure 8). [1] [10] [12]

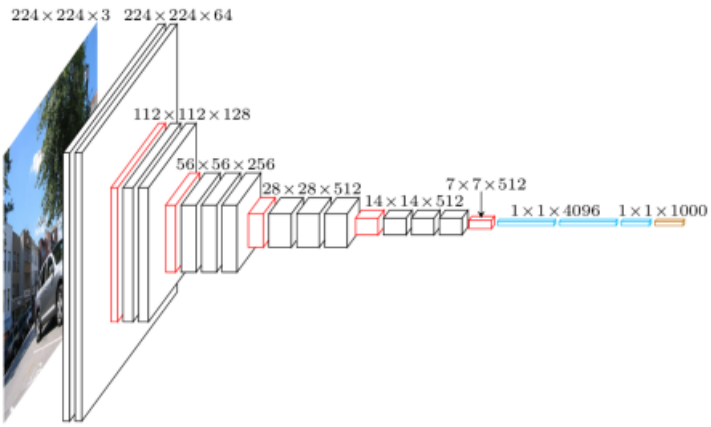


Figure 7: An illustration of a the VGGNet19 convolutional neural network. The different volumes represent the activations of the network at different depths. Each two-dimensional slice is a feature map. [13]

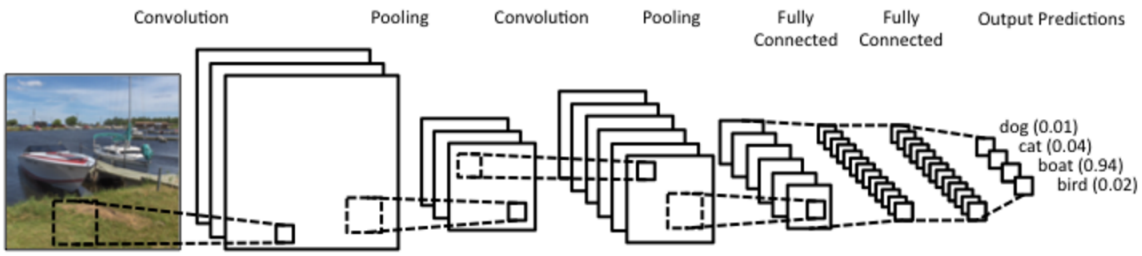


Figure 8: An illustration of a convolutional neural network. Each two-dimensional slice is a feature map. [14]

Every single type of layer in a convolutional neural network has two modes: forward propagation and backpropagation. The model is trained the same way as a feed-forward neural network is trained: A given input is forward propagated to get a loss which is then backpropageted though the network. Thus, for every layer there has to be a definition of forward propagation and backpropagation. In the forward propagation the activations from the previous layer are used to calculate the activations in the next layer. During backpropagation the delta-error from the next layer is used to calculate the delta-error at the previous layer. The delta-error is then used to compute the partial derivatives needed for the gradient. [1] [10]

Because every layer operation in the neural network is defined sequentially, we only have to define operations for two adjacent layers. Let the size of the tensor of activations at layer l have size $R \times C \times H \times W$. At the next layer,

$l+1$, the activations will be of size $R \times C' \times H' \times W'$. The prime notation specifies that the variable originates from the proceeding layer and is the same for the indices. The tensors are indexed by the four-tuple (r, c, h, w). The batch size always stays the same from layer to layer, while the spatial size can decrease depending on the layer type. [1] [10]

4.2 Convolution forward propagation

The basic building block of the convolutional neural network is the convolution and the convolutional layer. It uses the learnable parameters $W^{(l)} \in \mathbb{R}^{C' \times C \times k \times k}$, called a kernel, and is the before mentioned filter the network learns to apply to images. C and C' are the number of channels in layer l and $l+1$ respectively. k is a variable called the kernel size of the convolution. [1]

The activations at layer $l+1$ are derived from the activations at layer l by applying the kernel at every possible spatial location on the activations at layer l (see figure 9). Every neuron is multiplied by the value of the kernel at the same spatial location. The sum of all the products become the activation of a single neuron in layer $l+1$. Applying this operation on every region of neurons in the activation is called a convolution. The convolution operator is denoted by $*$. [1] [10] [12]

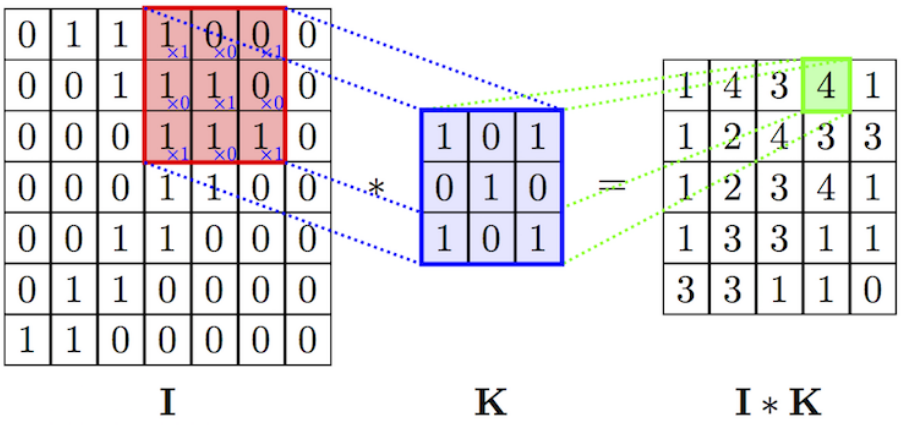


Figure 9: A kernel of size $1 \times 3 \times 3$ convolving over activations of size $1 \times 7 \times 7$, producing activations of size $1 \times 4 \times 4$ in the next layer. [15]

A feature map in layer $l+1$ is the result of a single kernel of size $1 \times C \times k \times k$ being convolved over the whole activation volume of the previous layer. C' is the number of kernels a layer has and is also the number of feature maps the next layer will have. [1] [10]

The kernels have two additional non-learnable hyperparameters: a stride s

and zero-padding p. s is the size of the step the kernel takes when it moves from one spatial location to the next during a convolution. The convolution in figure 9 has a stride of $s = 1$. Zero-padding is when you pad the edges of the activation tensor with p zeros (see figure 10). Since a convolution decreases the spatial size of the activations of the next layer, zero-padding is a way to control the size of the activations. [1] [10] [12]

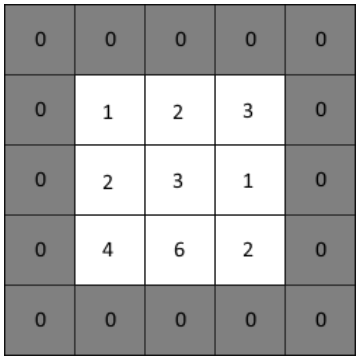


Figure 10: An activation with size $1 \times 3 \times 3$ is zero-padded with $p = 1$ and the resulting tensor is of size $1 \times 5 \times 5$.

Let $W^{(l)} \in \mathbb{R}^{C' \times C \times K_h \times K_W}$, $X^{(l)} \in \mathbb{R}^{R \times C \times (H+2p) \times (W+2p)}$ and $X^{(l+1)} \in \mathbb{R}^{R \times C' \times H' \times W'}$. The dimensions of layer $l+1$ are defiend by equations (20) and (21): [1] [10] [12]

$$W' = \frac{W - K_W + 2p}{s} + 1 \tag{20}$$

$$H' = \frac{H - K_H + 2p}{s} + 1 \tag{21}$$

Mathematically the convolutional layer is defined by the following equations: [1] [10]

$$w = sw' \tag{22}$$

$$h = sh' \tag{23}$$

$$\begin{aligned} \left[X^{(l+1)}\right]_{r,c',h',w'} &= X^{(l)}_{r,c',h',w'} * W^{(l)}_{c'} \\ &= \sum_{c=1}^{C} \sum_{j=1}^{K_H} \sum_{i=1}^{K_W} X^{(l)}_{r,c,h'+j,w'+i} W^{(l)}_{c',c,j,i} \end{aligned} \tag{24}$$

The index of the term which shall be used to convolve specifies which dimensions will be convolved upon and summed over. For instance, $W^{(l)}_{c'}$ implies that the C, H and W dimensions (all channels) should be convolved, while $W^{(l)}_{c',c}$ implies that only the H and W dimension (one channel) should be convolved.

Convolutions are in practice implemented with the functions $row2im$ and $im2row$ which enable the convolution to be computed with a single dot product. The underlying math is equivalent with the equations shown in this paper. However, $row2im$ and $im2row$ are outside of the scope of this paper and are left to the reader to research if a more computationally efficient implementation is required. [1] [10] [12]

4.3 Convolution backpropagation

At every layer l the delta-error of the proceeding layer $\delta^{(l+1)}$ has to be back-propagated to create the delta-error at the current layer. $\delta^{(l)}$ is then used to compute the partial derivatives of the loss with respect to the weights $W^{(l)}$ to be used in the gradient. [1] [10]

The backpropagation of the recursive delta-error $\delta^{(l+1)}$ is derived by the use of the chain rule. $\delta^{(l+1)} = \frac{\partial L(\theta)}{\partial X^{(l+1)}}$ is broken up into two smaller partial derivatives $\frac{\partial L(\theta)}{\partial X^{(l+1)}}$ and $\frac{\partial X^{(l+1)}}{\partial X^{(l)}}$. Additionally, because more than one single neuron in layer l is responsible for the delta-error at layer $l+1$, all the neurons of layer l have to be summed over, similar to equation (17), (18) and (15). This can be done since the derivative of a sum is equivalent to the sum of the derivatives of each element. $X^{(l+1)}_{r,c',h',w'}$ is then replaced by its definition from equation (24): [10] [16] [17] [18]

$$\begin{aligned}
\delta^{(l)}_{r,c,h,w} &= \frac{\partial L(\theta)}{\partial X^{(l)}_{r,c,h,w}} \\
&= \sum_{c'=1}^{C'} \sum_{h'=1}^{H'} \sum_{w'=1}^{W'} \frac{\partial L(\theta)}{\partial X^{(l+1)}_{r,c',h',w'}} \frac{\partial X^{(l+1)}_{r,c',h',w'}}{\partial X^{(l)}_{r,c,h,w}} \\
&= \sum_{c'=1}^{C'} \sum_{h'=1}^{H'} \sum_{w'=1}^{W'} \delta^{(l+1)}_{r,c',h',w'} \frac{\partial \sum_{c=1}^{C} \sum_{j=1}^{k_H} \sum_{i=1}^{k_W} X^{(l)}_{r,c,h'+j,w'+i} W^{(l+1)}_{c',c,j,i}}{\partial X^{(l)}_{r,c,h,w}}
\end{aligned} \tag{25}$$

Every partial derivative in the most inner sum will be equal to to zero if $X^{(l)}_{r,c,h'+j,w'+i} \neq X^{(l)}_{r,c,h,w}$. Using the substitutions $h = h' + j$ and $w = w' + i$ the three inner sums are cancelled out: [16] [17] [18]

$$\sum_{c'}^{C'}\sum_{h'=1}^{H'}\sum_{w'=1}^{W'}\delta^{(l+1)}_{r,c',h',w'}\frac{\partial\sum_{c=1}^{C}\sum_{j=1}^{K_H}\sum_{i=1}^{K_W}X^{(l)}_{r,c,h'+j,w'+i}W^{(l+1)}_{c',c,j,i}}{\partial X^{(l)}_{r,c,h,w}}$$
$$=\sum_{c'=1}^{C'}\sum_{h'=1}^{H'}\sum_{w'=1}^{W'}\delta^{(l+1)}_{r,c',h',w'}W^{(l+1)}_{c',c,(h-h'),(w-w')} \tag{26}$$

Which you can see is a sum of convolutions where a feature map of the delta-error of layer $l+1$ convolves over all the kernels of layer l where the kernels are rotated by 180°. This is intuitive since every feature map in $X^{(l)}$ is used to create a single feature map in $X^{(l+1)}$. Let the rotation of the kernel be denoted with the function $rot()$. The final equation of the backpropagation of the delta-error is defined by equation (27): [16] [17] [18]

$$\delta^{(l)}_{r,c,h,w}=\sum_{c'=1}^{C'}rot(W^{(l+1)}_{c',c,h,w})*\delta^{(l+1)}_{r,c'} \tag{27}$$

The partial derivative of the loss $L(\theta)$ with respect to the weights $L(\theta)$ is derived the same way the backpropagation of the error is derived. The only change is that the R-dimension is summed over since every example in the mini-batch affects the gradient. [1] [16] [17] [18]

$$\begin{aligned}\frac{\partial L(\theta)}{\partial W^{(l)}_{c',c,h,w}}&=\sum_{r=1}^{R}\sum_{c'=1}^{C'}\sum_{h'=1}^{H'}\sum_{w'=1}^{W'}\frac{\partial L(\theta)}{\partial X^{(l+1)}_{r,c',h',w'}}\frac{\partial X^{(l+1)}_{r,c',h',w'}}{\partial W^{(l)}_{r,c,h,w}}\\&=\sum_{r=1}^{R}\sum_{c'=1}^{C'}\sum_{h'=1}^{H'}\sum_{w'=1}^{W'}\delta^{(l+1)}_{r,c',h',w'}\frac{\partial\sum_{c=1}^{C}\sum_{j=1}^{K_H}\sum_{i=1}^{K_W}X^{(l)}_{r,c,h'+j,w'+i}W^{(l)}_{c',c,j,i}}{\partial W^{(l)}_{c',c,h,w}}\\&=\sum_{r=1}^{R}\sum_{c'=1}^{C'}\sum_{h'=1}^{H'}\sum_{w'=1}^{W'}X^{(l)}_{r,c,h'+h,w'+w}\delta^{(l+1)}_{r,c',h',w'}\\&=\sum_{r=1}^{R}\sum_{c'=1}^{C'}X^{(l)}_{r,c,h,w}*\delta^{(l+1)}_{r,c'}\end{aligned} \tag{28}$$

4.4 Activation function forward propagation

In the activation function layer an activation function f is applied element wise on every neuron in the activation tensor. Thus, the size of $X^{(l)}$ and

$X^{(l+1)}$ is the same. Any differentiable function can be used as an activation function, but the most commonly used ones are ReLU, sigmoid and tanh. The activation function layer does not have any learnable parameters. [10]

Forward propagation is defined by equation (29):

$$X^{(l+1)}_{r,c,h,w} = f(X^{(l)}_{r,c,h,w}) \tag{29}$$

Activation functions enable the network to learn faster while also increasing the accuracy of the predictions. [1]

4.5 Activation function backpropagation

Because the activation function layer does not have any learnable parameters only the recursive delta-error has to be backpropagated. It is derived by the use of the chain rule. $\frac{\partial L(\theta)}{\partial X^{(l)}_{r,c,h,w}}$ is split up into $\frac{\partial L(\theta)}{\partial X^{(l+1)}_{r,c,h,w}}$ and $\frac{\partial X^{(l+1)}_{r,c,h,w}}{\partial X^{(l)}_{r,c,h,w}}$. The first term is simply the delta-error of the proceeding layer and the second term is the derivative of the activation function: [1] [10]

$$\begin{aligned}
\delta^{(l)}_{r,c,h,w} &= \frac{\partial L(\theta)}{\partial X^{(l)}_{r,c,h,w}} \\
&= \frac{\partial L(\theta)}{\partial X^{(l+1)}_{r,c,h,w}} \frac{\partial X^{(l+1)}_{r,c,h,w}}{\partial X^{(l)}_{r,c,h,w}} \\
&= \delta^{(l+1)}_{r,c,h,w} f'(X^{(l)}_{r,c,h,w})
\end{aligned} \tag{30}$$

4.6 Maxpooling forward propagation

Maxpooling is a way to reduce the spatial size of the activations from one layer to the next. Every feature map in layer l is divided into a number of regions of size $k \times k$ where k is the hyperparameter called kernel size. One single $k \times k$ region corresponds to a single activation in the proceeding layer $l + 1$. The activation is given by the maximum value of the region (see figure 11). Additionally, maxpooling also has the hyperparameter s, called its stride, and works similar to the stride for the convolutional layer. s denotes the step size the $k \times k$ region uses when it traverses the volume of activations. Maxpooling does not have any learnable parameters. [1] [10] [12]

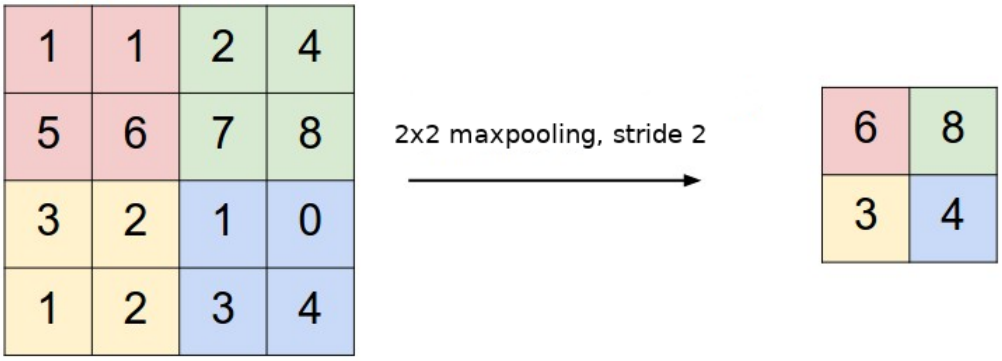


Figure 11: Maxpooling with kernel size $k = 2$ and stride $s = 2$ on an area of size 4×4. The resulting area has size 2×2 [14]

Similar to a convolution without zero-padding, the dimensions of the proceeding layer is defined by the following equations. The number of feature maps remain constant: [1] [10] [12]

$$W' = \frac{W - k}{s} + 1 \tag{31}$$

$$H' = \frac{H - k}{s} + 1 \tag{32}$$

$$C' = C \tag{33}$$

Equation (34) defines the maxpooling layer algebraically. [1] [10]

$$X^{(l+1)}_{r,c',h',w'} = \max_{0 \leq j < k,\ 0 \leq i < k} X^{(l)}_{r,c',(h's+j),(w's+i)} \tag{34}$$

4.7 Maxpooling backpropagation

Maxpooling does not have any learnable parameters and thus only the recursive delta-error has to be backpropagated. With the use of the chain rule the delta-error at layer l is split into two partial derivatives $\frac{\partial L(\theta)}{\partial X^{(l+1)}}$ and $\frac{\partial X^{(l+1)}}{\partial X^{(l)}}$. The first term is the delta-error at layer $l + 1$. $X^{(l+1)}$ is then substituted with its definition from equation (34): [1] [10] [18]

$$
\begin{aligned}
\delta^{(l)}_{r,c,h,w} &= \frac{\partial L(\theta)}{\partial X^{(l)}_{r,c,h,w}} \\
&= \frac{\partial L(\theta)}{\partial X^{(l+1)}_{r,c',h',w'}} \frac{\partial X^{(l+1)}_{r,c',h',w'}}{\partial X^{(l)}_{r,c,h,w}} \\
&= \delta_{r,c',h',w'} \frac{\partial \max\limits_{0 \leq j < k, 0 \leq i < k} X^{(l)}_{r,c',(h's+j),(w's+i)}}{\partial X^{(l)}_{r,c,h,w}}
\end{aligned}
\tag{35}
$$

The partial derivative in the last equation will be equal to 1 if and only if $X^{(l)}_{r,c',(h's+j),(w's+i)} = X^{(l)}_{r,c,h,w}$. For any other case $X^{(l)}_{r,c,h,w}$ will not have any effect on the neurons in the proceeding layer $l+1$ and the partial derivative will thus be 0:[1] [10] [18]

$$
\delta^{(l)}_{r,c,h,w} = \begin{cases} \delta_{r,c,h',w'} & \text{if } \begin{matrix} h = h's + j, \\ w = w's + i \end{matrix} \\ 0 & \text{otherwise} \end{cases}
\tag{36}
$$

The delta-error from layer $l+1$ is thus redirected to the neuron in layer l that is responsible for the activation which the delta error at layer $l+1$ corresponds to. If a neuron in layer l is responsible for two or more activations in layer $l+1$ its delta error will become the sum of the delta-errors of the activations in question. [1] [10] [18]

4.8 Batch Normalization forwardpropagation

Neural networks are hard to train because of their recursive nature. A small change in the weights of the first layer will have a cascading effect throughout the network: The changed second layer will create a slightly larger deviation in the third layer, and the change in the third layer will have an even bigger effect on the fourth layer, and so on. One small change in the first layers can have a dramatic effect on the final prediction of the neural network. This is called the internal covariate shift in the litterature and is what Batch Normalization (BN) sets out to fix. [1] [2]

Batch Normalization normalizes every feature map by dividing the feature map with the variance of the whole mini-batch's variance at the specified feature map, and subtracting the mean of the whole mini-batch's feature map. The cascading effect of a change in the first layer causing a bigger

change in the last layers will no longer take place since every layer aims to have a variance of 1 and mean of 0. The internal covatiate shift is thus minimized. [1] [2]

To derive the activations at layer $l+1$ the mean and variance of every feature map in layer l has to be computed. Let μ_c and σ_c^2 be the mean and variance of the feature map c defined be equations (37) and (38): [1] [2]

$$\mu_c = \frac{1}{RHW} \sum_{r=1}^{R} \sum_{h=1}^{H} \sum_{w=1}^{W} X_{r,c,h,w}^{(l)} \tag{37}$$

$$\sigma_c^2 = \frac{1}{RHW} \sum_{r=1}^{R} \sum_{h=1}^{H} \sum_{w=1}^{W} (X_{r,c,h,w}^{(l)} - \mu_c)^2 \tag{38}$$

Let $\hat{X}$ denote the normalized activations. It is defined by equation (39). [1] [2]

$$\hat{X}_{r,c,h,w} = (X_{r,c,h,w}^{(l)} - \mu_c)(\sigma_c^2)^{-\frac{1}{2}} \tag{39}$$

The normalized activations are then transformed by an affine transformation with the learnable parameters $\gamma_c^{(l)}$ and $\beta_c^{(l)}$. They enable the network to undo the normalization from equation (39) if the network deems it will result in more accurate predictions. The final activations at layer $l+1$ is defined by equation (40): [1] [2]

$$X_{r,c,h,w}^{(l+1)} = \gamma_c^{(l)} \hat{X}_{r,c,h,w} + \beta_c^{(l)} \tag{40}$$

When the network is used for predictions outside of the training, also called runtime, the network cannot calculate the needed statistics of the mini-batch to perform forward propagation since a batch size of 1 is usually used at runtime. To combat this, the statistics of the whole training data can be used to approximate the mean and variance of the activations. This can be done for small datasets, but is inpractical for training data with millions of examples. Instead, an exponentially weighted moving average which is updated at every forward propagation can be used to approximate the mean and variance of the whole population. [1] [2]

Let μ_{EWMA_c} and $\sigma_{EWMA_c}^2$ denote the exponentially weighted moving average for the mean and variance of the feature map c. Let λ be the weight decay term. The moving averages are then defined by equations (41) and (42):

$$\mu_{EWMA_c} \to \lambda \mu_c + (1-\lambda)\mu_{EWMA_c} \tag{41}$$

$$\sigma_{EWMA_c}^2 \to \lambda \sigma_c^2 + (1-\lambda)\sigma_{EWMA_c}^2 \tag{42}$$

4.9 Batch Normalization backpropagation

At every layer l the delta-error of the previous layer $\delta^{(l+1)}$ has to be back-propagated to create the delta-error at the current layer. The delta-error is then used to compute the partial derivatives of the loss with respect to the learnable parameters $\gamma_c^{(l)}$ and $\beta c^{(l)}$ to be used in the gradient. To aid the derivation of the backpropagation the kronecker-delta I is used. The kronecker-delta has the following properties: [19] [20]

$$I_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{43}$$

$$\frac{\partial a_j}{\partial a_i} = I_{i,j} \tag{44}$$

$$\sum_j a_i I_{i,j} = a_j \tag{45}$$

The backpropagation of the recursive delta-error $\delta^{(l+1)}$ is derived by the use of the chain rule. $\frac{\partial L(\theta)}{\partial X^{(l)}}$ is broken up into three partial derivatives and the sum of all the neurons is taken, similar to equation (25). Additionally, the R-dimension is summed over since every example in the mini-batch has an effect on a single neuron in proceeding layer. [19] [20]

$$\begin{aligned} \delta^{(l)}_{r,c,h,w} &= \frac{\partial L(\theta)}{\partial X^{(l)}_{r,c,h,w}} \\ &= \sum_{r'=1}^{R'} \sum_{c'=1}^{C'} \sum_{h'=1}^{H'} \sum_{w'=1}^{W'} \frac{\partial L(\theta)}{\partial X^{(l+1)}_{r',c',h',w'}} \frac{\partial X^{(l+1)}_{r',c',h',w'}}{\partial \hat{X}_{r',c',h',w'}} \frac{\partial \hat{X}_{r',c',h',w'}}{\partial X^{(l)}_{r,c,h,w}} \end{aligned} \tag{46}$$

$\frac{\partial L(\theta)}{\partial X^{(l+1)}_{r,c,h,w}}$ is the delta-error of the proceeding layer. $\frac{\partial X^{(l+1)}_{r',c',h',w'}}{\partial \hat{X}_{r',c',h',w'}}$ is easily differentiable since the terms have a linear relationship defined by equation (39): [19] [20]

$$\begin{aligned} \frac{\partial X^{(l+1)}_{r',c',h',w'}}{\partial \hat{X}_{r',c',h',w'}} &= \frac{\partial(\gamma^{(l)}_{c'} \hat{X}_{r',c',h',w'} + \beta^{(l)}_{c'})}{\partial \hat{X}_{r',c',h',w'}} \\ &= \gamma^{(l)}_{c'} \end{aligned} \tag{47}$$

The partial derivative of the normalized activations $\hat{X}$ with respect to the activations $X^{(l)}$ is derived by substituting $X^{(l)}$ with its definition from equa-

tion (39) and then using the product rule: [19] [20]

$$\begin{aligned}\frac{\partial \hat{X}_{r',c',h',w'}}{\partial X^{(l)}_{r,c,h,w}} &= \frac{\partial (X^{(l)}_{r',c',h',w'} - \mu_{c'})(\sigma^2_{c'})^{-\frac{1}{2}}}{\partial X^{(l)}_{r,c,h,w}} \\ &= (\sigma^2_{c'})^{-\frac{1}{2}} \frac{\partial (X^{(l)}_{r',c',h',w'} - \mu_{c'})}{\partial X^{(l)}_{r,c,h,w}} - \frac{1}{2}(X^{(l)}_{r',c',h',w'} - \mu_c)(\sigma^2_{c'})^{-\frac{3}{2}} \frac{\partial \sigma^2_{c'}}{\partial X^{(l)}_{r,c,h,w}}\end{aligned} \tag{48}$$

The derivative of the first factor with respect to the activation is derived by substituting the batch mean $\mu_{c'}$ with its definition from equation (37) and then using the kronecker-delta from equations (43), (44) and (45): [19] [20]

$$\begin{aligned}\frac{\partial (X^{(l)}_{r',c',h',w'} - \mu_{c'})}{\partial X^{(l)}_{r,c,h,w}} &= \frac{\partial (X^{(l)}_{r',c',h',w'} - \frac{1}{RHW} \sum_{r''=1}^{R} \sum_{h''=1}^{H} \sum_{w''=1}^{W} X^{(l)}_{r'',c',h'',w''})}{\partial X^{(l)}_{r,c,h,w}} \\ &= I_{r',r} I_{c',c} I_{h',h} I_{w',w} - \frac{1}{RHW} I_{c',c}\end{aligned} \tag{49}$$

The derivative of the second factor with respect to the activation is found in the same way as the previous equation. The batch variance $\sigma^2_{c'}$ is substituted with its definition from equation (38) and then using the kronecker-delta together with the chain rule: [19] [20]

$$\begin{aligned}\frac{\partial \sigma^2_{c'}}{\partial X^{(l)}_{r,c,h,w}} &= \frac{\partial \frac{1}{RHW} \sum_{r'=1}^{R} \sum_{h'=1}^{H} \sum_{w'=1}^{W} (X^{(l)}_{r',c',h',w'} - \mu_{c'})^2}{\partial X^{(l)}_{r,c,h,w}} \\ &= \frac{1}{RHW} \sum_{r'=1}^{R} \sum_{h'=1}^{H} \sum_{w'=1}^{W} 2(X^{(l)}_{r',c',h',w'} - \mu_{c'})(I_{r',r} I_{c',c} I_{h',h} I_{w',w} - \frac{1}{RHW} I_{c',c}) \\ &= \frac{2}{RHW}(X^{(l)}_{r,c',h,w} - \mu_{c'}) I_{c',c} - \frac{2}{(RHW)^2} \sum_{r'=1}^{R} \sum_{h'=1}^{H} \sum_{w'=1}^{W} (X^{(l)}_{r',c,h',w'} - \mu_c) \\ &= \frac{2}{RHW}(X^{(l)}_{r,c',h,w} - \mu_{c'}) I_{c',c}\end{aligned} \tag{50}$$

The last sum in equation (50) is equal to zero since it is sums up to be equal to the mean minus the mean.

Equations (47) to (50) are then substituted into equation (46) and simplified to form the final expression of the backpropagated delta-error:

$$
\begin{aligned}
\delta^{(l)}_{r,c,h,w} &= \sum_{r'=1}^{R}\sum_{c'=1}^{C'}\sum_{h'=1}^{H'}\sum_{w'=1}^{W'} \frac{\partial L(\theta)}{\partial X^{(l+1)}_{r',c',h',w'}} \frac{\partial X^{(l+1)}_{r',c',h',w'}}{\partial \hat{X}_{r',c',h',w'}} \frac{\partial \hat{X}_{r',c',h',w'}}{\partial X^{(l)}_{r,c,h,w}} \\
&= \sum_{r',c',h',w'} \delta^{(l+1)}_{r',c',h',w'} \gamma^{(l)}_{c'} (\sigma^2_{c'})^{-\frac{1}{2}} (I_{r',r} I_{c',c} I_{h',h} I_{w',w} - \frac{1}{RHW} I_{c',c}) \\
&\quad - \sum_{r',c',h',w'} \delta^{(l+1)}_{r',c',h',w'} \gamma^{(l)}_{c'} \frac{1}{RHW} (X^{(l)}_{r',c',h',w'} - \mu_{c'})(X^{(l)}_{r,c',h,w} - \mu_{c'})(\sigma^2_{c'})^{-\frac{3}{2}} I_{c',c} \\
&= \delta^{(l+1)}_{r,c,h,w} \gamma^{(l)}_{c} (\sigma^2_{c})^{-\frac{1}{2}} - \frac{1}{RHW} \sum_{r',h',w'} \delta^{(l+1)}_{r',c,h',w'} \gamma^{(l)}_{c} (\sigma^2_{c})^{-\frac{1}{2}} \\
&\quad - \frac{1}{RHW} \sum_{r',h',w'} \delta^{(l+1)}_{r',c,h',w'} \gamma^{(l)}_{c} (X^{(l)}_{r',c,h',w'} - \mu_{c'})(X^{(l)}_{r,c,h,w} - \mu_{c})(\sigma^2_{c})^{-\frac{3}{2}} \\
&= \frac{1}{RHW} \gamma^{(l)}_{c} (\sigma^2_{c})^{-\frac{1}{2}} \Bigg(RHW \delta^{(l+1)}_{r,c,h,w} - \sum_{r',h',w'} \delta^{(l+1)}_{r',c,h',w'} \\
&\quad - (X^{(l)}_{r,c,h,w} - \mu_c)(\sigma^2_c)^{-\frac{3}{2}} \sum_{r',h',w'} \delta^{(l+1)}_{r',c,h',w'} (X^{(l)}_{r',c,h',w'} - \mu_{c'}) \Bigg)
\end{aligned}
\tag{51}
$$

The derivation of the derivatives of the loss with respect to the parameters are straight-forward and found in a similar way as equation (46) to (51). [19] [20]

$$
\begin{aligned}
\frac{\partial L(\theta)}{\partial \gamma^{(l)}_c} &= \sum_{r}^{R}\sum_{c'}^{C'}\sum_{h'}^{H'}\sum_{w'}^{W'} \frac{\partial L(\theta)}{\partial X^{(l+1)}_{r,c',h',w'}} \frac{\partial X^{(l+1)}_{r,c',h',w'}}{\partial \gamma^{(l)}_c} \\
&= \sum_{r}^{R}\sum_{c'}^{C'}\sum_{h'}^{H'}\sum_{w'}^{W'} \delta^{(l+1)}_{r,c',h',w'} \frac{\partial (\gamma^{(l)}_{c'} \hat{X}_{r,c',h',w'} + \beta^{(l)}_{c'})}{\partial \gamma^{(l)}_c} \\
&= \sum_{r}^{R}\sum_{c'}^{C'}\sum_{h'}^{H'}\sum_{w'}^{W'} \delta^{(l+1)}_{r,c',h',w'} \hat{X}_{r,c,h',w'} I_{c',c} \\
&= \sum_{r}^{R}\sum_{h'}^{H'}\sum_{w'}^{W'} \delta^{(l+1)}_{r,c,h',w'} \hat{X}_{r,c,h',w'}
\end{aligned}
\tag{52}
$$

$$
\begin{aligned}
\frac{\partial L(\theta)}{\partial \beta_c^{(l)}} &= \sum_{r}^{R}\sum_{c'}^{C'}\sum_{h'}^{H'}\sum_{w'}^{W'} \frac{\partial L(\theta)}{\partial X_{r,c',h',w'}^{(l+1)}} \frac{\partial X_{r,c',h',w'}^{(l+1)}}{\partial \beta_c^{(l)}} \\
&= \sum_{r}^{R}\sum_{c'}^{C'}\sum_{h'}^{H'}\sum_{w'}^{W'} \delta_{r,c',h',w'}^{(l+1)} \frac{\partial(\gamma_{c'}^{(l)} \hat{X}_{r,c',h',w'} + \beta_{c'}^{(l)})}{\partial \beta_c^{(l)}} \\
&= \sum_{r}^{R}\sum_{c'}^{C'}\sum_{h'}^{H'}\sum_{w'}^{W'} \delta_{r,c,h',w'}^{(l+1)} I_{c',c} \\
&= \sum_{r}^{R}\sum_{h'}^{H'}\sum_{w'}^{W'} \delta_{r,c,h',w'}^{(l+1)}
\end{aligned}
\tag{53}
$$

4.10 Softmax forward propagation

Softmax is used in the last layer of a neural network to bound the predicted values to the interval [0, 1]. It has the properties that the sum of every training example in the mini-batch is equal to 1. The activations of the softmax layer can therefore be interpreted as probabilities. If the model wants to classify a given input into one of C classes, the output $\hat{y}$ can be interpreted as the the probability that the given input is of class c; $0 \leq c \leq C$, for every class prediction in the output vector. [1]

The input to the softmax layer is resized to a tensor of order 2 $X^{(l)} \in \mathbb{R}^{R \times C}$ and produces an output of the same size. Softmax is defined by equation (54): [1]

$$
X_{r,c}^{(l+1)} = \frac{e^{X_{r,c}^{(l)}}}{\sum_{c'=1}^{C} e^{X_{r,c'}^{(l)}}} \tag{54}
$$

4.11 Softmax backpropagation

The softmax layer does not have any layers and therefore only the recursive delta-error has to be backpropagated. The backpropagation of the recursive delta-error $\delta^{(l+1)}$ is derived by the use of the chain rule. $\frac{\partial L(\theta)}{\partial X^{(l)}}$ is broken up into the two partial derivatives $\frac{\partial L(\theta)}{\partial X^{(l+1)}}$ and $\frac{\partial X^{(l+1)}}{\partial X^{(l)}}$ and the sum of all the neurons in the mini-batch is taken, similar to equations (25) and (46). $X^{(l+1)}$ is then substituted with its definition from equation (54) and is differentiated with

the product rule and the kronecker-delta. The equation is then simplified with the use of basic algebra: [1] [21] [22]

$$\begin{aligned}
\delta_{r,c}^{(l)} &= \frac{\partial L(\theta)}{\partial X_{r,c}^{(l)}} \\
&= \sum_{c'=1}^{C} \frac{\partial L(\theta)}{\partial X_{r,c'}^{(l+1)}} \frac{\partial X_{r,c'}^{(l+1)}}{\partial X_{r,c}^{(l)}} \\
&= \sum_{c'=1}^{C} \delta_{r,c}^{(l+1)} \left(\frac{(e^{X_{r,c'}^{(l+1)}}) I_{c',c}}{\sum_{c''=1}^{C} e^{X_{r,c''}^{(l+1)}}} - \frac{(e^{X_{r,c'}^{(l+1)}})(e^{X_{r,c}^{(l+1)}})}{(\sum_{c''=1}^{C} e^{X_{r,c''}^{(l+1)}})^2} \right) \\
&= \sum_{c'=1}^{C} \delta_{r,c}^{(l+1)} X_{r,c'}^{(l+1)} (I_{c',c} - X_{r,c}^{(l+1)}) \\
&= \delta_{r,c}^{(l+1)} X_{r,c}^{(l+1)} \left(1 - \sum_{c'=1}^{C} X_{r,c'}^{(l+1)} \right)
\end{aligned} \tag{55}$$

5 Results and discussion

The code and weights for the models are publically available on my github profile `https://github.com/nikitazozoulenko` in the repositories *pytorch-face-recognition* and *FCC-vs-CNN-mnist*.

5.1 Classification of handwritten digits

A basic feed-forward or convolutional neural network can be used to classify handwritten digits. For this task the MNIST dataset for handwritten digits is used as training data. It contains 60 000 unique training examples in the training set and 10 000 testing examples in the test set. The images are in grayscale and are 28 times 28 pixels in size (see figure 12). [23]

Two different models were trained on the MNIST dataset: A convolutional neural network consisting of 6 convolutional layers with 32 channels each with stride 1 and kernel size 5 followed by two fully connected layers with 1024 and 10 hidden units respectively. The second model is a feed-forward neural network with 6 fully connected layers of size 784 followed by a layer of size 10. Every convolutional and fully connected layer is preceded by batch normalization and followed by a ReLU layer. Both models use softmax as its last layer to convert the predictions into probablities that each image is

Figure 12: Ten pictures of handwritten digits from the MNIST dataset. [23]

one of $C = 10$ classes where the classes correspond to the digits 0 to 9. The input to the convolutional neural network is a tensor of size $R \times 1 \times 28 \times 28$ and only uses one input channel since the images are in grayscale. For the feed-forward neural network the 28×28 images are reshaped into a 784-dimensional sized vector and stacked to form a tensor of size $R \times 784$. R denotes the batch size.

Let $L(\theta)$ be the loss function used for the classification task. The cross entropy function is used which acts on two probability distributions: the predicted probabilities $\hat{y}$ and the real probabilities $\hat{y}$. Cross entropy and its partial derivative is defined by the following equations: [1] [21]

$$L(\theta) = -\sum_{r=1}^{R}\sum_{c=1}^{C} y_{r,c} \ \log \hat{y}_{r,c} \tag{56}$$

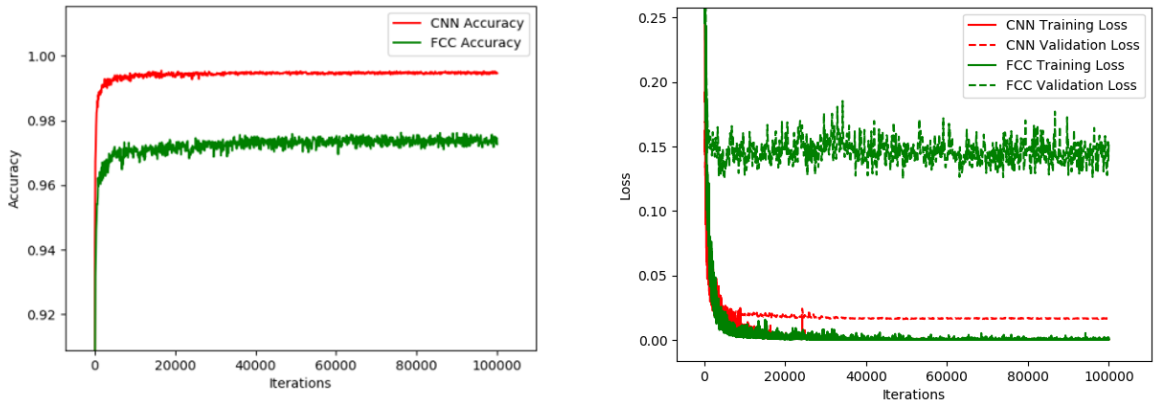
$$\frac{\partial L(\theta)}{\partial \hat{y}_{r,c}} = -\frac{y_{r,c}}{\hat{y}_{r,c}} \tag{57}$$

When training models, it is important to divide your data into a training set and a validation set. During training, the loss on the training set and the validation loss should be computed and tracked. However, only the loss of the training set should be used to train the neural network through Stochastic Gradient Descent. This is done to avoid overfitting the model on the training set. If the model is trained indefinitely on the training set it will learn to make extremely accurate predictions on exclusively the data it was trained on, failing to generalize to data it has not seen before. To combat this the loss or accuracy on the validation set is tracked and training is stopped when the validation accuracy stops improving.

For my own Python implementation of convolutional neural networks, the model was trained for 5 000 iterations on an a CPU with a learning rate

of 0.001 and a batch size of 50. The total training time was 4 hours and the model achieved an accuracy of 99.2% on the test set. Out of 10 000 handwritten the model had not seen previously it managed to classify 9 920 examples correctly. The implementation was deemed too computationally inefficient to do any further testing so I implemented the models in Facebook's GPU-accelerated tensor and dynamic neural network library PyTorch.

The models were trained for 100 000 iterations while tracking training and test loss simultaneously, as well as tracking the test accuracy, shown in figure 13. The initial learning rate was set to 0.001 and divided by 10 at 33 333 and 66 666 iterations. Note that the networks were only trained on the training set, and not on the test set.



(a) Accuracy (b) Losses

Figure 13: The accuracy (a) and loss (b) of the convolutional model (CNN) and the feed-forward model (FCC) as a function of the numbers of iterations trained. The training loss is the loss on the training set and the validation loss is the loss on the test set. Accuracy is measured on the test set.

The convolutional neural network is shown to be able to generalize better on the test set than its feed-forward variant. While the training loss is the same for both the convolutional and feed-forward models, the validation loss is approximately 7 times higher for the feed-forward model. The final accuracy of the two networks were 99.5% for the convolutional neural network and 97.3% for the feed-forward neural network, tested on the test set consisting of 10 000 handwritten digits.

5.2 Dense face detection and localization

For the task of detecting a variable number of faces in an iamge, the WIDER-Face dataset was used. It is a dataset consisting of 32 203 images of a total of 393 703 number of faces at different scales, lighting and occlusions. Every

training example consists of a number of bounding boxes surrounding all the faces in the image. A bounding box consists of four coordinates specifying its location: two for the upper left corner and two for the bottom right corner of the bounding box. The images are labaled by humans and is called the ground truth of the image. [24]

A convolutional neural network can be used to detect up to thousands of different faces in a single picture or video in real time. A custom region based one-shot detector is used to achieve this. One-shot detectors were first introduced in the paper *You Only Look Once: Unified, Real-Time Object Detection* and later modified and improved in the papers *SSD: Single Shot MultiBox Detector*, *YOLO9000: Better, Faster, Stronger*, *DSSD: Deconvolutional Single Shot Detector* and *Focal Loss for Dense Object Detection*. A one-shot detector works by predicting up to tens of thousands of bounding boxes in different spatial positions in an image, that sets out to detect every object in that image. In addition to predicting the bounding box coordinates, the model predicts probability scores of how likely it is that there exists an object inside of the bounding box. [25] [26] [27] [28] [29]

I constructed a model termed FaceResNet based on the current state of the art in object detection: RetinaNet, introduced in the paper *Focal Loss for Dense Object Detection*. It uses a feature pyramid architecture, introduced in the paper *"Feature Pyramid Networks for Object Detection"*, to be able to use feature maps from different depths in the neural network to predict objects of different scales. The backbone network RetinaNet uses for the feature pyramid network is a ResNet-101. The model predicts the bounding boxes by basing their predictions on a number of anchor boxes at different scales and positions in the image. For every anchor box, the network predicts four coordinate offsets to shift the anchor box with and a set of probabilities of there existing an object inside the predicted bounding box. This process is applied to every pyramid level. RetinaNet uses pyramid layers P3 throughout P7 and uses anchor boxes of sizes 32^2 to 512^2. Let the pyramid layers width and height be W and H and let A denote the number of differently shaped anchor boxes at every pyramid layer. In total the pyramid layers will have WHA unique anchor boxes spaced out evenly throughout every pyramid level. The coordinate offset regression and class probability classification for every anchor box are done by two smaller subnetworks called the regression head and the classification head (see figure 14). The classification head predicts $KWHA$ probabilities where K is the number of object classes. K consists of 1 foreground class (no object in the bounding box)

and $K-1$ foreground classes (objects). The regression head predicts $4WHA$ coordinate offsets: four offsets for every bounding box. [29] [30] [31]

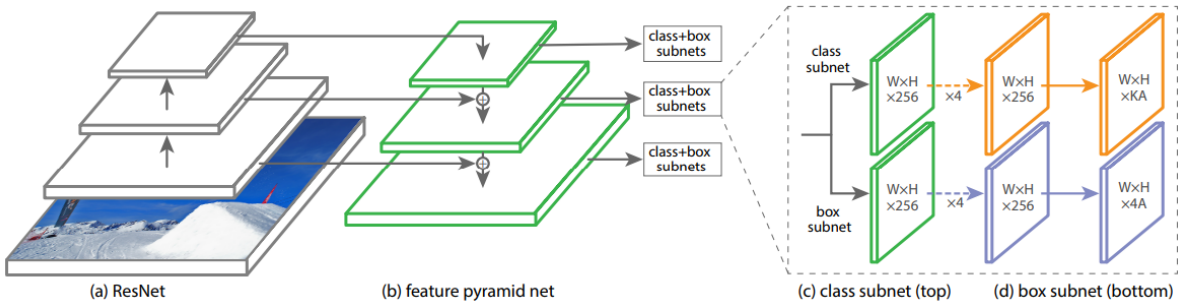


Figure 14: The architecuture of RetinaNet. The last feature maps of every pyramid level is fed into a classification and regression head. [29]

My model FaceResNet differs from RetineNet by using a ResNet-50 as a backbone network and by using $A = 6$ anchor boxes at every pyramid level of sizes 16^2 to 416^2 pixels, scaled by a factor of $2^{\frac{1}{3}}$, and using 2 different aspect ratios: 1:1 and 1:1.5 (width:height). Additionally, both subnetworks use a completely different architecture: Instead of using 5 normal 3×3 convolutions, FaceResNet uses 8 residual bottleneck building blocks; the same building blocks used in the ResNet architecture. FaceResNet only predicts $K = 2$ class probabilities: if the bounding box contains a face or not.

The model is trained by assigning every anchor box either a negative or a positive label, depending on if the anchor box contains a ground truth object. An anchor box is said to contain an object if the intersection over union (IoU) of the area of the anchor box and the area of the ground truth is bigger or equal to 0.5. The intersection over union is used to determine how similar two sets A and B are (see figure 15). It is defined by the following equation: [32]

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{58}$$

The loss function $L(\theta)$ used by FaceResNet is a combination of a coordinate regression loss L_r, and a bounding box classification loss L_c. L_r is the Smooth L1 Loss defined by equation 60. L_c is the Focal Loss defined by equation 59. The hyperparameters used by FaceResNet for the Focal Loss are the same as for RetinaNet, namely, $\gamma = 2$ and $\alpha = 3$ for the foreground class. [29]

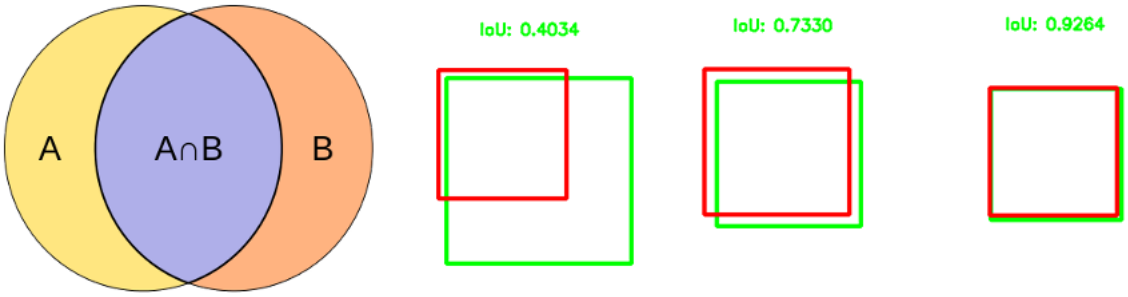


Figure 15: IoU is defined as the size of the union divided by the size of the intersection of two sets A and B. A bigger IoU implies that the predicted bounding box is closer to the ground truth. [32]

$$L_r(x) = \begin{cases} 0.5x^2 & \text{om } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases} \tag{59}$$

$$L_c(p, \hat{p}) = -\alpha(1-p)^\gamma p \log \hat{p} \tag{60}$$

The total loss $L(\theta)$ used by FaceResNet is the mean of the regression losses L_r for all positively assigned anchors plus the mean of the classification losses for all anchor boxes. Negatively assigned anchors do not have any effect on the regression loss. Let r_a and $\hat{r}_a$ be the predicted and ground truth coordinate offsets for anchor box a. Let p_a and $\hat{p}_a$ be the predicted and ground truth probability score for anchor a to contain an object. N is the number of anchor boxes and N_{pos} is the number of positively assigned anchor boxes. The total loss can then be described by the following equation:

$$\begin{aligned} L(\theta) = &\frac{1}{N} \sum_{a \in anchors} L_c(p_a, \hat{p}_a) \\ &+ \frac{1}{N_{pos}} \sum_{a \in positive} L_r(r_a - \hat{r}_a) \end{aligned} \tag{61}$$

The model was trained for 700 000 iterations on an Nvidia GTX 1080ti with an initial learning rate of 0.001. Due to memory limitations a batch size of 2 had to be used, the bare minimum for batch normalization to function. Every input image was randomly resized to be of size 512^2 or 640^2. Following industry standards for data augmentation, the network was trained on random image crops of the training data to artificially increase the size of the dataset. Additionally, the image was flipped horizontally with a probability of 0.5 and random color jitter was applied. Figures 16 - 18 show quantitative results on images not found in the WIDERFace data set.

Figure 16: Qualitative results of the model FaceResNet.

Figure 17: Qualitative results of the model FaceResNet.

Figure 18: Qualitative results of the model FaceResNet.

One forward pass of an image of size 512^2 takes 30 ms, enabling the model to be used in real time for real time applications. The model can for instance be applied to CCTV applications for security cameras or be used as a face detector for automatic tagging on social media or face detection in cameras.

An area where FaceResNet shows weakness is in detecting small faces. This can be due to there not being enough information in the image for the model to accurately detect the small faces since they only consist of 16 pixels in width and height. Another reason could be that the classification error of the negative boxes dominates the error of the positive boxes. Since the small bounding boxes are created from the lowest pyramid layer with the biggest spatial height and with, more smaller than bigger bounding boxes are created, while the number of ground truth boxes stays constant for approximately all sizes. The ratio of negative to positive bounding boxes is bigger for the small anchors, resulting in detection performance for small faces.

6 Conclusion

Feed-forward and convolutional neural networks are a type of artificial neural network consisting of sequential layers of neurons. In a feed-forward neural network every neuron from the previous layer is connected to all the neurons in the proceeding layer. In a convolutional neural network, only nearby neurons are connected to the neurons of the proceeding layer through what is called the convolution operation. Forward propagation is the the process of forwarding the signal from your input neurons to your output neurons through the neural network. Backpropagation is the method of training the neural network through repeated use of the chain rule to find how the neural network's parameters should be modified to minimize a determined loss function. Classification of handwritten digits was achieved by using a 7 layer deep feed-forward neural network with hidden layer sizes 1000, as well by using a 7 layer deep convolutional neural network with kernel size 5, followed by two fully connected layers. Additionally, a deep convolutional neural network based on a state of the art one-shot detector was used to detect a variable number of faces in an image.

References

[1] *CS231n: Convolutional Neural Networks for Visual Recognition.* F. Li, A. Karpathy and J. Johnson. Stanford University, university course, winter 2016.

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

[3] *Eigen v3.* G. Guennebaud and B. Jacob and others. URL http://eigen.tuxfamily.org. 2010

[4] *The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30.* S. van der Walt, S. C. Colbert and G. Varoquaux. (2011), DOI:10.1109/MCSE.2011.37

[5] *TensorFlow: Large-scale machine learning on heterogeneous systems*, M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A.Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah,

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. URL https://www.tensorflow.org/. 2015.

[6] *Automatic differentiation in PyTorch.* A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer 2017

[7] *A simple neural network with Python and Keras.* A. Rosebrock. PyImageSearch URL https://www.pyimagesearch.com/. 26 September 2016.

[8] *Unsupervised Feature Learning and Deep Learning.* Standford University, Department of Computer Science. URL http://ufldl.stanford.edu/wiki/. Last updated 31 Mars 2013.

[9] *Scientific Computing 2013, Worksheet 6: Optimization: Gradient and steepest descent.* University of Tartu, Estonia. 2013.

[10] *Introduction to Convolutional Neural Networks.* J. Wu. National Key Lab for Novel Software Technology, Nanjing University, China. 1 May, 2017.

[11] *Tuning the learning rate in Gradient Descent.* V. Vryniotis. URL http://blog.datumbox.com/tuning-the-learning-rate-in-gradient-descent/. 27 October 2013.

[12] *A guide to convolution arithmetic for deep learning.* V. Dumoulin and F. Visin. FMILA, Université de Montréal. AIRLab, Politecnico di Milano. 24 mars, 2016.

[13] *Very Deep Convolutional Networks for Large-Scale Image Recognition.* K. Simonyan and A. Zisserman. arXiv preprint arXiv:1409.1556, 2014

[14] *Understanding Convolutional Neural Networks for NLP.* D. Britz. WildML, Artificial Intelligence, Deep Learning, and NLP. 7 November 2015.

[15] *Deep learning for complete beginners: convolutional neural networks with keras.* P. Veličković. Camebridge Spark. URL https://cambridgespark.com/content. Last updated 20 Mars 2017.

[16] *Backpropagation In Convolutional Neural Networks.* J. Kafunah. DeepGrid, Organic Deep Learning. URL http://www.jefkine.com/. 5 September 2016.

[17] *Note on the implementation of a convolutional neural networks.* C. Thorey. Machine Learning Blog. URL http://cthorey.github.io/. 2 February 2016.

[18] *Convolutional Neural Networks.* A. Gibiansky. URL http://andrew.gibiansky.com. 24 February 2014.

[19] *What does the gradient flowing through batch normalization looks like?* C. Thorey. Machine Learning Blog. URL http://cthorey.github.io/. 28 January 2016.

[20] *Understanding the backward pass through Batch Normalization Layer.* Flaire of Machine Learning URL https://kratzert.github.io. 5 September 2016.

[21] *Notes on Backpropagation.* P. Sadowski. University of California Irvine Department of Computer Science.

[22] *Classification and Loss Evaluation - Softmax and Cross Entropy Loss.* P. Dahal. DeepNotes. URL https://deepnotes.io/softmax-crossentropy. 24 February 2014.

[23] *The MNIST database of handwritten digits* Y. LeCun, C. Cortes and C. Burges. Courant Institute, NYU. Google Labs, New York. Microsoft Research, Redmond. URL http://yann.lecun.com/exdb/mnist/. 3 November 2017.

[24] *WIDER FACE: A Face Detection Benchmark.* Yang, Shuo and Luo, Ping and Loy, Chen Change and Tang, Xiaoou IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016

[25] *You only look once: Unified, real-time object detection.* J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. arXiv preprint arXiv:1506.02640, 2015.

[26] *SSD: Single Shot MultiBox Detector.* W. Liu and D. Anguelov and D. Erhan and C. Szegedy and S. Reed and C. Fu and A. Berg. arXiv preprint arXiv:1512.02325, 2015

[27] *YOLO9000: Better, Faster, Stronger.* J. Redmon and A. Farhadi. arXiv preprint arXiv:1612.08242, 2016.

[28] *DSSD : Deconvolutional Single Shot Detector.* C. Fu and W. Liu and A. Ranga and A. Tyagi and A. C. Berg arXiv preprint arXiv:1701.06659, 2017

[29] *Focal Loss for Dense Object Detection.* Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He and Piotr Dollár. arXiv preprint arXiv:1708.02002, 2017

[30] *Feature Pyramid Networks for Object Detection.* Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan and Serge J. Belongie. arXiv preprint arXiv:1612.03144, 2016

[31] *Deep residual learning for image recognition.* K. He, X. Zhang, S. Ren, and J. Sun. arXiv preprint arXiv:1512.03385, 2015.

[32] Jaccard Index. Wikipedia. URL https://en.wikipedia.org/wiki/Jaccard_index. 20 January 2018