Derivation of a Convolutional Neural Network Based Dense Face Detector

February 20, 2018
Nikita Zozoulenko Na15b
nikita.zozoulenko@gmail.com
Katedralskolan Linköping
Supervisor: Rickard Engström
VT 2017 - HT 2018

Abstract

Abstract Here......

In this paper the mathematical model behind the standard Artificial Neural Network and the Convolutional Neural Network is derived. A Convolutional Neural Network is then adapted to read handwritten digits and to make a cutting edge face detector, successfully detecting over 50 faces in a crowded scene.

Contents

1 Introduction

1.1 Background

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]

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 out of a number of layers of neurons. The layers are built recursively such that the output of one layer is the input of the next layer. They are stacked on top of each other and the number of layers an Artificial neural network has is called its depth. How the neurons of the previous layer are connected to the neurons of the next layer depends on what type on Artificial neural network it is. [1]

1.2 Purpose

When I was first starting out in the field of deep learning I found 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 case. 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 Feedforward neural network and Convolutional neural network? What is forward- and backpropgation and how is it derived in a Feedforward neural network and 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

The majority of the paper and the derivations of the models are based on the course material from Stanford's "CS231n: Convolutional neural networks for Visual Recognition" course. 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 and 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 with their numerical approximations using the formal definition of a derivative.

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 I implemented the models in Google's machine learning library TensorFlow and Facebook's GPU-accelerated tensor and dynamic neural network library PyTorch.

3 Results and discussion

3.1 Feed-forward neural networks

A feed-forward neural network is the most elementary version of an artificial neural network. As all varying kinds of 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 1). [1]

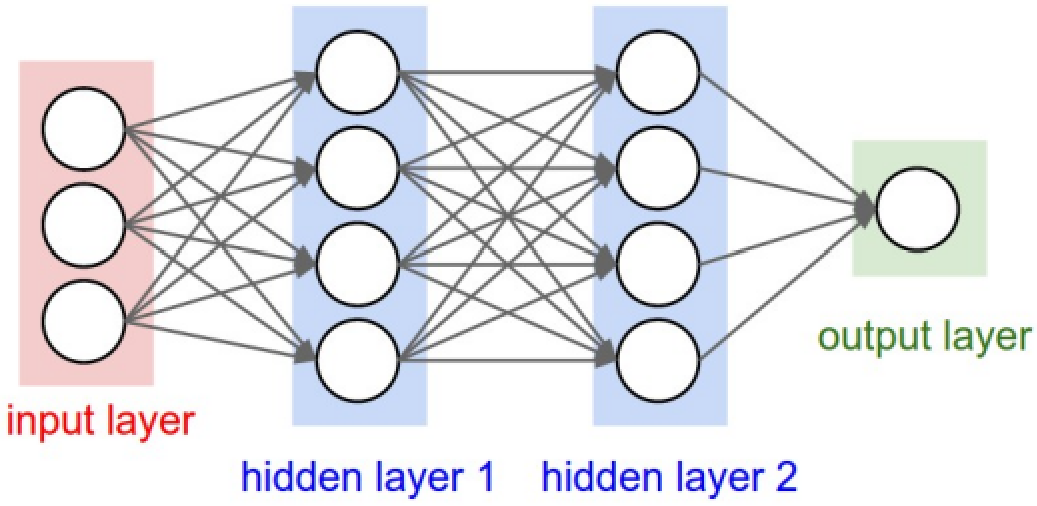

Figure 1: An illustration of a simple feed-forward neural network. It consists of 4 layers: 1 input layer (red), 2 hidden layers (blue) and 1 output layer (green). A circle represents a neuron. Every neuron in a layer is connected to all the neurons in the following layer, shown by the grey lines between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

A neuron is represented as 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 where n and m are the number neurons in the input and output layers. Every data point or feature in the input are set as neurons in the input layer. For instance, if you want to predict a score given 12 different variables every single variable would correspond to one neuron in the input layer of size 12 and the score would correspond to a single neuron in the output layer of size 1. The value of the input neurons are transferred over to the next layer depending on the strength of the different connections every neuron has to the neurons in the next layer. This is repeated 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 all the layers in the neural network. [1]

3.1.1 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 vector $x \in \mathbb{R}^N$ and is a row vector 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 a vector of 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 $0 \leq r < R$, $0 \leq c < C$, $0 \leq h < H$ och $0 \leq w < W$. [1]

3.1.2 Forward propagation

In figure 1 only a single training example was worked on at a time. In practice a mini-batch of R training examples are propagated forward in a neural network at the same time. [1] [2]

Let M denote the number of layers in the neural network and l; $0 \leq l < M$, one specific layer in the neural network. Let $N^{(l)}$ denote the number of neurons in layer l. The activation 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 $0 \leq r < R$ and $0 \leq i < N$. In addition to the normal neurons a layer has, let $b^{(l)}$ be a bias neuron for the layer l (see figure 2). It is called a bias neuron because its value is independent to what input the neural network is given. [1] [2]

The weights expressing how strong the connection between the neurons of layer l and $l + 1$ are can also be 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] [2]

To calculate the value of a neuron in layer $l + 1$ every neuron in layer l is multiplied by its corresponding weight in $W^{(l)}$ and summed together with the layers bias neuron. The sum is then put in a so called activation function f. The value of the activation function is the neurons activation in layer $l+1$. [1] [2]

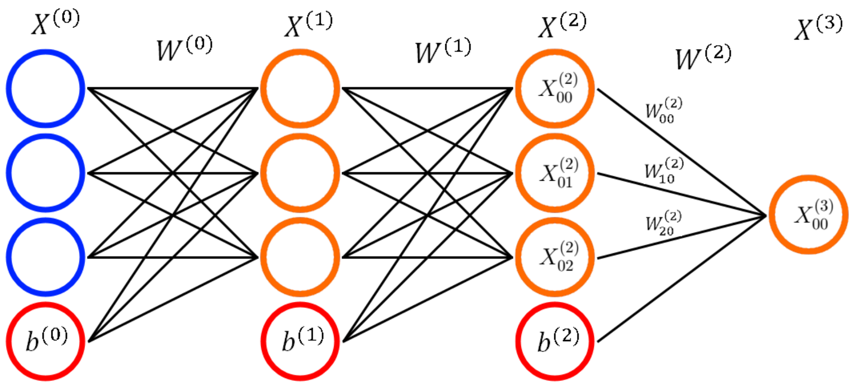

Figure 2: An example of a 4 layer feed-forward neural network. The input neurons are marked with blue. The bias neurons are marked with red. Black lines between neurons symbolyze the weights between every pair of neurons between two layers.

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 $X^{(0)}$ the forward propagation can be expressed recursively as: [1] [2]

$$Z^{(l+1)}_{rj} = b^{(l)} + \sum_{i=0}^{N^{(l)}-1} X^{(l)}_{r,i} W^{(l)}_{i,j} \tag{1}$$

$$X^{(l+1)}_{rj} = f(Z^{(l+1)}_{rj}) \tag{2}$$

The tensor of activations and weights are constructed in such as way that one forward pass through a single layer can be computed with a single dot product and having the bias term added to every element in the computed dot product. The activation function f is then applied element-wise on each neuron: [1] [2]

$$Z^{(l+1)} = X^{(l)} W^{(l)} + b^l \tag{3}$$

$$X^{(l+1)} = f(Z^{(l+1)}) \tag{4}$$

Common activation functions are Rectified Linear Units (ReLU), sigmoid (σ) and hyperbolic tangent (tanh) and are defined by equation (5) - (7) (see figure 3). 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]

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

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

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

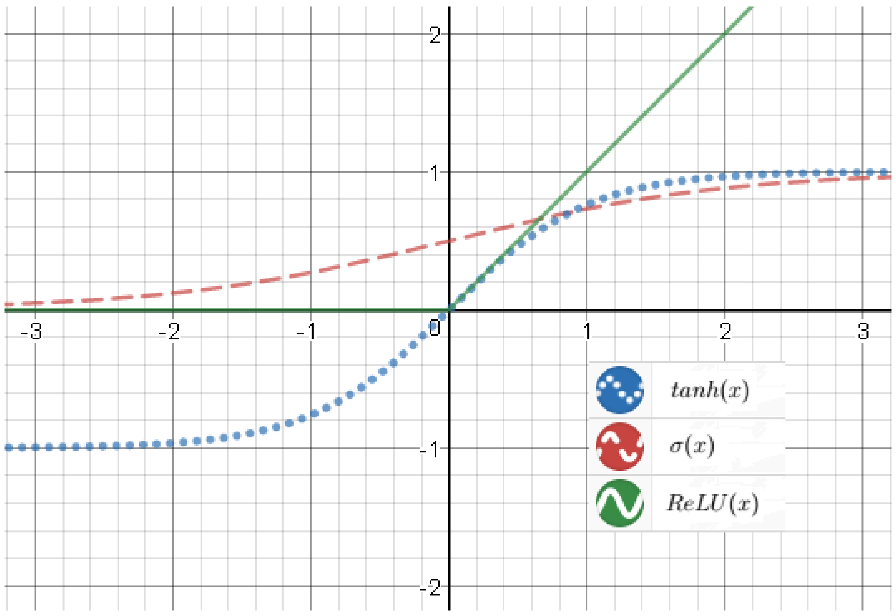

Figure 3: A graph of ReLU, σ och tanh.

3.1.3 Cost 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 cost 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 cost function describes the quality of the prediction $\hat{y}$ such that a lower cost represents a more accurate prediction. One way of defining the cost function is to use the mean squared error as in equation (8):

$$L(\theta) = \frac{1}{RN} \sum_{r=0}^{R-1} \sum_{i=0}^{N-1} (\hat{y}_{r,i} - y_{r,i})^2 \tag{8}$$

Where R is the batch size and N is the number of features in the last layer.

The neural network is trained to predict accurate results by iterating through the training data and minimizing the cost function. Since the given input X stays fixed, the network learns to optimize its weights $W^{(l)}$ and biases $b^{(l)}$ for every layer l.

3.1.4 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 equation (9) and (10): [7] [4]

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

$$\nabla L(\theta) = \left(\frac{\partial L(\theta)}{\partial \theta_0}, \ \frac{\partial L(\theta)}{\partial \theta_1}, \ \cdots, \ \frac{\partial L(\theta)}{\partial \theta_{n-1}} \right) \tag{10}$$

The gradient $\nabla L(\theta)$ shows the direction of steepest ascent in the point (θ_0, θ_0, ..., θ_0) 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 multivariate function $f(x, y)$ of two variables x and y the gradient would be the a two-dimensional vector of the slope in the x dimension and y dimension respectively.

Gradient descent is the method of iteratively changing the values of θ proportionally to the negative gradient $-\nabla L(\theta)$ to minimize the function $L(\theta)$ (see figure 4). The most basic version of gradient descent is called Stochastic Gradient Descent (SGD) and uses the hyperparameter α, called learning rate, to control the magnitude of the gradient. Stochastic Gradient Descent is defined by equations (11) and (12).

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

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

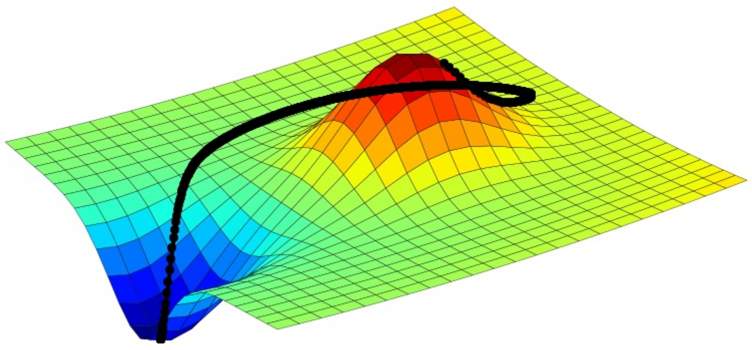

Figure 4: An illustration of Stochastic Gradient Descent on a function of two variables. Red symbolizes a high value of the function while blue symbolizes a low value. The parameters are initialized at the global maximum and their values are altered iteratively to move in the direction of the negative gradient: the direction of steepest descent. [17]

3.1.5 Backpropagation

Backpropagation stands for "backwards propagation of errors" and is the process of calculating the partial derivatives of the loss function with respect to the model's parameters, or in other terms calculating the gradient. [2] [7]

The partial derivatives can be approximated numerically with the formal definition of a derivative defined by equation (13): [2] [7]

$$\frac{\partial L(\theta)}{\partial \theta_i} = \frac{L(\theta_0, ..., \theta_i + h, ..., \theta_{n-1}) - L(\theta)}{h} \tag{13}$$

This would not be a problem for the small neural network in figure 2 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.

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

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

The delta error is the partial derivative of the loss with respect to a specific neuron in the model and is needed to efficiently calculate the gradient. The delta error can be interpreted as how much a specific neuron affects the total loss of the cost function. A deviation 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 $\frac{\partial L(\theta)}{\partial X_{r,j}^{(l+1)}}$, $\frac{\partial X_{r,j}^{(l+1)}}{\partial Z_{r,j}^{(l+1)}}$ and $\frac{\partial Z_{r,j}^{(l+1)}}{\partial X_{r,i}^{(l)}}$. 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:

$$\begin{aligned}
\delta_{r,i}^{(l)} &= \frac{\partial L(\theta)}{\partial X_{r,i}^{(l)}} \\
&= \sum_{j=0}^{N^{(l+1)}} \frac{\partial L(\theta)}{\partial X_{r,j}^{(l+1)}} \frac{\partial X_{r,j}^{(l+1)}}{\partial Z_{r,j}^{(l+1)}} \frac{\partial Z_{r,j}^{(l+1)}}{\partial X_{r,i}^{(l)}} \\
&= \sum_{j=0}^{N^{(l+1)}} \delta_{r,j}^{(l+1)} f'(Z_{r,i}^{(l+1)})\, W_{j,i}^{(l)}
\end{aligned} \tag{15}$$

The delta error of the last layer $L-1$ depends on what loss function is used. For the mean squared error the delta error of the output layer is defined by equation (16):

$$\begin{aligned}
\delta^{(L-1)} &= \frac{\partial L(\theta)}{\partial \hat{y}} \\
&= \frac{2}{RN}(\hat{y} - y)
\end{aligned} \tag{16}$$

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) $\frac{\partial L(\theta)}{\partial X^{(l)}}$ is broken up into three partial derivatives $\frac{\partial L(\theta)}{\partial X_{r,i}^{(l+1)}}$, $\frac{\partial X_{r,i}^{(l+1)}}{\partial Z_{r,i}^{(l+1)}}$ and $\frac{\partial Z_{r,i}^{(l+1)}}{\partial W_{i,j}^{(l)}}$. All the examples in the mini-batch are summed over since the weights affect every single example. $\frac{\partial L(\theta)}{\partial X_{r,i}^{(l+1)}}$ is the delta error and the other two partial derivatives are derived from equations (1) and (2):

$$\begin{aligned}\frac{\partial L(\theta)}{\partial W_{j,i}^{(l)}} &= \sum_{r=0}^{R-1} \frac{\partial L(\theta)}{\partial X_{r,i}^{(l+1)}} \frac{\partial X_{r,i}^{(l+1)}}{\partial Z_{r,i}^{(l+1)}} \frac{\partial Z_{r,i}^{(l+1)}}{\partial W_{i,j}^{(l)}} \\ &= \sum_{r=0}^{R-1} \delta_{r,i}^{(l+1)} f'(Z_{r,i}^{(l+1)})\ X_{r,j}^{(l)}\end{aligned} \tag{17}$$

The partial derivative of the loss function with respect to the biases are found in a similar way to equation (15) and (17):

$$\begin{aligned}\frac{\partial L(\theta)}{\partial b^{(l)}} &= \sum_{r=0}^{R-1} \frac{\partial L(\theta)}{\partial X_{r,i}^{(l+1)}} \frac{\partial X_{r,i}^{(l+1)}}{\partial Z_{r,i}^{(l+1)}} \frac{\partial Z_{r,i}^{(l+1)}}{\partial b_{i,j}^{(l)}} \\ &= \sum_{r=0}^{R-1} \delta_{r,i}^{(l+1)} f'(Z_{r,i}^{(l+1)})\end{aligned} \tag{18}$$

3.1.6 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. [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`.

3.2 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. for. 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 5). These filters are stacked on top of each other in terms of layers and allows the networks to learn higher level features the deeper the network architecture goes. 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.

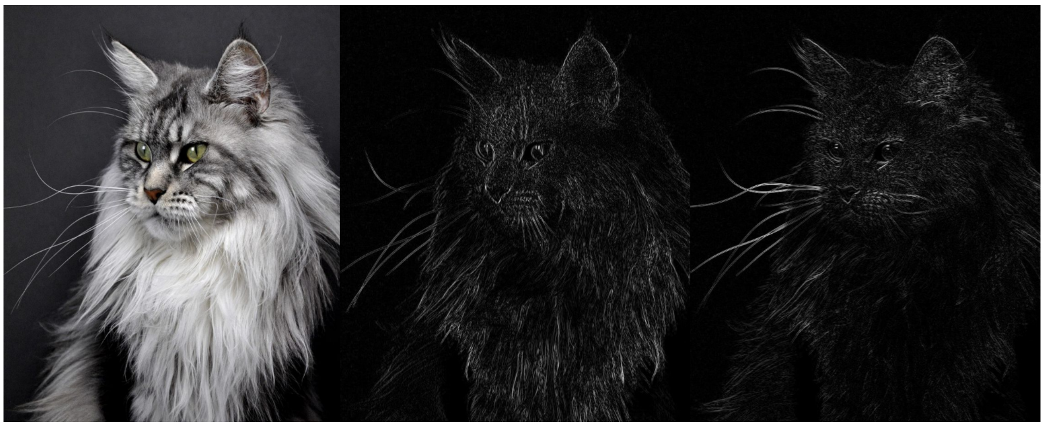

Figure 5: The result of a filter for vertical and horizontal edge detection applied on a picture of a cat.

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 that 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 is currently the state of the art in computer vision.

3.2.1 Model structure, parameters and notation

Convolutional neural networks are constructed out 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 only have 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.

At every layer l there are parameters $\theta^{(l)}$ and activations (neurons) $X^{(l)}$. The last layer is denoted by $\hat{y}$ and $X^{(L-1)}$ where L is the number of layers in the network. Given an input $X^{(0)}$ 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). Just like in a feed-forward neural network, the prediction is then used to determine a cost with the use of a loss function $L(\theta)$ which is later used by the backpropagation algorithm.

$$X^{(0)} \xrightarrow{\theta^{(0)}} X^{(1)} \xrightarrow{\theta^{(1)}} \cdots \xrightarrow{\theta^{(L-3)}} X^{(L-2)} \xrightarrow{\theta^{(L-2)}} X^{(L-1)} = \hat{y} \tag{19}$$

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. Too 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.

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 ????) or as stacked feature maps (see figure ????).

Every single type of layer in a convolutional neural network has to two modes: forward propagation and backpropagation. The model is trained the same way 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. In the backprop-

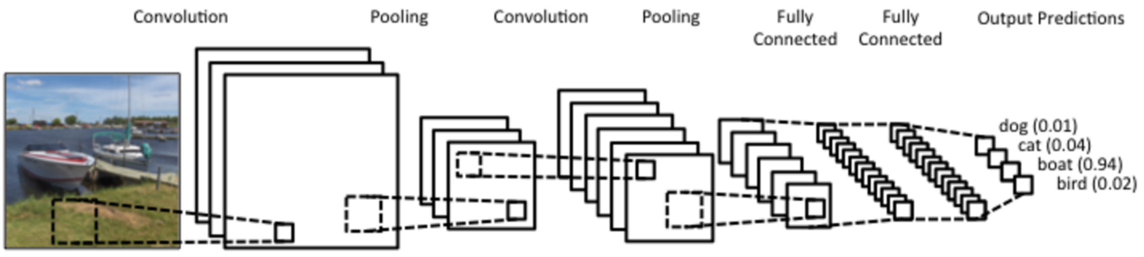

Figure 6: En illustration av ett konvolutionellt neuralt nätverk. Varje skiva är en egen *feature map.* [18]

agation 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.

Because every layer operation in the neural network is defined recursively, 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 zero-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 differ depending on the layer type.

3.2.2 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. k is the kernel size of the convolution.

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 7). 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 $l+1$. This process being applied to every single neuron is called a convolution. The convolution operator is denoted by $*$.

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'

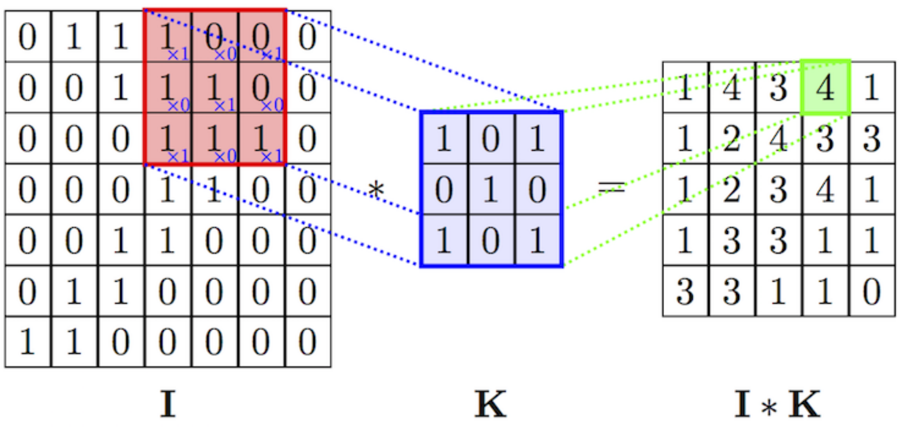

Figure 7: A kernel of size $1 \times 3 \times 3$ convolving over activations of size $1 \times 7 \times 7$, producing activations of size $1 \times 4 \times 4$ in the next layer. [19]

is the number of kernels a layer has and is also the number of feature maps the next layer will have. [1] [4]

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 7 has a stride of $s = 1$. Zero-padding is when you pad the edges of the activation tensor with p zeros (see figure 8). 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] [4] [5]

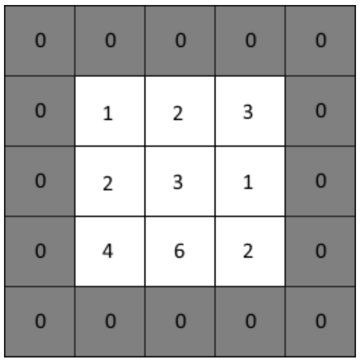

Figure 8: An activation with size $1 \times 3 \times 3$ is zero-padded with $p = 1$ and the resulting tensor is of size $1 \times 5 \times 5$.

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] [4] [5]

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

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

Mathematically the convolutional layer is defined by the following equations: [1] [4]

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

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

$$\begin{aligned} \left[X^{(l+1)}\right]_{r,c',h',w'} &= X^{(l)}_{r,c',h',w'} * W^{(l)}_{c'} \\ &= \sum_{c=0}^{C-1} \sum_{j=0}^{K_H-1} \sum_{i=0}^{K_W-1} X^{(l)}_{r,c,h'+j,w'+i} W^{(l)}_{c',c,j,i} \end{aligned} \tag{24}$$

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 efficient implementation is required. [1] [4] [5]

3.2.3 Convolution backpropagation

At every layer l the delta-error of the proceeding layer $\delta^{(l+1)}$ has to be backpropagated 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.

The backpropagation of the recursive delta-error $\delta^{(l+1)}$ is derived by the use of the chain rule. $\delta^{(l+1)} = \frac{\partial L(W)}{\partial X^{(l+1)}_{r,c',h',w'}}$ is broken up into two smaller partial derivatives $\frac{\partial L(W)}{\partial X^{(l+1)}_{r,c',h',w'}}$ and $\frac{\partial X^{(l+1)}_{r,c',h',w'}}{\partial X^{(l)}_{r,c,h,w}}$. 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): [4] [10] [12] [14]

$$
\begin{aligned}
\delta^{(l)}_{r,c,h,w} &= \frac{\partial L(W)}{\partial X^{(l)}_{r,c,h,w}} \\
&= \sum_{c'=0}^{C'-1}\sum_{h'=0}^{H'-1}\sum_{w'=0}^{W'-1} \frac{\partial L(W)}{\partial X^{(l+1)}_{r,c',h',w'}} \frac{\partial X^{(l+1)}_{r,c',h',w'}}{\partial X^{(l)}_{r,c,h,w}} \\
&= \sum_{c'=0}^{C'-1}\sum_{h'=0}^{H'-1}\sum_{w'=0}^{W'-1} \delta^{(l+1)}_{r,c',h',w'} \frac{\partial \sum_{c=0}^{C-1}\sum_{j=0}^{k_H-1}\sum_{i=0}^{k_W-1} X^{(l)}_{r,c,h'+j,w'+i} W^{(l+1)}_{c',c,j,i}}{\partial X^{(l)}_{r,c,h,w}}
\end{aligned}
\tag{25}
$$

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: [10] [12] [14]

$$
\begin{aligned}
\sum_{c'}^{C'-1}\sum_{h'=0}^{H'-1}\sum_{w'=0}^{W'-1} \delta^{(l+1)}_{r,c',h',w'} \frac{\partial \sum_{c=0}^{C-1}\sum_{j=0}^{K_H-1}\sum_{i=0}^{K_W-1} X^{(l)}_{r,c,h'+j,w'+i} W^{(l+1)}_{c',c,j,i}}{\partial X^{(l)}_{r,c,h,w}} \\
= \sum_{c'=0}^{C'-1}\sum_{h'=0}^{H'-1}\sum_{w'=0}^{W'-1} \delta^{(l+1)}_{r,c',h',w'} W^{(l+1)}_{c',c,(h-h'),(w-w')}
\end{aligned}
\tag{26}
$$

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): [10] [12] [14]

$$
\delta^{(l)}_{r,c,h,w} = \sum_{c'=0}^{C'-1} rot(W^{(l+1)}_{c',c,h,w}) * \delta^{(l+1)}_{r,c'} \tag{27}
$$

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] [10] [12] [14]

$$\begin{aligned}
\frac{\partial L(W)}{\partial W^{(l)}_{c',c,h,w}} &= \sum_{r=0}^{R-1}\sum_{c'=0}^{C'-1}\sum_{h'=0}^{H'-1}\sum_{w'=0}^{W'-1} \frac{\partial L(W)}{\partial X^{(l+1)}_{r,c',h',w'}} \frac{\partial X^{(l+1)}_{r,c',h',w'}}{\partial W^{(l)}_{r,c,h,w}} \\
&= \sum_{r=0}^{R-1}\sum_{c'=0}^{C'-1}\sum_{h'=0}^{H'-1}\sum_{w'=0}^{W'-1} \delta^{(l+1)}_{r,c',h',w'} \frac{\partial \sum\limits_{c=0}^{C-1}\sum\limits_{j=0}^{K_H-1}\sum\limits_{i=0}^{K_W-1} X^{(l)}_{r,c,h'+j,w'+i} W^{(l)}_{c',c,j,i}}{\partial W^{(l)}_{c',c,h,w}} \\
&= \sum_{r=0}^{R-1}\sum_{c'=0}^{C'-1}\sum_{h'=0}^{H'-1}\sum_{w'=0}^{W'-1} X^{(l)}_{r,c,h'+h,w'+w} \delta^{(l+1)}_{r,c',h',w'} \\
&= \sum_{r=0}^{R-1}\sum_{c'=0}^{C'-1} X^{(l)}_{r,c,h,w} * \delta^{(l+1)}_{r,c'}
\end{aligned} \tag{28}$$

3.2.4 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. [4]

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]

3.2.5 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. $\frac{\partial L(W)}{\partial X^{(l)}_{r,c,h,w}}$ is split up into $\frac{\partial L(W)}{\partial X^{(l+1)}_{r,c,h,w}}$ 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] [4]

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

3.2.6 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 9). 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] [4] [5]

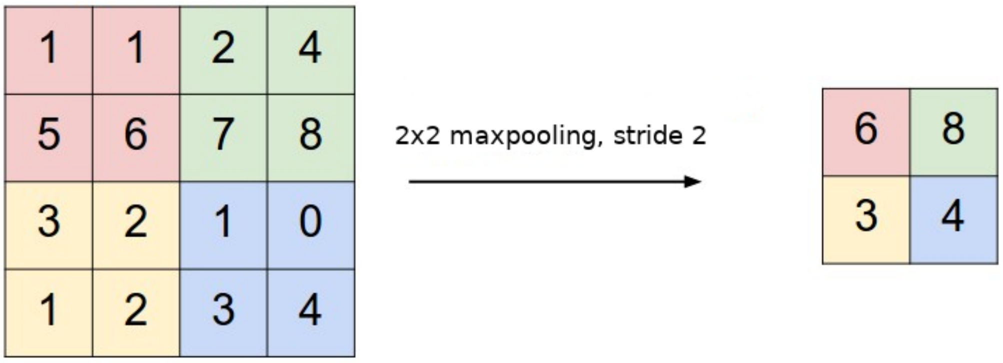

Figure 9: Maxpooling with kernel size $k = 2$ and stride $s = 2$ on an area of size 4×4. The resulting area has size 2×2

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] [4] [5]

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

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

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

Equation (34) defines the maxpooling layer algebraically. [1] [4]

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

3.2.7 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 $\frac{\partial L(W)}{\partial X^{(l+1)}_{r,c',h',w'}}$ and $\frac{\partial X^{(l+1)}_{r,c',h',w'}}{\partial X^{(l)}_{r,c,h,w}}$. The first term is the delta-error at layer $l+1$. $X^{(l+1)}_{r,c',h',w'}$ is then substituted with its definition from equation (34): [1] [4] [14]

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

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] [4] [14]

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

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] [4] [14]

3.2.8 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] [9]

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] [9]

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] [9]

$$\mu_c = \frac{1}{RHW} \sum_{r=0}^{R-1} \sum_{h=0}^{H-1} \sum_{w=0}^{W-1} X_{r,c,h,w}^{(l)} \tag{37}$$

$$\sigma_c^2 = \frac{1}{RHW} \sum_{r=0}^{R-1} \sum_{h=0}^{H-1} \sum_{w=0}^{W-1} (X_{r,c,h,w}^{(l)} - \mu_c)^2 \tag{38}$$

Let $\hat{X}$ denote the normalized activations. It is defined by equation (39). [1] [9]

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

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] [9]

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

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] [9]

Let μ_{EWMA_c} and $\sigma^2_{EWMA_c}$ 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):

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

$$\sigma^2_{EWMA_c} \to \lambda\sigma^2_c + (1-\lambda)\sigma^2_{EWMA_c} \tag{42}$$

3.2.9 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: [11] [13]

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

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

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

The backpropagation of the recursive delta-error $\delta^{(l+1)}$ is derived by the use of the chain rule. $\frac{\partial L(W)}{\partial X^{(l)}}$ 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. [11] [13]

$$\begin{aligned}\delta^{(l)}_{r,c,h,w} &= \frac{\partial L(W)}{\partial X^{(l)}_{r,c,h,w}} \\ &= \sum_{r'=0}^{R'-1}\sum_{c'=0}^{C'-1}\sum_{h'=0}^{H'-1}\sum_{w'=0}^{W'-1} \frac{\partial L(W)}{\partial X^{(l+1)}_{r',c',h',w'}} \frac{\partial X^{(l+1)}_{r',c',h',w'}}{\partial \hat{X}_{r',c',h',w'}} \frac{\partial \hat{X}_{r',c',h',w'}}{\partial X^{(l)}_{r,c,h,w}}\end{aligned} \tag{46}$$

$\frac{\partial L(W)}{\partial X^{(l+1)}_{r,c,h,w}}$ 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): [11] [13]

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

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 equation (39) and then using the product rule: [11] [13]

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

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): [11] [13]

$$\begin{aligned}\frac{\partial(X^{(l)}_{r',c',h',w'} - \mu_{c'})}{\partial X^{(l)}_{r,c,h,w}} &= \frac{\partial(X^{(l)}_{r',c',h',w'} - \frac{1}{RHW}\sum\limits_{r''=0}^{R-1}\sum\limits_{h''=0}^{H-1}\sum\limits_{w''=0}^{W-1} X^{(l)}_{r'',c',h'',w''})}{\partial X^{(l)}_{r,c,h,w}} \\ &= I_{r',r}I_{c',c}I_{h',h}I_{w',w} - \frac{1}{RHW}I_{c',c}\end{aligned} \tag{49}$$

The derivative of the second factor with respect to the activation is found in the same way as the previous equation. The batch variance $\sigma_{c'}^2$ is substituted with its definition from equation (38) and then using the kronecker-delta together with the chain rule: [11] [13]

$$
\begin{aligned}
\frac{\partial \sigma_{c'}^2}{\partial X_{r,c,h,w}^{(l)}} &= \frac{\partial \frac{1}{RHW} \sum\limits_{r'=0}^{R-1} \sum\limits_{h'=0}^{H-1} \sum\limits_{w'=0}^{W-1} (X_{r',c',h',w'}^{(l)} - \mu_{c'})^2}{\partial X_{r,c,h,w}^{(l)}} \\
&= \frac{1}{RHW} \sum_{r'=0}^{R-1} \sum_{h'=0}^{H-1} \sum_{w'=0}^{W-1} 2(X_{r',c',h',w'}^{(l)} - \mu_{c'})(I_{r',r} I_{c',c} I_{h',h} I_{w',w} - \frac{1}{RHW} I_{c',c}) \\
&= \frac{2}{RHW}(X_{r,c',h,w}^{(l)} - \mu_{c'}) I_{c',c} - \frac{2}{(RHW)^2} \sum_{r'=0}^{R-1} \sum_{h'=0}^{H-1} \sum_{w'=0}^{W-1} (X_{r',c,h',w'}^{(l)} - \mu_c) \\
&= \frac{2}{RHW}(X_{r,c',h,w}^{(l)} - \mu_{c'}) I_{c',c}
\end{aligned}
\tag{50}
$$

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:

$$
\begin{aligned}
\delta^{(l)}_{r,c,h,w} &= \sum_{r'=0}^{R-1}\sum_{c'=0}^{C'-1}\sum_{h'=0}^{H'-1}\sum_{w'=0}^{W'-1} \frac{\partial L(W)}{\partial X^{(l+1)}_{r',c',h',w'}} \frac{\partial X^{(l+1)}_{r',c',h',w'}}{\partial \hat{X}_{r',c',h',w'}} \frac{\partial \hat{X}_{r',c',h',w'}}{\partial X^{(l)}_{r,c,h,w}} \\
&= \sum_{r',c',h',w'} \delta^{(l+1)}_{r',c',h',w'} \gamma^{(l)}_{c'} (\sigma^2_{c'})^{-\frac{1}{2}} (I_{r',r} I_{c',c} I_{h',h} I_{w',w} - \frac{1}{RHW} I_{c',c}) \\
&\quad - \sum_{r',c',h',w'} \delta^{(l+1)}_{r',c',h',w'} \gamma^{(l)}_{c'} \frac{1}{RHW} (X^{(l)}_{r',c',h',w'} - \mu_{c'})(X^{(l)}_{r,c',h,w} - \mu_{c'})(\sigma^2_{c'})^{-\frac{3}{2}} I_{c',c} \\
&= \delta^{(l+1)}_{r,c,h,w} \gamma^{(l)}_{c} (\sigma^2_{c})^{-\frac{1}{2}} - \frac{1}{RHW} \sum_{r',h',w'} \delta^{(l+1)}_{r',c,h',w'} \gamma^{(l)}_{c} (\sigma^2_{c})^{-\frac{1}{2}} \\
&\quad - \frac{1}{RHW} \sum_{r',h',w'} \delta^{(l+1)}_{r',c,h',w'} \gamma^{(l)}_{c} (X^{(l)}_{r',c,h',w'} - \mu_{c'})(X^{(l)}_{r,c,h,w} - \mu_{c})(\sigma^2_{c})^{-\frac{3}{2}} \\
&= \frac{1}{RHW} \gamma^{(l)}_{c} (\sigma^2_{c})^{-\frac{1}{2}} \Bigg(RHW \delta^{(l+1)}_{r,c,h,w} - \sum_{r',h',w'} \delta^{(l+1)}_{r',c,h',w'} \\
&\quad - (X^{(l)}_{r,c,h,w} - \mu_{c})(\sigma^2_{c})^{-\frac{3}{2}} \sum_{r',h',w'} \delta^{(l+1)}_{r',c,h',w'} (X^{(l)}_{r',c,h',w'} - \mu_{c'}) \Bigg)
\end{aligned} \tag{51}
$$

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). [11] [13]

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

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

3.2.10 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):

$$
X_{r,c}^{(l+1)} = \frac{e^{X_{r,c}^{(l)}}}{\sum_{c'=0}^{C-1} e^{X_{r,c'}^{(l)}}} \tag{54}
$$

3.2.11 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] [3] [15]

$$\begin{aligned}
\delta_{r,c}^{(l)} &= \frac{\partial L(W)}{\partial X_{r,c}^{(l)}} \\
&= \sum_{c'=0}^{C-1} \frac{\partial L(W)}{\partial X_{r,c'}^{(l+1)}} \frac{\partial X_{r,c'}^{(l+1)}}{\partial X_{r,c}^{(l)}} \\
&= \sum_{c'=0}^{C-1} \delta_{r,c}^{(l+1)} \left(\frac{(e^{X_{r,c'}^{(l+1)}}) I_{c',c}}{\sum_{c''=0}^{C-1} e^{X_{r,c''}^{(l+1)}}} - \frac{(e^{X_{r,c'}^{(l+1)}})(e^{X_{r,c}^{(l+1)}})}{(\sum_{c''=0}^{C-1} e^{X_{r,c''}^{(l+1)}})^2} \right) \\
&= \sum_{c'=0}^{C-1} \delta_{r,c}^{(l+1)} X_{r,c'}^{(l+1)} (I_{c',c} - X_{r,c}^{(l+1)}) \\
&= \delta_{r,c}^{(l+1)} X_{r,c}^{(l+1)} \left(1 - \sum_{c'=0}^{C-1} X_{r,c'}^{(l+1)} \right)
\end{aligned} \tag{55}$$

3.3 Problem Cases

The code for All kod till de exempel på praktiska tillämpningar kan hittas på github: `https://github.com/nikitazozoulenko`

3.3.1 Classification of handwritten digits

En enkel CNN-modell kan användas för att klassificera handskriva siffror. För att uppnå detta har MNISTdatabasen för handskriva siffror används. Den består av 60 000 unika exempel av handskrivna siffror. [16]

Låt modellens prognos betecknas med $\hat{y}$. Aktiveringsfunktionen softmax används i det sista lagret för att gränsa värdena till intervallet $[0, 1]$ och har egenskapen att alla prognostiserade värdena i ett exempel summeras till 1. Följaktligen kan varje värde i $\hat{y}$ tolkas som sannolikheten att bilden är av varje klass. [1]

Input för modellen är en $R \times 1 \times 28 \times 28$ tensor där R står för minihopstorleken. Modellen prognostiserar tio värden per bild i form av en tensor $\hat{y} \in \mathbb{R}^{R \times C}$, ett värde för varje klass $C = 10$ av siffra. Konstandsfunktionen som har minimerats under träningstiden är funktionen *cross-entropy* betecknat med L. Den verkar på två sannolikhetsfördelningar: De verkliga

Figure 10: Tio bilder av handskrivna siffror från MNISTdatabasen. [16]

sannolikheterna y och de prognostiserade sannolikheterna $\hat{y}$: [1] [3]

$$L(W) = -\sum_{r=0}^{R-1}\sum_{c=0}^{C-1} y_{r,c} \; \log \hat{y}_{r,c} \tag{56}$$

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

Endast en modell tränades på grund av begränsningar i datakraft. Den bestod av tre stycken konvolutionsblock, följt med ett softmaxlager (se figur 10). En mini-hopstorlek av 50 användes och 5000 iterationer av framåt- och bakåtpropagering kördes på min egenimplementerade modell skriven i python. Totalt tog det 4 timmar att träna nätverket på min CPU.

Den slutgiltiga precisionen blev 99.2% på 10 000 nya handskrivna siffror modellen aldrig hade sett tidigare. Av 10 000 handskrivna siffror lyckades modellen klassifisera 9 919 siffror rätt.

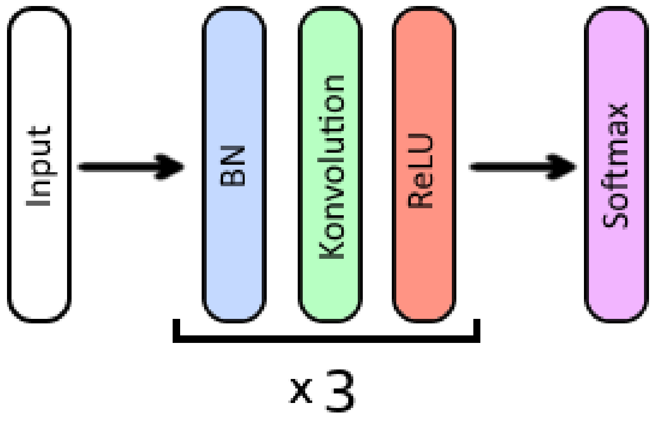

Figure 11: Modellen som användes för att läsa av handskrivna siffror. Batch Normalization benämns med BN, konvolution menas en konvolution med kärnstorlek k = 5 och stride s = 2. ReLU är aktiveringsfunktionen Rectified Linear Units och softmax är softmaxlagret.

3.3.2 Dense Face Detection and Localization

Ett konvolutionellt neuralt nätverk är anpassat för att detektera upp till flera tusen ansikten åt gången för realtidsvideo. Modellen producerar ett bestämt antal *boudning boxes*: koordinater som ska detektera alla olika objekt i bilden. Jag utgick från architekturen från RetinaNet från *Focal Loss for Dense Object Detection* (Lin et al.) och anpassade den till $K = 1$ klasser. I varje position i outputlagret föreslår modellen en *boudning box* med fyra koordinater: två punkter för det övre vänstra hörnet respektive nedre hörna hörnet av den positionens *bounding box*. Utöver det förutsägs det $K+1$ sannolikheter att *bounding boxen* innehåller ett objekt av alla K förgrundklasser och 1 bakgrundklass (lådan innehåller inga objekt).[1][23]

Modellen för objektdetektering går ut på att man utgår från ett antal så kallade anchor boxes vid varje rumslig position i det sista konvolutionella lagret. Om lagret har bredden och höjden W respektive H och A olika storlekar på anchor boxes har lagret totalt WHA olika anchor boxes. Vid träning tilldelas en anchor box ett objekt om dens *intersection over union* (IoU) med den verkliga bounding boxen är större än 0.5 (se figur 11 IoU används för att beräkna hur lika två olika mängder är. I detta fall är mängderna areorna av en anchor box och ett objekts verkliga bounding box. IoU definieras som storleken av snittet av två mänger A och B dividerat med storleken av unionen av A och B: [1] [22]

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

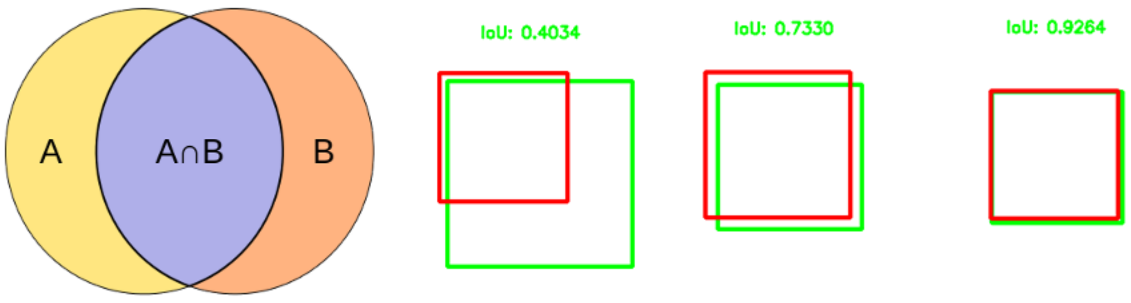

Figure 12: IoU definieras som storleken av snittet av två mänger A och B dividerat med storleken av unionen av A och B. En större IoU medför att prognosen är närmare sanningen. [22]

RetinaNet använder sig av en *feature pyramid*-arkitektur beskriven i *"Feature Pyramid Networks for Object Detection"* (Lin et al.) och en ResNet101 (He et al.) som *backbone*. Featurepyramiden används för att få ut *feature maps* från olika delar av det konvolutionella neurala nätverket för att kunna detektera objekt av olika storlekar. Varje pyramidnivå matas in i ett klassifikationshuvud respektive regressionshuvud (se figur 12) för att räkna ut sannolikheterna att varje anchor box innehåller de $K+1$ olika klasserna, samt att förfina varje anchor box koordinater genom att prognostisera 4 olika offsets för varje sida av anchor boxens bounding box. [21] [23] [24]

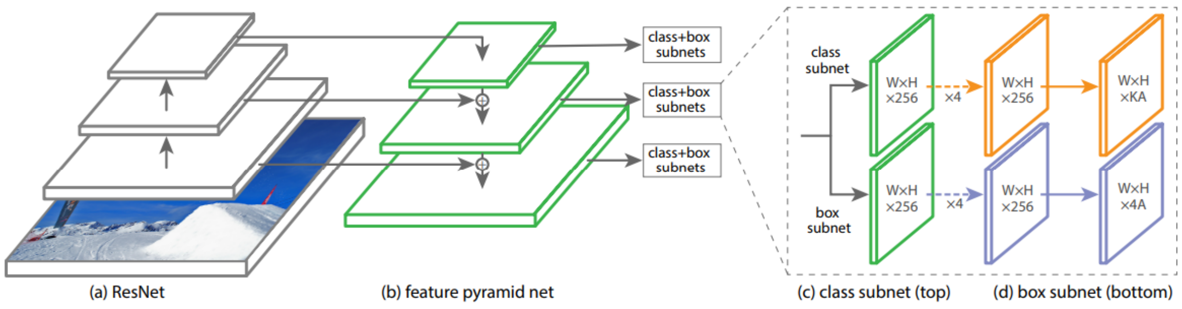

Figure 13: RetinaNets arkitektur från *Focal Loss for Dense Object Detection* (Lin et al.). Varje pyramidnivå matas in i ett klassifikationshuvud och ett regressionshuvud. [23]

Till skillnad från RetinaNet har min modell, FaceNet, ett mindre antal pyramidnivåer för att lätta på beräkningskraften och för att detektera mindre objekt. RetinaNets anchors har areor från 32^2 till 512^2 pixlar spridda över pyramidnivåer P3 till P7. FaceNet använder sig av areor 14^2 till 220^2 pixlar spridda över pyramidnivåer P3 till P6. Regressionshuvudet och klassifikationshuvudet i RetinaNet består av 5 konvolutionslager med kärnstorlek 3. Facenet använder istället 2 och 3 konvolutionslager för regressionshuvudet respektive klassifikationshuvudet. I det originala Feature Pyramidnätverket användes linjär interpolation för att göra storleken av högre feature maps större. I FaceNet används k-nearest neighbors (k-NN) istället. ResNet50 användes istället för ett ResNet101 som backbone för att lätta på beräkningskostnaderna. [21] [23] [24]

Kostnadsfunktionen som användes var en kombination av Focal Loss (Lin et al.) och Smooth L1 Loss. FaceNet använder samma hyperparametrar som uppnådde bästa resultat för RetinaNet: $\alpha = 3$ för förgrundklassen och $\gamma = 2$. [23]

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

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

$$
\begin{aligned}
L(W) = &\sum_{k \in pyramid} \frac{1}{N_c^k} \sum_{a \in anchors} L_r(r_a - \hat{r}_a) \\
&+ \sum_{k \in pyramid} \frac{1}{N_c^r} \sum_{a \in anchors} L_c(p_a, \hat{p}_a)
\end{aligned}
\tag{61}
$$

N_c^k och N_r^k betäcknar alla anchors som deltar i klassifikationskonstnaden respektive regressionskonstnaden i pyramidnivå k. För klassifikationskonstnaden är $\hat{p}$ 1 för alla anchors som har blivit tillgivna ett objekt och 0 för alla som bara innehåller bakgrundsklassen. Endast anchors som innehåller ett objekt deltar i regressionskostnaden. r_a innehåller de 4 koordinaterna den slutgiltiga boudning boxen har.

Modellen tränades i 350 000 iterationer med en mini-hopsstorlek av 3. Träningshastigheten började på 0.005 och dividerades med 10 vid 200 000 iterationer. Varje träningsexempels storlek gjordes slumpmässigt om till 512^2, 576^2 eller 640^2 pixlar och speglades horizontellt med sannolikhet 0,5 för att artificiellt utöka mängden träningsdata. Den totala träningstiden var 18 timmar på en NVIDIA GTX 1080ti. Figur 13 visar kvantitativa resultat från WIDERFace valideringsdata.

En iteration av framåtpropageringen tar 20 ms vilket möjliggör att modellen kan köras i realtid, 50 gånger i sekunden.

4 Conclusion

Vad är ett konvolutionellt neuralt nätverk? Hur härleds framåt- och bakåtpropagering i konvolutionella neurala nätverk? Hur kan modellen tillämpas för att implementera sifferavläsning, ansiktsigenkänning och objektdetektering i bilder?

Den matematiska modellen av artificiella neurala nätverk och konvolutionella neurala nätverk, samt bakåt- och framåtpropagering förklaras och härleds i sektion 3.2 och 3.3. Ett exempel på hur modellen kan tillämpas är att klassificera handskrivna siffror, som i sektion 3.4.1. En annan möjlighet är att anpassa ett konvolutionellt neuralt nätverk för att detektera alla ansikten i en digital bild. På grund av att en iteration av algoritmen tar 20 ms är det möjligt att köra modellen i realtid. Det kan exempelvis användas i övervakningskameror eller inom robotik för att känna igen människoansikten i realtid.

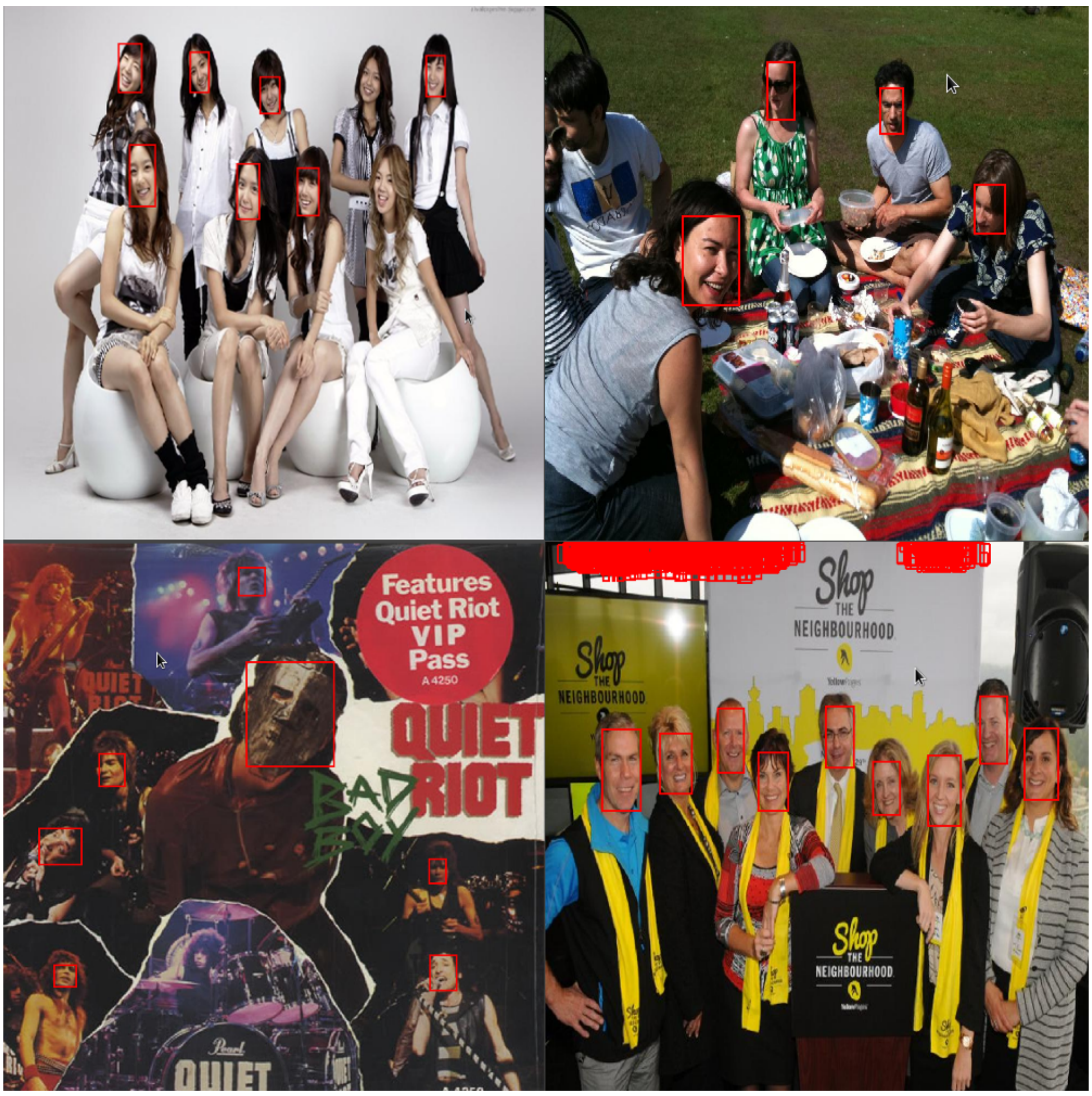

Figure 14: Kvantitativa resultat från valideringsdatan från WIDERFace: Bilder som modellen aldrig har sett tidigare.

Algoritmen är dock inte perfekt och har flera brister. Modellen prognostiserar att det finns ansikten på flera ställen där det inte finns några i verkligheten. För att undankomma detta låter jag endast modellen visa upp bounding boxes med sannolikhet större än 0.5 att det finns ett ansikte där. Resultatet blir att modellen inte hittar 100% av all ansikten i bilden (se figur 13). Ytterligare har modellen problem med små ansikten. Detta är dels på grund av att modellen inte kan urskilja ett ansikte med så lite information (ca minimum 10^2 pixlar per ansikte), och dels för att FaceNet prognostiserar ca 30 000 olika anchors för de minsta storlekarna. Detta leder till att klassifikationskostnaden består av till mestadels negativa exempel. Delta-felet av bakgrundsklasserna dominerar och gör att delta-felet av förgrundsklassen (ansikten) inte har en stor påverkan på den slutgiltiga gradienten.

References

[1] *CS231n: Convolutional Neural Networks for Visual Recognition.* F. Li, A. Karpathy och J. Johnson. Stanford University, föreläsning, vinter 2016.

[2] *Unsupervised Feature Learning and Deep Learning.* Standford University, Department of Computer Science. URL http://ufldl.stanford.edu/wiki/. Senast uppdaterad 31 mars 2013.

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

[4] *Introduction to Convolutional Neural Networks.* J. Wu. National Key Lab for Novel Software Technology, Nanjing University, Kina. 1 maj, 2017.

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

[6] *High Performance Convolutional Neural Networks for Document Processing.* K. Chellapilla, S. Puri, P. Simard. Tenth International Workshop on Frontiers in Handwriting Recognition. La Baule, Frankrike, Suvisoft. Oktober 2006.

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

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

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

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

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

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

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

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

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

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

[17] *Pygradsc.* J. Komoroske. URL https://github.com/joshdk/pygradesc. 12 oktober 2012.

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

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

[20] *Deep learning for complete beginners: convolutional neural networks with keras.* P. Veličković. Camebridge Spark. URL https://cambridgespark.com/content. Senast uppdaterad 20 mars 2017.

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

[22] Jaccard Index. Wikipedia. URL https://en.wikipedia.org/wiki/Jaccard_index. Hämtad 20 januari 2018

[23] *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

[24] *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

[25] *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