SAVONIA

Thesis – Bachelor's Degree Programme

Technology, Communication and Transport

NEURAL NETWORKS AND THEIR INTERNAL PROCESSES: FROM THE GROUND UP

AUTHOR/S: Ilia Kutenkov

Field of Study Technology, Communication and Transport	
Degree Programme Bachelor's Degree Programme in Information Technology, Internet of Things	
Author(s) Ilia Kutenkov	
Title of Thesis Neural Networks and Their Internal Processes: From The Ground Up	
Date 30 June 2022	Pages/Appendices 64/1
Client Organisation /Partners University of Turku	

Abstract
The theses' aim is to explain and practically show the operation of different types of neural networks. Thesis will prove that they all have the same main idea at the core but have different goals, methods and components. The thesis also shows a process of preparing a custom object detection dataset that could be used for training neural networks.

Author decided to show the neural networks from the ground up. First, he is explaining an abstract operation of a chosen tool. Then, a mathematical explanation is written. Finally, the code implementation or use case of a tool is shown.

As a result, thesis has a lot of information about neural networks with a code examples associated. It also contains an explanation of processes that happen in neural network during their forward and backward operations. 3 different datasets were used to train neural networks. The thesis' goals were successfully achieved.

Keywords
Neural network, convolution, image classification, object detection, ANN, CNN, dataset, Python, NumPy, TensorFlow

CONTENTS

1 INTRODUCTION

For many years people have been automating their jobs using programming. At the beginning, computers were used to automatically compute massive amounts of mathematical operations, such as addition, subtraction, multiplication and division at an incomparable speed as compared to human beings. As time passed, people were able to increase computational speeds a lot. Despite this, programming techniques were also evolving. As a result, the computer won a chess game against world champion, Garry Kasparov, on February 10, 1996 (Dennis DeCoste 1997, 1).

However, computers have been only following instructions given by a human. This has been limiting possibilities a lot. People were haunting a solution for computers to actually "learn" to solve a problem without a predefined plan. This resulted in the creation of a whole new world inside of computer science: neural computations. Later, it will progress to a field, currently known as machine learning.

The main idea of this thesis is to take a look at the processes that happen inside of different neural networks. First, the thesis will show a process of creating an artificial neural network using only mathematical functions. The model will be able to classify 28x28 pixel grayscale pictures of handwritten digits. Second, it will exhibit a way to use ready-made building blocks in order to create a convolutional neural network that is going to be able to classify a 32x32 pixel RGB image into 10 different classes: car, cat, dog, frog, etc. Third, it will show a process of creating and preparing complex datasets for training neural networks using already existing network.

2 ARTIFICIAL NEURAL NETWORKS

2.1 Overview

2.1.1 Architecture

Artificial neural networks are the simplest form of neural networks. It consists of three parts (see Figure 1): input nodes, hidden nodes, and output nodes. All nodes of subsequent layers are interconnected between each other. Each connection has a special parameter, called "weight". Each node has a special parameter called "bias". Normally, each layer is assigned its own activation function that is giving them a preferred logic.

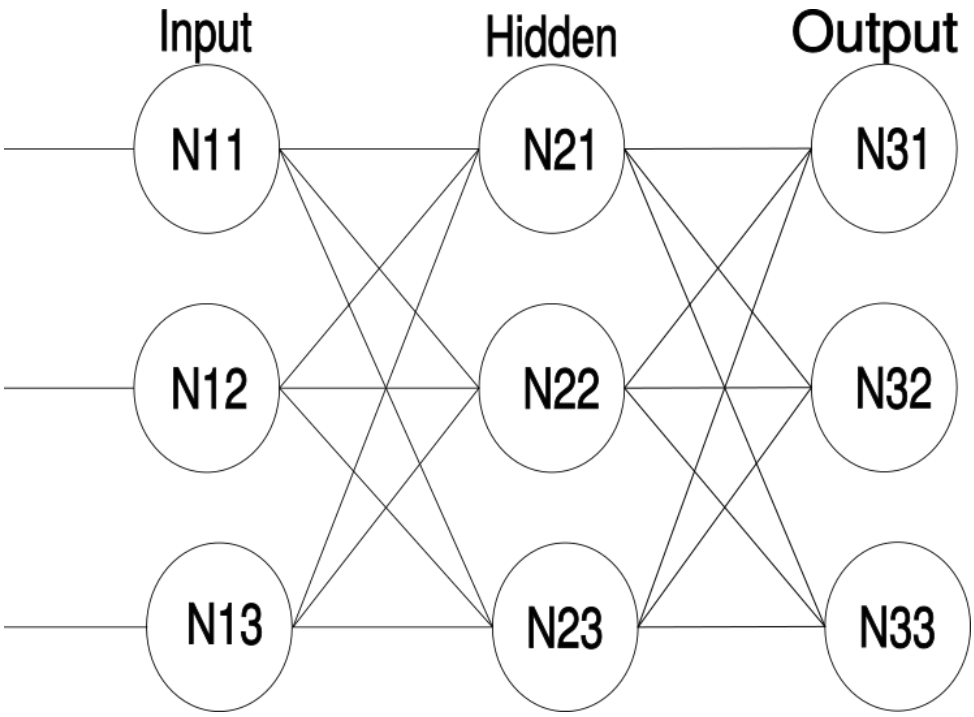

FIGURE 1. Artificial neural network

2.1.2 Similarity to actual neuron

The artificial neural network node is very similar to the actual neuron (see Figure 2) in a human brain: inputs to a soma (node/artificial neuron) are provided by dendrites (input connections), which are connected to the axon terminals (output connections) via synapses (adjacent weighting/biasing) of previous neurons. The axon itself may be represented as an activation function.

Actual neurons do not have a perfectly similar structure as an artificial neural network. Some neurons are connected recursively. Some axon terminals and dendrites are not connected to anything at all. New data (irritation) may be inputted to different parts of a network, the signal may come asynchronously, and so on. (Sidiropoulou, Kyriaki and Poirazi 2006, 3(888)-4(889))

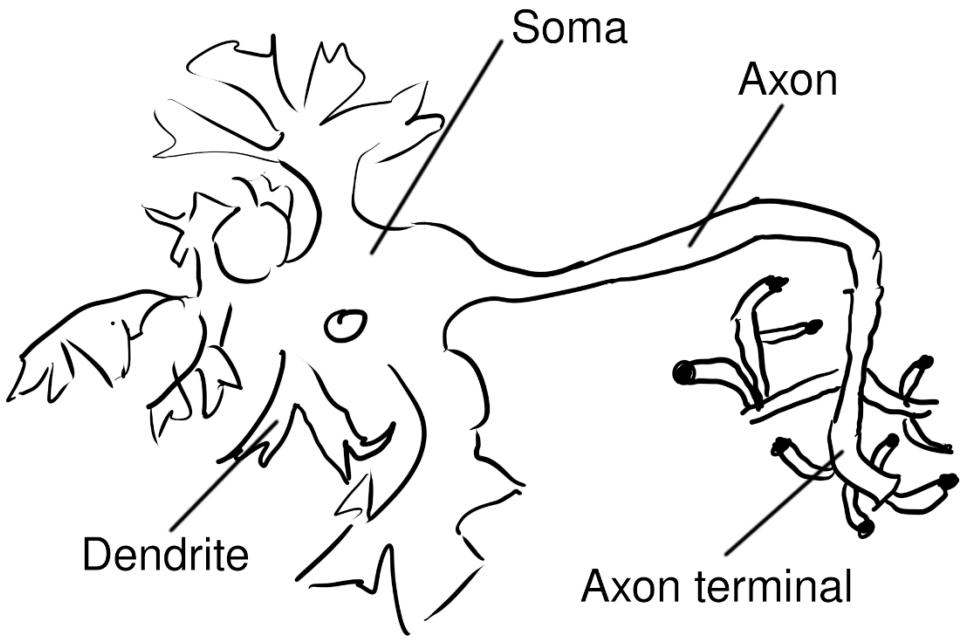

FIGURE 2. Depicture of an actual neuron inside of human brain

2.2 Components

All the neural networks consist of small parts that communicate in a specified way. In this section there is going to be explained the main parts that artificial neural networks are made of.

2.2.1 Neuron

The main part of a neural network is called a neuron. It may be represented as a mathematical function that takes some input, applies some logic to it, and produces an output. Most of the neurons inside of a neural network are getting inputs from the output of another neuron. In artificial neural networks, only the input layer neurons are getting information from different type of source. Neural network learns to predict by changing its internal parameters: input weights and biases (Schmidhuber 2014, 4).

2.2.1.1 Weights

All the inputs to a neuron are multiplied with the corresponding weights. The bigger the weight, the more impact will the corresponding input produce on behalf of the neuron's output.

2.2.1.2 Bias

Each neuron has its own single bias. The bias is simply added to the weighted input's sum. It helps to adjust neuron's behaviour separately from inputs.

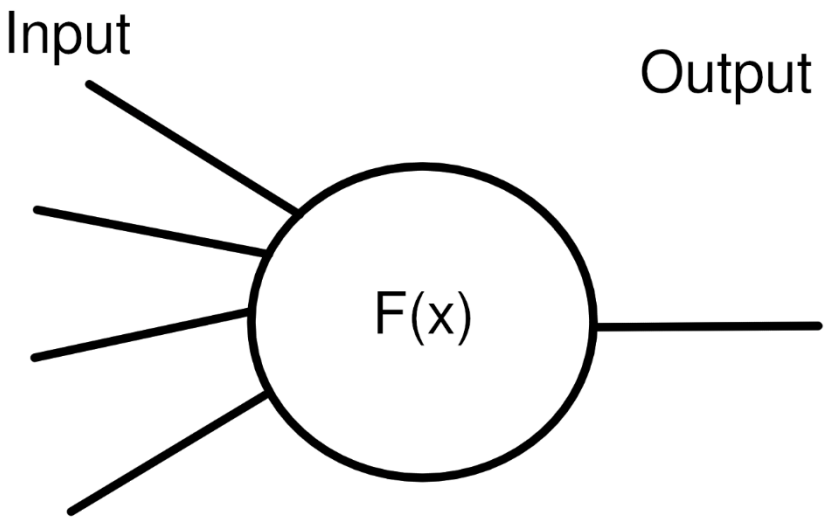

FIGURE 3. Depicture of an artificial neuron

2.2.2 Layer

Artificial neural networks have a layered structure. It means that neurons are held and connected in batches. A layer may consist of 1 or more neurons. The main idea of adding more layers is to make a neural network to learn more complicated relationships inside of the input data. It could be correlated to nature: on average, the bigger the brain, the more complicated behaviour an animal has. Neural network engineers do not really understand which function does each layer accomplishes in a huge neural network, same as in a real human brain. In modern approach, developers are using layers as building blocks of a neural network.

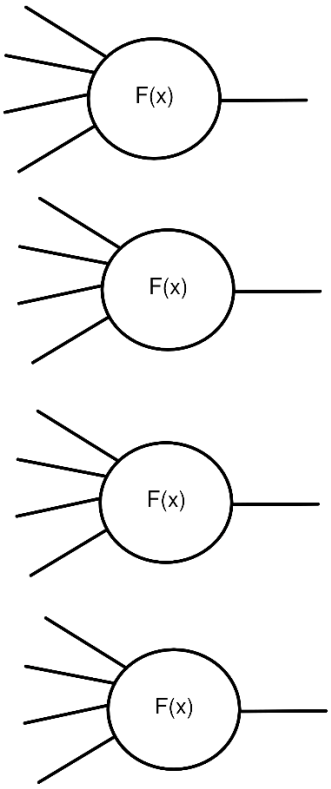

FIGURE 4. A neural network layer consisting of 4 artificial neurons

2.2.2.1 Activation function

Each layer has its own activation function. It is applied to every single neuron's output. It is mainly used to normalise the neuron's output and/or add nonlinearity to the whole model (Szandała 2010, 1-2). The most widely used activation functions nowadays are: ReLu, LeakyReLu, Linear, SoftMax and Sigmoid.

The code that was used for generating the graphs could be found in appendix #1.

2.2.2.1.1 ReLu

ReLu activation function simply passes further the input if it is bigger than 0, or 0 if the input value is <0. It is a very simple, yet effective activation function that is used to add nonlinearity to the network.

The formula is: $ReLu(x) = \begin{cases} x, & if\ x > 0 \\ 0, & if\ x \leq 0 \end{cases}$ (Nwankpa et al. 2018, 8).

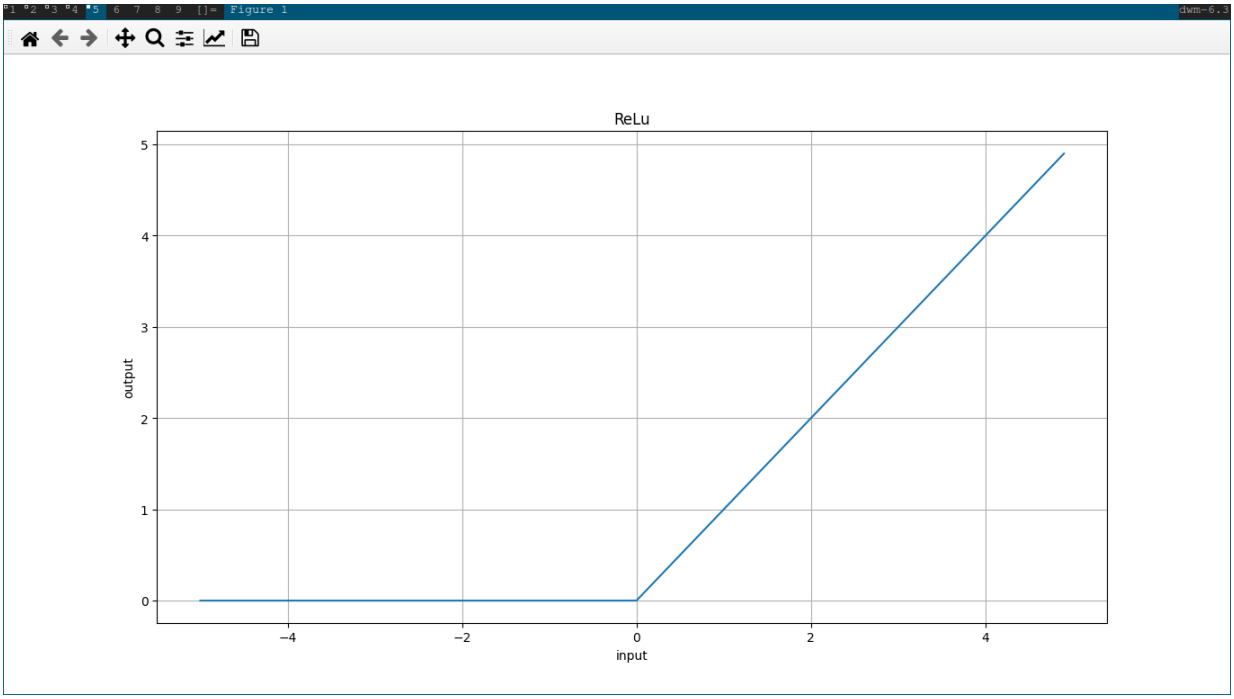

FIGURE 5. A graph of ReLu

2.2.2.1.2 Linear

Linear activation is the simplest activation function. It outputs the input value multiplied by predefined constant. As the name suggests, it does not introduce nonlinearity to the system. The formula is: $f(x) = a * x$ (Szandała 2020, 4).

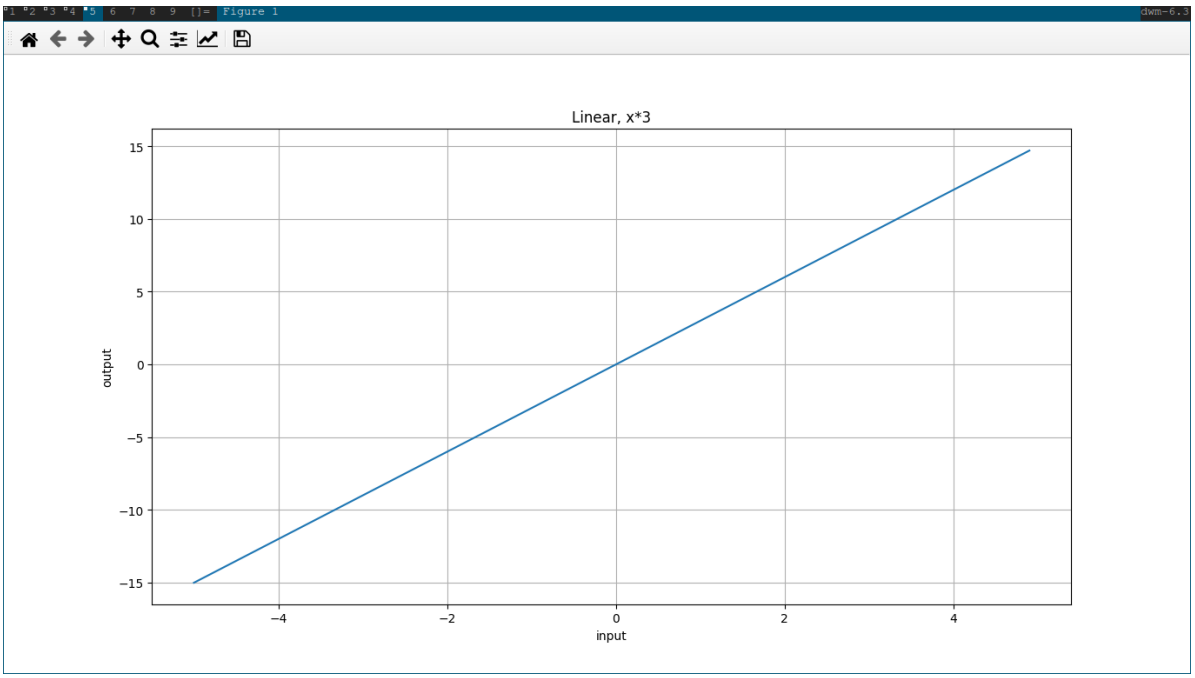

FIGURE 6. A graph of linear function

2.2.2.1.3 Sigmoid

The function of a Sigmoid is $\frac{1}{1+e^{-x}}$ (Nwankpa, et al. 2018, 5). It ensures that the output is always in bounds of 0 and 1. The output is 0.5 at x = 0, 1 at x = ∞, and 0 at x = - ∞. It is mainly used to normalise the output.

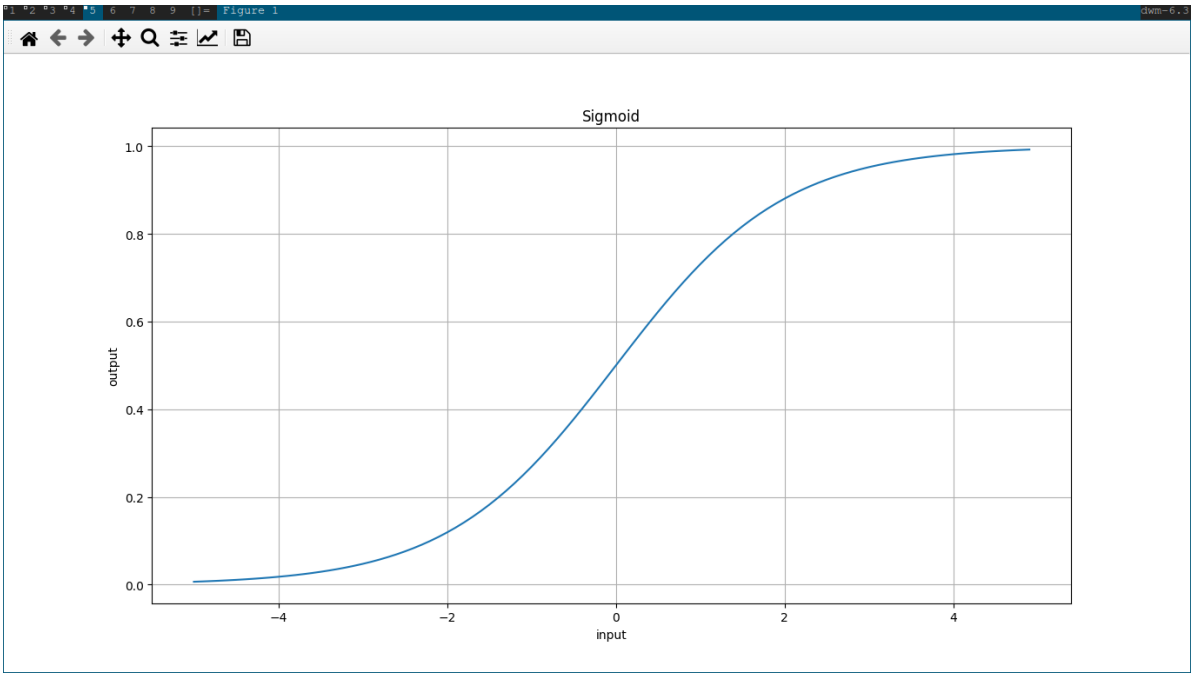

FIGURE 7. A graph of a Sigmoid activation function

2.2.2.1.4 SoftMax

SoftMax is mainly implemented in the output layer. It is used to produce confidence-type output in classifying neural networks. The formal name for this type of output is called probability distribution. For example, an output layer may have two neurons: cat and dog. SoftMax will normalize and scale

outputs so that they represent fraction of an exponentiated total sum, e.g. certainty if the input is a cat or a dog. All the output values will sum up to 1. The function is $S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$. (Nwankpa et al. 2018, 8)

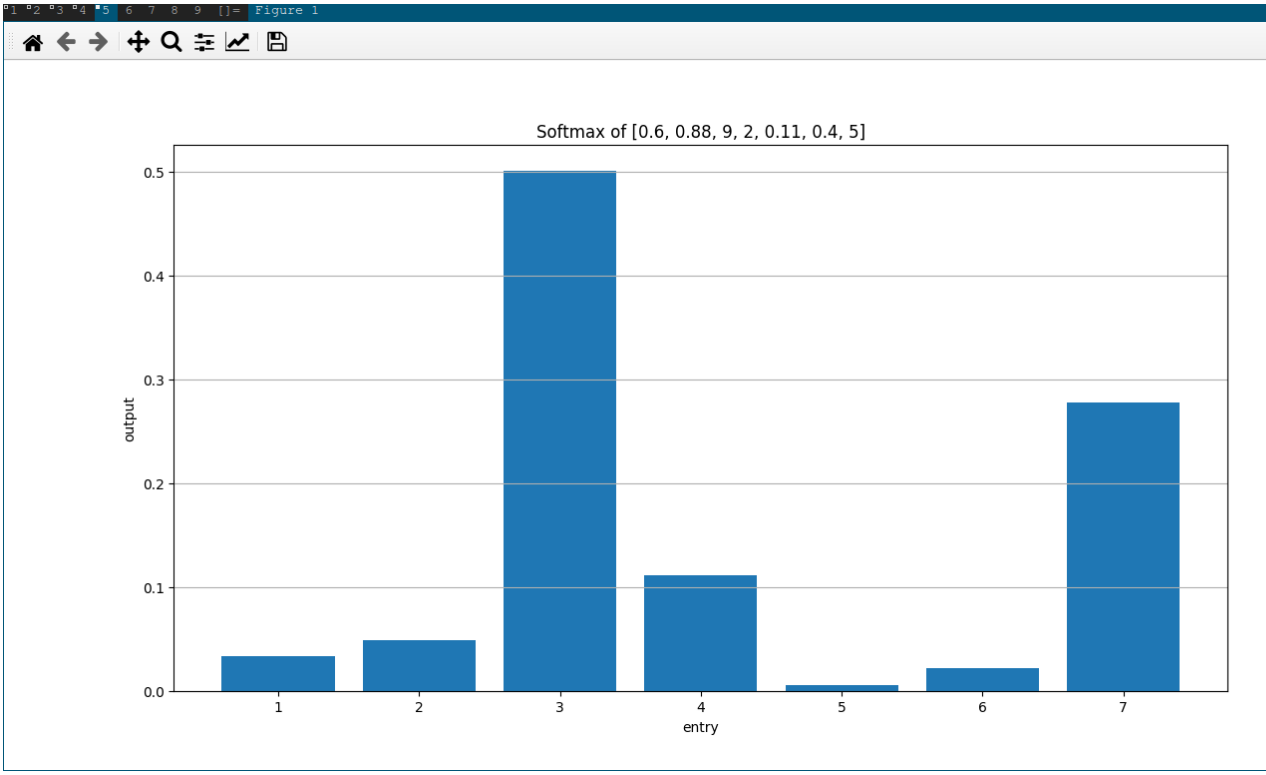

FIGURE 8. A bar plot of Sigmoid activation function. X-axis is an entry index

2.2.3 Network

A network consists of two or more layers connected to each other. In Figure 9 it is shown that each neuron of the first layer is connected to all the neurons in the second layer.

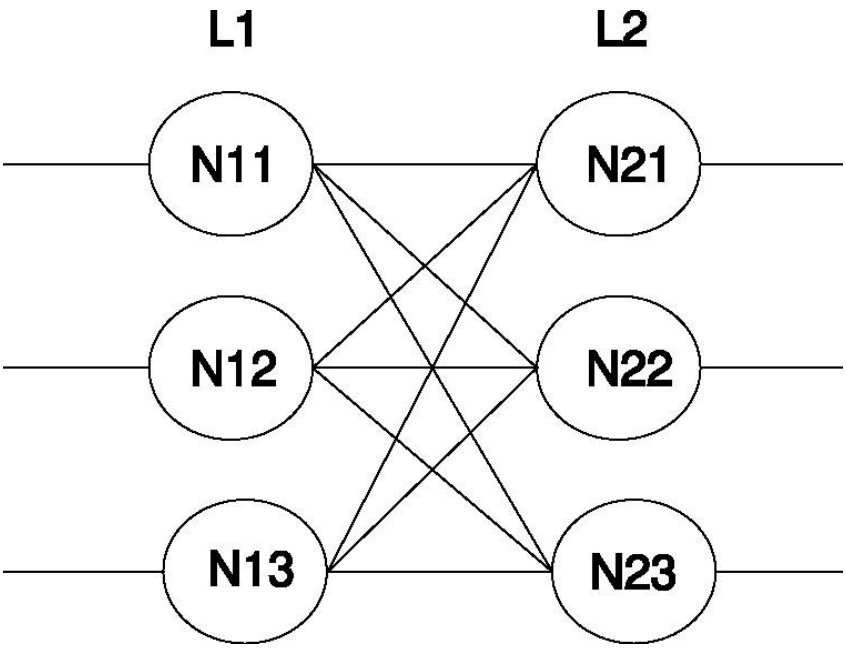

FIGURE 9. Neural network

2.3 Mathematical representation of operation

This section shows the actual operation of a neural network. Batching will not be considered for the sake of simplicity.

2.3.1 Forward operation

Everything in a neural network is represented as scalars (single numbers), vectors (1-dimensional arrays), matrices (2-dimensional arrays), and n-dimensional tensors (3 and more dimensional arrays).

The input to a neural network's dense layer is put into a 1-dimensional array. So, the input is going to be a vector, where each element is an input value. Example: [1, 2, 3].

Weights are stored in a 2-dimensional matrix, where each row is a set of weights for one single neuron in a layer. Example: [[1,2,3], [4,5,6], [7,8,9]].

Biases are stored in a 1-dimensional vector, where each element represents a bias for certain neuron. Example: [4,2,3]

First, the layer gets an input from the previous layer. Then, all the input values are multiplied with the corresponding weights and summed up as per neuron. This is achieved by applying a widely used matrix operation named dot-product. It simply multiplies each element in 1st matrix's row by each element in 2nd matrix's column, and then sums up all the resulting numbers as per 1st matrix's row. The output shape is going to be a number of rows in 1st matrix × number of columns in 2nd matrix. Example: $\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\end{pmatrix}\cdot\begin{pmatrix}1 & 2\\3 & 4\\5 & 6\end{pmatrix}=\begin{pmatrix}1*1+2*3+3*5 & 1*2+2*4+3*6\\4*1+5*3+6*5 & 4*2+5*4+6*6\end{pmatrix}=\begin{pmatrix}22 & 28\\49 & 64\end{pmatrix}$.

Since the nature of dot product (and because it is needed to get weights as per neuron), a transpose of a weights matrix is taken.

Example of transpose operation on a vector: $(1 \quad 2 \quad 3)^T=\begin{pmatrix}1\\2\\3\end{pmatrix}$.

Example of transpose operation on a matrix: $\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\end{pmatrix}^T=\begin{pmatrix}1 & 4\\2 & 5\\3 & 6\end{pmatrix}$.

Example using defined numbers: $(1 \quad 2 \quad 3)\cdot\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{pmatrix}^T=(1 \quad 2 \quad 3)\cdot\begin{pmatrix}1 & 4 & 7\\2 & 5 & 8\\3 & 6 & 9\end{pmatrix}=$

$(14 \quad 32 \quad 50)$.

After that, it is needed to add a corresponding bias to every neuron. This is accomplished by simply adding biases element-wise to a result: $(14 \quad 32 \quad 50)+(4 \quad 2 \quad 3)=(14+4 \quad 32+2 \quad 50+3)=$

$(18 \quad 34 \quad 53)$.

The next step is applying an activation function. Let's consider that this is the output layer, and its activation function is Softmax.

The formula is: $S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$, where y is the output of neurons, i is index of certain neuron, and j is amount of neurons. $\sum_j e^{y_j}$ is going to be : $e^{18} + e^{34} + e^{53} \approx 65659969.1373 + 5.8346174 * 10^{14} + 1.0413759 * 10^{23} \approx 1.0413759 * 10^{23}$.

The numbers became irrationally large. That is why normalization is required throughout the whole neural network. The general idea of normalization is keeping values in certain bounds. The most common boundaries are -1 and 1. Another scenario: input values are multiplied by 0.01, and now are $(0.01 \quad 0.02 \quad 0.03)$. Applying calculations, node's values are now also 1000 times smaller than they were before: $(0.18 \quad 0.34 \quad 0.53)$. Now, $\sum_j e^{y_j} = e^{0.18} + e^{0.34} + e^{0.53} = 1.19721736312 + 1.40494759056 + 1.69893230862 = 4.3010972623$. This makes much more sense. The sigmoid activation function for the local case now looks like this: $S(y_i) = \frac{e^{y_i}}{4.3010972623}$. Applying activation function: $S(output) = \left(\frac{e^{0.18}}{4.3010972623} \quad \frac{e^{0.34}}{4.3010972623} \quad \frac{e^{0.53}}{4.3010972623}\right) = \left(\frac{1.19721736312}{4.3010972623} \quad \frac{1.40494759056}{4.3010972623} \quad \frac{1.69893230862}{4.3010972623}\right) = (0.27835161357 \quad 0.32664864449 \quad 0.39499974192) \approx (0.28 \quad 0.33 \quad 0.40)$.

This is an output of a layer. It could be sent to the next layer if needed or be used as an output of a network. The network predicts 1st class with 28% certainty, 2nd class with 33% certainty, and 3rd class with 40% certainty.

2.3.2 Backward operation

2.3.2.1 Loss

Loss shows how much the obtained output differs from predefined target values. There are many loss functions, but in this thesis, there will be implemented and used the most common one: Categorical Cross Entropy Loss (Feng et al. 2020, 2(2207)). It measures the deviation of a classifying neural network. Cross Entropy Loss requires the input values to be in range of 0-1 and represent probability distribution. The formula is: $CELoss(y, \hat{y}) = -\sum_{i=0}^{len(y)} y_i * log_{10}(\hat{y}_j)$, where y are target values, and $\hat{y}$ are predicted values. The bigger the deviation, the bigger the loss value. $log_{10}(n)$ is negative when n is smaller than 1, so the sum must be multiplied by -1 in the end in order to get a positive loss value. If target values are one-hot encoded, and there is only one correct target class, then it is possible to simplify the function a lot: $CELoss(y_{\text{right}}, \hat{y}_{\text{right}}) = -1 * log_{10}(\hat{y}_{right})$.

Let's say that the target class for the described previously forward pass was class #2. In one-hot encoded form, it is going to be represented as: $y = (0 \quad 1 \quad 0)$. Now, using a simplified formula, the loss value is: $CELoss(1, 0.33) = -1 * log_{10}(0.33) = -1 * (-0.48148606012) \approx 0.48$.

2.3.2.2 Gradient propagation

The purpose of neural network's learning is to give more correct outputs, e.g. decrease the loss value. In order to do that, the network needs to find out how does every single changeable parameter (weights and biases) is affecting the loss. This is done by taking derivatives of every parameter over the loss value. The gradient, e.g. derivatives, are propagated in reverse order, so it is needed to derivate the loss function's inputs first. The derivative of a categorical cross entropy loss is as follows: $\frac{dinput}{dloss} = -1 * \left(\frac{y}{\hat{y}}\right)$ (Feng et al. 2020, 2(2207)). Applying it to the obtained loss value: $\frac{dinput}{dloss} = \left(0 \quad -1 * \left(\frac{1}{0.33}\right) \quad 0\right) = (0 \quad -3.03 \quad 0)$.

The next step is to propagate the gradient back to the previous action. The last step of a layer operation was an activation function. It is needed to propagate derivation through it, since it takes effect on the final output. The derivative of a SoftMax function is:

$\frac{doutput_i}{dinput_j} = \begin{cases} output_i * (1 - output_j) & if \quad i = j \\ -output_j * output_i & if \quad i \neq j \end{cases}$ (Bendersky 2016). Applying it to the local case values, a matrix is obtained:

$$\frac{doutput}{dinput} =$$

$$= \begin{pmatrix} 0.28 * (1 - 0.28) & -0.34 * 0.28 & -0.40 * 0.28 \\ -0.28 * 0.33 & 0.33 * (1 - 0.33) & -0.40 * 0.33 \\ -0.28 * 0.40 & -0.33 * 0.40 & 0.40 * (1 - 0.40) \end{pmatrix} = \begin{pmatrix} 0.2016 & -0.0924 & -0.1120 \\ -0.0924 & 0.2211 & -0.1320 \\ -0.1120 & -0.1320 & 0.2400 \end{pmatrix}$$

This is an output gradient for every output value. Then, it is needed to apply a dot product between an obtained Jacobean matrix and the next gradient in order to get a gradient as per neuron for the whole set:

$$\frac{doutput}{dloss} = \begin{pmatrix} 0.2016 & -0.0924 & -0.1120 \\ -0.0924 & 0.2211 & -0.1320 \\ -0.1120 & -0.1320 & 0.2400 \end{pmatrix} \cdot (0 \quad -3.03 \quad 0)^T =$$

$$= \begin{pmatrix} 0.2016 & -0.0924 & -0.1120 \\ -0.0924 & 0.2211 & -0.1320 \\ -0.1120 & -0.1320 & 0.2400 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ -3.03 \\ 0 \end{pmatrix} = (0.279972 \quad -0.669933 \quad 0.39996)$$

The next step is to derivate weights and biases. Derivation rules say that the derivative of a multiplication that involves variable in 1st power and a constant is a constant itself. E.g. $y = 5xz; \frac{dx}{dy} = 5 * 1 * x^{1-1} * z = 5z$. Derivative of a constant is 0. Applying derivation rules to the neuron function, the equations are: $y = f(x, w, b) = x * w + b$; $\frac{dx}{dy} = w$; $\frac{dw}{dy} = x$; $\frac{db}{dy} = 1 * b^0 = 1$, where y is the output (input of the activation function), x is an input of a neuron, w are weights and b is a bias. It is also needed to propagate the gradient from the activation function down to this level. Applying formulas and gradients to the local case:

$$\frac{dw}{dloss} = x * gradient_{actfun} = (0.1 \quad 0.2 \quad 0.3)^T \cdot (0.279972 \quad -0.669933 \quad 0.39996)$$
$$= \begin{pmatrix} 0.0279972 & -0.0669933 & 0.039996 \\ 0.0559944 & -0.1339866 & 0.07992 \\ 0.0839916 & -0.2009799 & 0.119988 \end{pmatrix}$$

$$\frac{db}{dy} = (1); \frac{db}{dloss} = (1) \cdot (0.279972 \quad -0.669933 \quad 0.39996) = (0.279972 \quad -0.669933 \quad 0.39996)$$

$$\frac{dx}{dloss} = w * gradient_{actfun}^T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \cdot \begin{pmatrix} 0.279972 \\ -0.669933 \\ 0.39996 \end{pmatrix} = \begin{pmatrix} 0.139986 \\ 0.169983 \\ 0.19998 \end{pmatrix}$$

The input gradient should be a row vector, so the transpose of an obtained vector should be taken:

$$\frac{dx}{dloss} = \begin{pmatrix} 0.139986 \\ 0.169983 \\ 0.19998 \end{pmatrix}^T = (0.139986 \quad 0.169983 \quad 0.19998).$$

Now the process may be repeated on all the preceding layers using an obtained input gradient.

2.3.2.3 Optimization

When neural network learns, it is changing its internal parameters based on the obtained gradient. The gradient descent algorithm will be used to optimize this demonstrational network. Gradient descent is the simplest optimization method for neural networks. It simply multiplies an obtained gradient by a negative of predefined learning rate value, and then adds it up to the network's attribute itself. Learning rate of 2 will be used here. Normally, learning rate's value is in range 0-1. This demonstration seeks to get an instant result, so it will use an enormously huge learning rate. Adding a product of gradient and a negative learning rate to the original values, the results are:

$$w_{new} = w_{old} + \frac{dw}{dloss} * (-lr) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} + \begin{pmatrix} 0.0279972 & -0.0669933 & 0.039996 \\ 0.0559944 & -0.1339866 & 0.07992 \\ 0.0839916 & -0.2009799 & 0.119988 \end{pmatrix} * (-2)$$
$$= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} + \begin{pmatrix} -0.0559944 & 0.1339866 & -0.079920 \\ -0.1119888 & 0.2679732 & -0.159984 \\ -0.1679832 & 0.4019598 & -0.239976 \end{pmatrix} \approx \begin{pmatrix} 0.94 & 2.13 & 2.92 \\ 3.89 & 5.27 & 5.84 \\ 6.83 & 8.40 & 8.76 \end{pmatrix}$$

$$b_{new} = b_{old} + \frac{db}{dloss} * (-lr) = (0.4 \quad 0.2 \quad 0.3) + (0.279972 \quad -0.669933 \quad 0.39996) * (-2) =$$
$$(0.4 \quad 0.2 \quad 0.3) + (-0.559944 \quad 1.339866 \quad -0.7992) \approx (-0.16 \quad 1.54 \quad -0.50).$$

Now, a test forward pass on the optimized network could be made to see the results: $y = x * w^T + b = (0.1 \quad 0.2 \quad 0.3) \cdot \begin{pmatrix} 0.94 & 3.89 & 6.83 \\ 2.13 & 5.27 & 8.40 \\ 2.92 & 5.84 & 8.76 \end{pmatrix} + (-0.16 \quad 1.54 \quad -0.50) = (1.40 \quad 3.20 \quad 5.00) +$ $(-0.16 \quad 1.54 \quad -0.50) = (1.24 \quad 4.74 \quad 4.5)$. Applying activation function, the output is: $output = \left(\frac{e^{1.24}}{207.91} \quad \frac{e^{4.74}}{207.91} \quad \frac{e^{4.5}}{207.91}\right) = (0.016 \quad 0.550 \quad 0.433)$. After optimization, the network is predicting class #2 with 55% certainty. Comparing it to the 33% in an unoptimized network, the result is clearly closer to the desired one.

2.4 Code implementation

This chapter contains the code solution for an artificial neural network python library. It will be implemented using NumPy library. This library contains the needed tools for working with matrices:

dot product, transpose, etc. The code will be made to work with data in batches. When the data is batched, optimization steps are smoothed out. This lets the network to not to take extreme changes every training input, but rather slowly progress towards the ideal operation. The input data is now in such a matrix form, that there is a single data entry per row:

$input = \begin{pmatrix} in1val1 & in1val2 & \cdots & in1valn \\ \vdots & & \ddots & \vdots \\ inmval1 & inmval2 & \cdots & inmvaln \end{pmatrix}$. Minibatch Stochastic Gradient Descent will be used for optimizing the neural network.

2.4.1 Library

2.4.1.1 layer.py

This file will contain the logic of a layer. It will also store the momentums for Minibatch Stochastic Gradient Descent.

```
import numpy as np

class Dense:

    # Layer initialization
    def __init__(self, inputs, outputs, actfun):
        #1 column - weight set per neuron e.g. transposed from the initialization
        #We need to have small weights at initialization. This will ease the training
process, since the changes made during initial training will make a considerable effect
on output
        self.weights = 0.01 * np.random.randn(inputs, outputs)
        #row - 1, since there's 1 bias per neuron, column - amount of neurons
        self.biases = np.zeros((1, outputs))
        #Activation function pointer
        self.actfun = actfun()
        #Initialize the weight momentum for the optimization
        self.weight_momentum = np.zeros_like(self.weights)
        #Initialize the bias momentum for the optimization
        self.bias_momentum = np.zeros_like(self.biases)

    # Forward pass
    def forward(self, inputs):
        #Memorize input values for the backpropagation
        self.inputs = inputs
        #Multipy inputs by corresponding weights, add bias. Memorize output for the
backpropagation
        self.output = np.dot(inputs, self.weights) + self.biases
        #Send the neurons' output to the activation function
        self.actfun.forward(self.output)

    # Backward pass
    def backward(self, actfun_next_grad):
        #Backpropagate through the activation function first
```

```
        self.actfun.backward(actfun_next_grad)

        #The derivative of a neuron: f(wei,inp,bias) = inp * wei + bias
        #di/df(w,i,b) = w. inp is also a function, so f' = w * f'(i,w,b)
        #This means that di for each neuron is going to be a matrix of weights:
[[w1,w2,w3]]

        #self.actfun.input_gradient - gradient obtained from the layer's activation
function
        #It is a matrix of derivatives as per neuron output
        #Each row is a vector of derivatives for each training set in a batch

        #Set the weight and bias gradients for the neurons
        #We need to transpose weights so that we are matching the shape of inputs(they
are not transposed from the initialization)
        self.weight_gradient = np.dot(self.inputs.T, self.actfun.input_gradient)

        #Derivative of a bias calculation is always 1. By the chain rule, we just need
to get the overall gradient that we got from the next layer and multiply it by 1.
        #We also need to add another dimension since we have lost one in the np.sum()
function
        self.bias_gradient = np.array([np.sum(self.actfun.input_gradient, axis=0)])

        #Set the input gradient for further backpropagation
        #The operation is a bit different from the mathematical representation due to
batching
        #We need to transpose weights so that we are matching the shape of inputs(they
are not transposed from the initialization)
        self.input_gradient = np.dot(self.actfun.input_gradient, self.weights.T)
```

2.4.1.2 actfun.py

This file contains 2 activation functions: ReLu and Softmax. They iteratively transform outputs from neurons.

```
import numpy as np

#max(0, x)
class ReLu:
    def forward(self, inputs):
        #Remember input values
        self.inputs = inputs
        #Replace all the negative values in inputs with 0
        self.output = np.maximum(0, inputs)

    def backward(self, next_grad):
        #dReLU/dx = 1 if x > 0, 0 if x <= 0
        #Make a copy of next_grad to not to overwrite original values
        self.input_gradient = next_grad.copy()

        #Change each value at index i to 0 where actfun's inputs[i] was < 0
        self.input_gradient[self.inputs <= 0] = 0
```

```
#e^i/sum(e^0-len)
class Softmax:
    def forward(self, inputs):

        #Memorize inputs for the backpropagation
        self.inputs = inputs
        #Get an exponintiated array
        exponents = np.exp(inputs)
        #Get the dimensions
        dims = np.shape(inputs)
        #Get a vector of sums(as per output set in a batch)
        expsum = np.sum(exponents, axis=1)
        #Reshape the output, so that each value is a vector
        expsum = expsum.reshape(dims[0], 1)

        #Divide exponentiated values by the sum of each exponentiated output set(dim 1)
        self.output = exponents / expsum

    def backward(self, next_grad):

        #Gradient placeholder
        self.input_gradient = np.zeros_like(next_grad)

        #For each grad set in a batch
        for index, (out, grad) in enumerate(zip(self.output, next_grad)):
            #Flatten the output
            out = out.reshape(-1, 1)

            #Calculate the Jacobian matrix
            #Jacobian matrix used to decrease the computations amount
            #Diagflat - matrix which is one-hot encoded by the values at indices(puts
all the values diagonally)
            jacobian = np.diagflat(out) - np.dot(out, out.T)

            #Append the gradient list with propagated values
            self.input_gradient[index] = np.dot(jacobian, grad)
```

2.4.1.3 lossfun.py

The file contains just one loss function: categorical cross entropy loss. It is able to receive target values in 2 forms: index-encoded and one-hot encoded. It is differentiated by looking at the dimensionality of an array: if it is a vector, then the values are index-encoded (due to batched form of operation). If the target array is a 2-dimensional matrix, then algorithm assumes that targets are one-hot encoded.

```
import numpy as np
```

```
class CategoricalCrossEntropy():

    def forward(self, result, target):

        #Check if target values are one-hot encoded.
        #If it is, convert it to index vector form
        if len(target.shape) == 2:
            #Get a 1-dimensional array with indexes of "hot"(1) values
            target_idx = np.argmax(target, axis=1)
        elif len(target.shape) == 1:
            target_idx = target

        #Array of resulting values as per target
        needed_vals =[]

        #Fill the vals array with results at "right" indexes
        for i in range(len(target_idx)):
            needed_vals.append(result[i, target_idx[i]])

        #Convert an array to numpy array
        needed_vals = np.array(needed_vals)
        #Replace all 0 and negative values in the array to prevent division by 0
        needed_vals = np.where(needed_vals <= 0, 1e-10, needed_vals)
        #Replace all 1(full match) and bigger values in the array to prevent overflowing
(log(x) < 0 when x > 1)
        needed_vals = np.where(needed_vals >= 1, 1 - 1e-10, needed_vals)

        #Apply natural logarithm to the values. Smaller value -> bigger abs(output)
        #Multiply the result by -1
        loss = -1 * np.log(needed_vals)

        #Get an average value of array
        avg_loss = np.mean(loss)
        #Return the average loss value
        return avg_loss

    def backward(self, result, target):
        #Input derivative of a cross entropy loss is -1 * (target/result), elementwise

        #Check if target values are index encoded
        #Convert them to one-hot encoded form
        if len(np.shape(target)) == 1:
            target_one_hot = []
            #Go through each set in a batch
            for sample in target:
                #Create empty array of the same size as result
                one_hot = np.zeros(len(result[0]))
                #Encode value at needed index to 1
                one_hot[sample] = 1
                #Append new set to the array
                target_one_hot.append(one_hot)
        else:
            target_one_hot = target
```

```
        #Get the input gradients
        self.input_gradient = -1 * (target_one_hot/result)
        #Normalize the gradient. This is needed since we are working with batches
        self.input_gradient = self.input_gradient / len(result)
```

2.4.1.4 model.py

This file introduces the model level of abstraction. It is used to chain and automate layer-layer processes. It also greatly eases the initialization of a neural network.

```
import numpy as np

class Model():
    def __init__(self, loss_function):
        #Initialize the array that will hold layer objects
        self.layers = []
        #Initialize the loss function
        self.lossfun = loss_function()

    #Add layer to the model
    def addLayer(self, layer):
        #Add layer to the holder array
        self.layers.append(layer)

    def forward(self, inputs):
        #Transform inputs to the numpy array
        inputs = np.array(inputs)
        #If the inputs are just 1 sample outside the batch, add 1 more dimension
        if inputs.shape == 1:
            inputs = np.reshape(inputs, (1, inputs.shape))
        #Go throug the whole network, starting at index 0
        for layer in self.layers:
            #Do a forward pass on the neuron (actfun(x*w + b))
            layer.forward(inputs)
            #Memorize activation function's output for next step/returning the predic-
tion
            inputs = layer.actfun.output
        #Return last layer's activation function's output as the network prediction
        return inputs

    #Calculate the loss value using a provided dataset
    def calculate_loss(self, inputs, targets):
        #Convert arrays to numpy arrays
        inputs = np.array(inputs)
        targets = np.array(targets)
        #Return the average loss value
        return(self.lossfun.forward(self.forward(inputs), targets))
```

```
    #This function is used to find gradient of a whole network
    def backward(self, inputs, targets):
        #Transform inputs to needed form(batched)
        inputs = np.array(inputs)
        if inputs.shape == 1:
            inputs = np.reshape(inputs, (1, inputs.shape))
        #Transform targets to needed form(batched)
        targets = np.array(targets)
        if targets.shape == 1:
            targets = np.reshape(targets, (1, targets.shape))

        #Do a forward pass so that inputs and outputs of each layer are memorized
        self.forward(inputs)

        #Calculate the loss gradient using output of the last layer's activation func-
tion and targets
        self.lossfun.backward(self.layers[len(self.layers) - 1].actfun.output, targets)

        #The main operation is to find gradients of weights and biases for each of the
neurons.

        #Backpropagate lossfun's unput gradient through the last layer
        self.layers[len(self.layers) - 1].backward(self.lossfun.input_gradient)

        #Go through each layer and backpropagate the gradient
        #Order: last -> first layer
        for lay in range(len(self.layers)-2, -1, -1):
            #Backpropagate through layer using next layer's input gradient
            self.layers[lay].backward(self.layers[lay+1].input_gradient)

    #This function is used to save the trained model(weights and biases) to a file
    def save_model(self, filename):
        with open(filename, 'wb') as f:
            for layer in self.layers:
                np.save(f, layer.weights)
                np.save(f, layer.biases)

    #This function is used to load the trained model attributes from a file
    def load_model(self, filename):
        with open(filename, 'rb') as f:
            for layer in self.layers:
                layer.weights = np.load(f)
                layer.biases = np.load(f)
```

2.4.1.5 optimizer.py

This file contains the SGD (Stochastic Gradient Descent) class, which is responsible for tweaking the network's attributes. It has 3 parameters: learning rate, decay and momentum. Decay is used to decrease the learning rate on every optimization step. It lets the network to tweak its parameters

using a really small learning rate in the end, so that the loss gets closer to the actual minimum. Momentum is used to follow the trend, e.g. "jump over" the local minimums. It does so by memorizing changes that were made in the previous optimizations.

```
import numpy as np

class SGD:
    def __init__(self, model, learning_rate, decay = 0.01, momentum = 0.5):
        #Set the model to be optimized
        self.model = model
        #Initialize starting learning rate
        self.learning_rate = learning_rate
        #Initialize actual learning rate
        self.lr = learning_rate
        #Set the decay
        self.decay = decay
        #Set the momentum
        self.momentum = momentum
        #Initialize "step" counter for updating the learning rate
        self.step = 0

    def forward(self):
        #Decay the current learning rate
        #When decay is 0, it doesen't affect the learning rate at all
        self.lr = self.learning_rate * (1. / (1. + self.decay * self.step))
        #Go through each layer in the model
        for lay in self.model.layers:
            #Calculate and set new momentums for the layer
            #When momentum is 0, it doesen't affect the learning rate at all
            lay.weight_momentum = self.momentum * lay.weight_momentum - self.lr *
lay.weight_gradient
            lay.bias_momentum = self.momentum * lay.bias_momentum - self.lr *
lay.bias_gradient
            #Update layer's parameters based on the calculated momentums
            lay.weights += lay.weight_momentum
            lay.biases += lay.bias_momentum
        #Update step counter
        self.step += 1
```

2.4.2 MNIST database

MNIST is one of the most common datasets for testing different architectures of neural networks. It contains 70000 grayscale images that depict handwritten Arabic numerals. An example except is shown in Figure 10. Every image has a special class assigned to it, representing the associated number 0-9. It consists of two separate unions: train and test. Train set contains 60000 evenly distributed images, and test set has 10000 images in it. (LeCun et al. 1998, 9–10). This is a common structure of a dataset. It lets programmers to test the network on data that was not used for training the network, thus identifying the problem of overfitting (neural network memorized an input-output pairs instead of learning features).

FIGURE 10. Example subset of 100 MNIST database pictures in grayscale representation (LeCun et al. 1998, 10)

2.4.3 Architecture

MNIST is a very simple dataset, so the neural network may be minimal, but still be able to have a good accuracy. Thesis will show a 4-layered neural network for a demonstration purpose. The input layer will receive an individual pixel from an image, so it is going to be 28 * 28 = 784 neurons. The first hidden layer will have 20 neurons. This will result in 784 * 20 = 15680 weighted connections. The second hidden layer will consist of 15 neurons, which results in 20 * 15 = 300 weighted connections. The output layer will have 10 neurons, where each neuron is going to be responsible for representing a single number (0-9). This results in 15 * 10 = 150 weighted connections. This architecture has a total of 784 + 20 + 15 + 10 = 829 neurons, thus, 829 biases and 15680 + 300 + 150 = 16130 weights. The architecture will use ReLu activation function on the hidden layers. Since MNIST is a classification problem, the model will have a SoftMax activation function on the output layer. This will provide a probability distribution of an image's class as an output. A Categorical Cross Entropy loss will be used as this is a categorical problem. Minibatch Stochastic Gradient Descent with capability of momentum and decay calculations will be used to optimize the attributes.

2.4.4 Code

2.4.4.1 main.py

Main file will have a logic that will be operating on the parts that are taken from a custom library. The input layer is emulated, so there is no initialization for it. This code is made just for demonstration purposes.

```
import layer
import model
import lossfun
import optimizer
import actfun
import os

import numpy as np



def mnist_train(model_to_use=0, learning_rate=0.5):
    #If model is not provided
    if model_to_use == 0:
        #Initialize the model with cross entropy loss
        model_to_use = model.Model(lossfun.CategoricalCrossEntropy)

        #Create layers
        dense1 = layer.Dense(784, 15, actfun.ReLu)
        dense2 = layer.Dense(15, 20, actfun.ReLu)
        dense3 = layer.Dense(20, 10, actfun.Softmax)

        #Add layers to the model
        model_to_use.addLayer(dense1)
        model_to_use.addLayer(dense2)
        model_to_use.addLayer(dense3)

    # Create optimizer
    opt = optimizer.SGD(model_to_use, learning_rate=learning_rate, decay=0.01, momen-
tum=0.5)

    #Open the mnist train csv file
    mnist_file = open(os.sys.path[0] + "\\mnist_train.csv")
    #Read each line as row into the mnist_data array
    mnist_data = mnist_file.readlines()
    #Close the mnist file
    mnist_file.close()

    #Split each line(row(string) in mnist data) by comma
    for img in mnist_data:
        img = np.array(img.split(","))

    #Array of class(target) values
    classes = []
```

```
    #Go through each image in mnist_data
    for image in range(len(mnist_data)):
        #Append new class value(see the mnist dataset structure)
        classes.append(int(mnist_data[image][0]))
        #Append new image(indexes 1:785)
        #We want to have neural network's inputs in the range of 0.00001 - 0.99999
        mnist_data[image] = ((np.asfarray(mnist_data[image].split(",")[1:]) / 255.0 *
0.99999) + 0.00001)

    #Set the batch size
    batch_size = 50
    #Set the desired amount of epochs
    epochs_amount = 10

    #Find amount of batches
    batch_amount = (len(mnist_data)/batch_size)

    # Train in loop
    for epoch in range(epochs_amount):

        #Variable for calculating accuracy as per whole epoch
        accuracy_epoch = 0
        #Variable for calculating loss as per whole epoch
        loss_epoch = 0

        #Go through all images using step of batch size
        for image_number in range(0,(len(mnist_data)),batch_size):
            #Get the arrays of images and classes
            batch = mnist_data[image_number:image_number+batch_size]
            class_batch = classes[image_number:image_number+batch_size]

            #Check for last entry overflow
            if len(mnist_data) <= image_number+batch_size:
                batch = mnist_data[image_number:len(mnist_data)-1]
                class_batch = classes[image_number:len(classes)-1]

            #Get the network's predictions
            result = model_to_use.forward(batch)
            #Get the most certain prediction
            predictions = np.argmax(result, axis=1)
            #Convert classes to numpy array
            target = np.array(class_batch)
            #Find the accuracy
            accuracy = np.mean(predictions==target) * 100
            #Add batch accuracy to epoch accuracy
            accuracy_epoch += accuracy


            #Uncomment to see the amount of correct guesses
            #strike = 0
            #for a,b in zip(predictions, target):
            #    if a == b:
```

```
            #        strike += 1
            #print("Amount of correct answers in batch: ", strike, "/", bsize)

            #Find the gradient
            model_to_use.backward(batch, target)

            #Add loss to the whole loss
            loss_epoch += model_to_use.calculate_loss(batch, target)


            #Optimize network's parameters
            opt.forward()
        #Print the information about an epoch pass
        print("Epoch: {}\tAccuracy in the last batch: {:.3f} %\tAverage accuracy in
epoch: {:.3f} %\tLast batch loss: {:.5f}\tAverage loss in epoch: {:.5f}".format\
            (epoch, accuracy, accuracy_epoch/batch_amount, model_to_use.calcu-
late_loss(batch, target),loss_epoch/batch_amount))
        #Accuracy may be 100% while loss is not 0.
        #This is due to the network being "uncertain"

    #Return the trained model
    return model_to_use

def test_model(model):
    #Open the mnist test csv file
    mnist_file = open(os.sys.path[0] + "\\mnist_test.csv")
    #Read each line as row into the mnist_data array
    mnist_data = mnist_file.readlines()
    #Close the mnist file
    mnist_file.close()

    #Split each line(row(string) in mnist data) by comma
    for img in mnist_data:
        img = np.array(img.split(","))

    #Array of class(target) values
    classes = []

    #Go through each image in mnist_data
    for image in range(len(mnist_data)):
        #Append new class value(see the mnist dataset structure)
        classes.append(int(mnist_data[image][0]))
        #We want to have neural network's inputs in the range of 0.00001 - 0.99999
        mnist_data[image] = ((np.asfarray(mnist_data[image].split(",")[1:]) / 255.0 *
0.99999) + 0.00001)

    #Make a forward pass using all the images
    test_result = model.forward(mnist_data)
    #Convert the results to index form
    test_result = np.argmax(test_result, axis=1)
    #Find accuracy
    accuracy = np.mean(test_result == np.array(classes)) * 100
```

```
    #Print the testing results
    print("Accuracy in the test batch: {:.6f} %\tLoss: {:.6f}".format(accuracy,
model.calculate_loss(mnist_data, classes)))

    #Accuracy may be 100% while loss is not 0.
    #This is due to the network being "uncertain"


def load_model(filename='params.npy'):
    #Loading the model may be encapsulated to another layer of complexity(architecture),
but i think it is enough for the demonstration purposes

    #Initialize the model
    loaded_model = model.Model(lossfun.CategoricalCrossEntropy)

    #Create layers
    dense1 = layer.Dense(784, 15, actfun.ReLu)
    dense2 = layer.Dense(15, 20, actfun.ReLu)
    dense3 = layer.Dense(20, 10, actfun.Softmax)

    #Add layers to the model
    loaded_model.addLayer(dense1)
    loaded_model.addLayer(dense2)
    loaded_model.addLayer(dense3)

    #Load parameters to the model
    loaded_model.load_model(filename)

    #Return the loaded model
    return loaded_model

def main():
    #Create and train a model
    model = mnist_train()
    #Do a test on a model using mnist_test dataset
    test_model(model)
    #Train a model 2nd time
    model = mnist_train(model, learning_rate=0.01)
    #Do a test on a model using mnist_test dataset
    print("Testing the final model:")
    test_model(model)
    #Save the model to a file
    model.save_model('mnist_model_params.npy')
    #Load the model from a file
    loaded_model = load_model('mnist_model_params.npy')
    #Do a test on a loaded model using mnist_test dataset
    print("Testing the loaded model:")
    test_model(loaded_model)


if __name__ == "__main__":
    main()
```

2.4.4.2 Function list

2.4.4.2.1 mnist_train()

This function is used to train the network using MNIST train set. It could either create the described model from the ground up or continue training the existing model object. It makes predefined number of passes through the whole dataset. This pass is called an epoch. Normally, the more epochs a network has learned, the better the resulting accuracy. Also, it uses batched inputs. That means that it is taking multiple entries as a single input in order to smooth out the learning process.

At the end of operation, the function returns a trained model.

2.4.4.2.2 test_model()

This function is used to test the network using MNIST test set. It takes a model as an argument, forwards the testing dataset and then calculates and prints the resulting accuracy and loss values.

2.4.4.2.3 load_model()

This function is used to create a model object, and then load all the parameters stored in a NumPy binary file. The model is then returned.

2.4.4.2.4 main()

The main function is the one that gets executed first. In it, the model is obtained by calling an mnist_train() function. After that, it shows the results using test_model(). Then, the model is further trained in mnist_train() function. The final result is shown, and then the model is saved. After that, the saved model is loaded and tested to demonstrate saving capabilities.

2.4.4.3 Loading a dataset

The MNIST dataset in .csv format could be taken from https://pjreddie.com/projects/mnist-in-csv/. The format is such that each row in a table is a single data entry, e.g. picture. The first value in a row is a label, and the next 784 values are pixels' values. The pixel values are in such a form, that a batch of first 28 entries is 1st pixel row, next batch is 2nd pixel row, and so on.

2.4.4.4 Results

After 200 attempts using this architecture and algorithm (1 pass @ learning rate = 0.5, momentum = 0.5, decay = 0.01, batch size = 50, epochs = 10; 2 pass @ learning rate = 0.01, momentum = 0.5, decay = 0.01, batch size = 50, epochs = 10), the best accuracy obtained was 92.929044930027%.

Experimentally the biggest accuracy obtained was 98.380199 at epoch#102. All the properties, except for epochs amount and a base architecture, were the same. Epochs amount was increased to let the neural network learn for longer. The architecture was:

```
dense1 = layer.Dense(784, 40, actfun.ReLu)
dense2 = layer.Dense(40, 20, actfun.ReLu)
dense3 = layer.Dense(20, 10, actfun.Softmax)
```

3 CONVOLUTIONAL NEURAL NETWORKS

3.1 Overview

Artificial neural networks have a problem when working with huge and diverse images. For example, if an image is RGB, and its size is 32*32 pixels, the model needs to have 3072 weights at the first layer. If RGB image has a size of 416*416 pixels, the initial weight amount is equal to 519168. This leads to overfitting and a large amount of computational power consumption. (Gogul and Kumar 2017, 3). Also, artificial neural networks are not able to find patterns in different parts of an image. In image classification problems, they are simply applying masks to find which class is the most probable one.

Convolutional neural networks are made to solve these problems. They are sequentially applying special mathematical filters that are extracting different patterns of either image itself, or the feature map received from the previous filter. The network learns to give correct outputs by optimizing internal parameters of filters. CNNs are spatially invariant, meaning that the feature may be found on any part of an image.

The operation of machine vision convolutional neural network is very similar to the vision of a human. Humans are also able to extract and compile many different features from any part of received visual information. (Carandini 2006, 1(463)-3(465)).

3.2 Components

3.2.1 Network

Normally, convolutional neural networks are made up of 2 parts: convolutional part, e.g. feature extraction, at the beginning and fully connected part, e.g. classification, at the end. (Kang, Song and Sun 2019, 3-4).

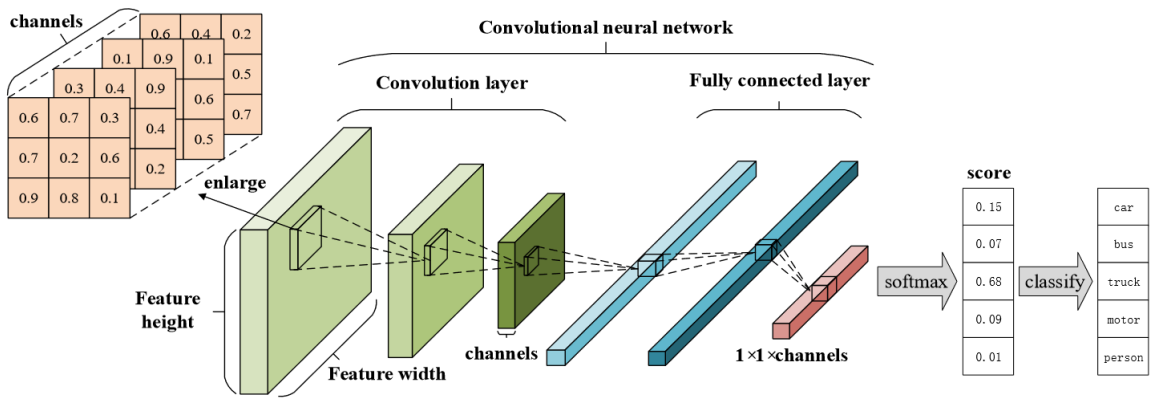

FIGURE 11. Architecture of a Convolutional Neural Network (Kang, Song and Sun 2019, 4)

Convolutional part normally consists of convolutional and pooling layers. Each convolutional layer consists of 1 or more filters. Layers are connected sequentially, so that the output (feature set) of the first layer is the input to a second.

Fully connected part may be represented as a simple artificial neural network which has a flattened convolutional layer's output as input. Its main goal is to compile and mask many different features to a required output.

3.2.1.1 Common sliding window operation

Both convolution and pooling operations are sliding window algorithms. It means that there is a special window, e.g. kernel, that slides through the data and applies special logic to a scope. The kernel has two properties: width and height. Commonly, the kernel's size is represented as a single number, representing both width and height. The algorithm itself also has two properties: stride and padding. Sometimes, it could have up to four properties due to different horizontal and vertical strides and paddings. Stride means the amount of data entries a window will move in a single step. Padding means the amount of additional data entries that are added to the "sides" of an original data. Commonly, padding is adding specified number of zeroes. This is called a zero padding. The sliding window operation flow is as follows:

1. The padding is applied to the data
2. The window is put to the first position, e.g. top-left corner, of an image or feature set
3. The logic is applied
4. The window is horizontally slided further, e.g. right, by the stride amount of data entries. If window gets out of bounds, it moves vertically, e.g. down, by the specified stride amount of data entries
5. The process is finished when the last scope was processed (the window finished horizontal stride, but there are no more entries in vertical domain)

If the algorithm returns a single value from a scope, then a special formula could be used to find output's spatial size (dimensions) of a sliding window operation:

$height = \frac{input\ height - kernel\ height + 2 * padding}{vertical\ stride} + 1;\ width = \frac{input\ width - kernel\ width + 2 * padding}{horizontal\ stride} + 1$ (Dumoulin and Visin 2018, 15). Later these two formulae are going to be referred to as "spatial extent formula".

3.2.2 Convolutional layer

A convolutional layer may be represented as a set of filters that are applied to the input. Each filter in a layer will add 1 channel to the output. All filters are applied to the input, and the result is forwarded further.

3.2.2.1 Filter

A filter may be represented as a 2-dimensional matrix that has a multiplier in each cell. There are more dimensional filters present. They have similar logic but are out of the scope of this thesis. Each filter is seeking for one particular feature in an input.

For example, here is a 3x3 filter kernel that seeks for a horizontal line: $\begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$.

The dimensions of a kernel are predefined, and normally have a size of 3x3 up to 5x5.

3.2.3 Pooling layer

A pooling layer is used to decrease the spatial dimensionality of processed data. It applies a special kernel that is extracting important information from multiple "pixels" and puts the compressed result to only 1 cell. As a result, it outputs compressed version of input. This process is also called "downsampling".

3.2.3.1 MaxPool

MaxPool is one of the most common pooling methods. It simply takes the biggest number in a scope and sets it as a region output.

3.2.3.2 AvgPool

As the name suggests, AvgPool is taking an average of a scope. Then, the value is put to a corresponding output's cell.

3.3 Representation of operation

3.3.1 Abstract

3.3.1.1 Convolution

In this part, there is going to be shown an abstract representation of a convolution operation. Such representation is human-friendly due to visual abstraction.

Example filter that is seeking for a descending diagonal line:

1	-0.5
-0.5	1

Example input (each value could be seen as pixel's grayscale intensity, so that if value is 1, the pixel is black, and if the value is 0, the pixel is white):

0.1	0.9	0.3	0.4
0.15	0.25	0.9	0.45
0.25	0.35	0.9	0.55
0.3	0.4	0.5	0.9

This demo layer has only 1 filter. Attributes of a layer: vertical stride = 2, horizontal stride = 1, padding = 0.

The filter is slided from top-left corner of an input down to bottom-right corner. It first slides horizontally, taking a step of horizontal stride. When it runs to an edge, it is moved down by vertical stride amount, and starts the process again at the left edge.

The first scope is going to be:

0.1	0.9
0.15	0.25

The second scope is going to be:

0.9	0.3
0.25	0.9

The fourth scope is going to be:

0.25	0.35
0.3	0.4

Applying stride and filter size, 6 scopes are obtained (purple and light green are both the filter and a part of a filter):

0.1	0.9	0.3	0.4
0.15	0.25	0.9	0.45
0.25	0.35	0.9	0.55
0.3	0.4	0.5	0.9

Applying a filter to a scope is done by multiplying corresponding cell values, and then summing up all the resulting products.

For example, an output of a first scope is going to be: $1 * 0.1 - 0.5 * 0.9 - 0.5 * 0.15 + 1 * 0.25 = -0.175$.

This is going to be the first value (top left) in an output. Processing every scope, the result is:

-0.175	1.525	0.1
0.325	0.2	1.275

This is an output of a convolution. Now, an activation function is applied. Applying ReLu:

0	1.525	0.1
0.325	0.2	1.275

3.3.1.2 Pooling

Normally, after the convolutional layer there is a pooling layer that is used to save computational power. Applying 1x2 MaxPool with a stride of 1 and padding of 0 to the output:

0	1.525	0.1
0.325	0.2	1.275

$\max(0, 0.325) = 0.325; \max(1.525, 0.2) = 1.525; \max(0.1, 1.275) = 1.275$

Result:

0.325	1.525	1.275

3.3.2 Mathematical

If the input is an RGB image, a model receives 3-dimensional data. In this example, there will be used 1-channeled input so that the channel dimension is removed for the sake of simplicity. $input = \begin{pmatrix} 0.30 & 0.90 & 0.30 & 0.90 \\ 0.15 & 0.00 & 0.90 & 0.45 \\ 0.55 & 0.90 & 0.00 & 0.45 \\ 0.90 & 0.15 & 0.50 & 0.90 \end{pmatrix}$; $filter = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}$. The vertical stride is 2 and horizontal stride is 2.

Padding = 1.

The first step is to apply padding. This is done by placing a number of 0 values at edges of a matrix. $input = \begin{pmatrix} 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.30 & 0.90 & 0.30 & 0.90 & 0.00 \\ 0.00 & 0.15 & 0.00 & 0.90 & 0.45 & 0.00 \\ 0.00 & 0.55 & 0.90 & 0.00 & 0.45 & 0.00 \\ 0.00 & 0.90 & 0.15 & 0.50 & 0.90 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$. The goal is to make a dot product on kernel and every possible scope. For that, it is needed to get a matrix, where each column is going to be a single scope. $input_{scoped} = \begin{pmatrix} 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.45 & 0.00 & 0.15 & 0.90 \\ 0.00 & 0.00 & 0.00 & 0.15 & 0.90 & 0.00 & 0.90 & 0.50 & 0.00 \\ 0.00 & 0.90 & 0.90 & 0.00 & 0.90 & 0.45 & 0.00 & 0.00 & 0.00 \\ 0.30 & 0.30 & 0.00 & 0.55 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$. Now the dot product between the flattened filter kernel and a scoped input may be taken. $filter_{flattened} = (-1 \quad 1 \quad 1 \quad -1)$.

$output = (-1 \quad 1 \quad 1 \quad -1) \cdot \begin{pmatrix} 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.45 & 0.00 & 0.15 & 0.90 \\ 0.00 & 0.00 & 0.00 & 0.15 & 0.90 & 0.00 & 0.90 & 0.50 & 0.00 \\ 0.00 & 0.90 & 0.90 & 0.00 & 0.90 & 0.45 & 0.00 & 0.00 & 0.00 \\ 0.30 & 0.30 & 0.00 & 0.55 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix} =$

$(-0.30 \quad 0.6 \quad 0.9 \quad -0.40 \quad 1.80 \quad 0.00 \quad 0.9 \quad 0.35 \quad -0.9)$.

The output of this convolutional operation, according to the spatial extent formula, is going to be:

$h = \frac{4-2+2*1}{2} + 1 = 3; w = \frac{4-2+2*1}{2} + 1 = 3$.

Now, the output could be reshaped to known needed dimensions of 3x3:

$output = \begin{pmatrix} -0.30 & 0.60 & 0.90 \\ -0.40 & 1.80 & 0.00 \\ 0.90 & 0.35 & -0.90 \end{pmatrix}$. Applying a ReLu activation function to an output:

$output_{actfun} = \begin{pmatrix} 0.00 & 0.60 & 0.90 \\ 0.00 & 1.80 & 0.00 \\ 0.90 & 0.35 & 0.00 \end{pmatrix}$.

The next step is going to be a pooling layer. Properties: AvgPool, padding=0, stride=1, kernel size = 2x2. There are a couple of ways to achieve this, but the thesis will show a convolutional approach. The goal is to make a dot product on every possible scope using a reciprocal of the kernel size. The first step is to create a row matrix, consisting of n values that are equal to kernel size. It is going to be: $kernel = (0.25 \quad 0.25 \quad 0.25 \quad 0.25)$. Now, it is needed to get a matrix, where each column is a scope: $input_{scoped} = \begin{pmatrix} 0.00 & 0.60 & 0.00 & 1.80 \\ 0.60 & 0.90 & 1.80 & 0.00 \\ 0.00 & 1.80 & 0.90 & 0.35 \\ 1.80 & 0.00 & 0.35 & 0.00 \end{pmatrix}$. Applying kernel: $output_{scoped} =$

$(0.25 \quad 0.25 \quad 0.25 \quad 0.25) \cdot \begin{pmatrix} 0.00 & 0.60 & 0.00 & 1.80 \\ 0.60 & 0.90 & 1.80 & 0.00 \\ 0.00 & 1.80 & 0.90 & 0.35 \\ 1.80 & 0.00 & 0.35 & 0.00 \end{pmatrix} = (0.60 \quad 0.825 \quad 0.7625 \quad 0.5375)$. Shaping back to complex form: $output = \begin{pmatrix} 0.60 & 0.825 \\ 0.7625 & 0.5375 \end{pmatrix}$. Now, the output could be sent to either next convolutional/pooling layer or to the flattening and dense layers.

Flattening is done by placing all the values to a row vector. The operation is such that each channel's rows are subsequently placed one after another, and then the resulting vectors are also successively concatenated together to form a final vector. Example of 2-channelled input:

$$flatten\left(\left(\begin{pmatrix}0.00 & 0.60 & 0.90\\ 0.00 & 1.80 & 0.00\\ 0.90 & 0.35 & 0.00\end{pmatrix}\ \begin{pmatrix}-0.30 & 0.60 & 0.90\\ -0.40 & 1.80 & 0.00\\ 0.90 & 0.35 & -0.90\end{pmatrix}\right)\right) = flatten\left(\begin{pmatrix}0.00 & 0.60 & 0.90\\ 0.00 & 1.80 & 0.00\\ 0.90 & 0.35 & 0.00\end{pmatrix}\right) +$$

$$fletten\left(\begin{pmatrix}-0.30 & 0.60 & 0.90\\ -0.40 & 1.80 & 0.00\\ 0.90 & 0.35 & -0.90\end{pmatrix}\right) = (0.00\quad 0.60\quad 0.90\quad 0.00\quad 1.80\quad 0.00\quad 0.90\quad 0.35\quad 0.00) +$$

$$(-0.3\quad 0.60\quad 0.90\quad -0.4\quad 1.80\quad 0.00\quad 0.90\quad 0.35\quad -0.90) =$$

$$(0.00\quad 0.60\quad 0.90\quad 0.00\quad 1.80\quad 0.00\quad 0.90\quad 0.35\quad 0.00 -0.30\quad 0.60\quad 0.90\quad \cdots\quad -0.9)\ .$$

It is now possible to send this vector to a regular dense layer.

The optimization is done by automatically tweaking the convolutional filters' multipliers. The core idea stays the same: find a derivative of every single multiplier with respect to the loss value and tweak them in needed direction.

3.4 Practical example

TensorFlow library will be used for implementing a practical example. It is much faster than self-made tools since it could use GPU powers for computations. TensorFlow is made with simplicity in mind, so the code is going to be easy to read, write and understand. (Abadi et al. 2016, 1).

3.4.1 Image classification

Convolutional neural networks are the most used tool in image classification problems. Example classification problems that CNN could be used to solve: cat or dog, damaged or not damaged car, fox's species, etc.

3.4.1.1 CIFAR 10

CIFAR 10 is an image classification dataset. It consists of 60000 32x32 pixel colored images. All the images are break into 10 classes. The classes are: airplane, automobile (but not truck or pickup truck), bird, cat, deer, dog, frog, horse, ship, and truck (but not pickup truck). The dataset is split to two subsets: 50000 images in training set, and 10000 in testing set. (Krizhevsky 2009, 32).

FIGURE 12. An example picture of a deer in CIFAR-10 dataset

The best accuracy achieved at the time of writing this thesis was 99.4% (Touvron et al. 2021, 16).

CIFAR 10 is one of the most referenced datasets. Figure 13 shows the graph of the most referenced datasets in scientific papers over time.

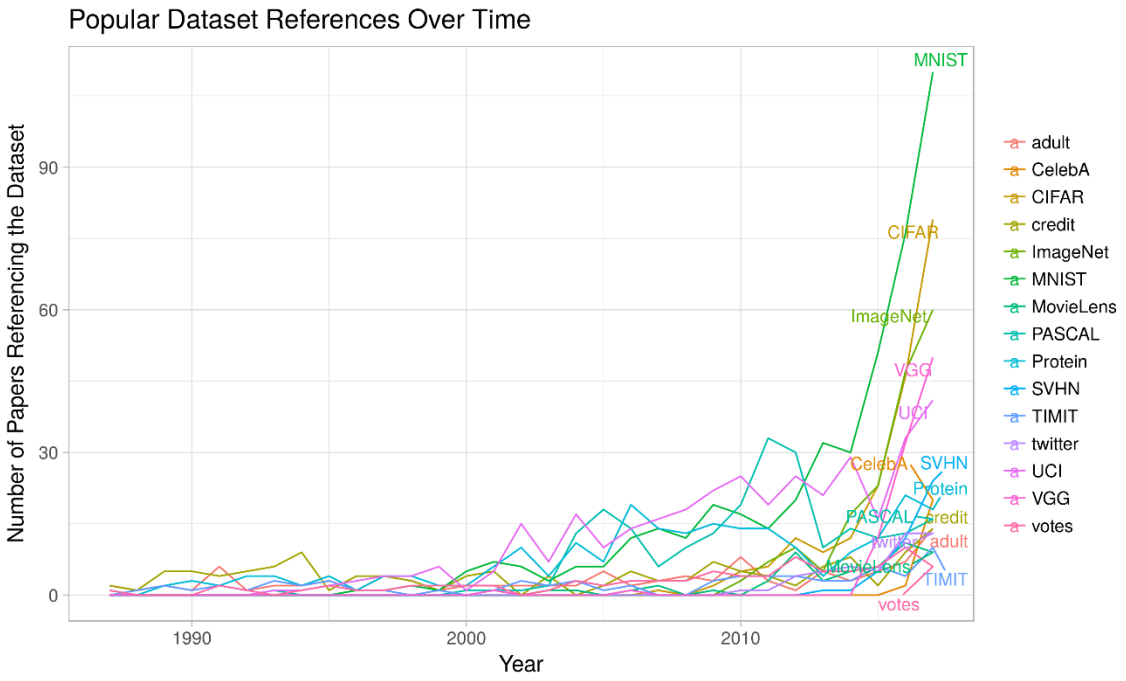

FIGURE 13. Popular Dataset References Over Time (Ben Hamner 2017)

3.4.1.2 Code

The code contains an example solution for the CIFAR-10 dataset. It is provided with the architecture that has achieved the best results throughout thesis research process.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' #Disable unnecessary logging

import tensorflow as tf
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, Activation
```

```
from tensorflow.keras import datasets, Model
import numpy as np
import matplotlib.pyplot as plt
import random
from datetime import datetime

#Print the amount of usable GPUs
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))

#List of indexed labels.
cifar_labels = ['airplane', 'auto', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck']
#Batch size for training process. This value was obtained experimentally
BATCH_SIZE = 8
#Batch size for testing process. This is needed for reducing memory consumption
TEST_BATCH_SIZE = 100
#Length of an image. Needed for scoping in test pass
IMG_LEN = 32 * 32

#Amount of epochs to be trained
EPOCHS = 30
#Variable that is used for timing of updating predictions
CHECK_EVERY = 100
#Length of subdataset. This is needed for reducing memory consumption
DEN = 1000
#Grid of pictures to show in the test step
IMG_ROWS = 3
IMG_COLS = 10

#Load the CIFAR10 dataset from the keras
(x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()
#Normalize the inputs to be on the scale 0.0-1.0
x_train, x_test = x_train / 255.0, x_test / 255.0

#Put test pictures in batches. Shuffling will not take any effect, so it may be skipped
test_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(TEST_BATCH_SIZE)

#Define the class for custom model
class MDL(Model):
    def __init__(self):
        #Call the parent class initialization
        super(MDL, self).__init__()

        #An array, where the convolutional part will be stored
        self.conv_part = []

        #Append needed layers one by one
        #SAME padding means that the padding will be toggled to make output's spatial
size the same as input's one.
        self.conv_part.append(Conv2D(64, 3,  padding="SAME"))
        self.conv_part.append(Activation("relu"))
        self.conv_part.append(MaxPooling2D((2,2)))
        self.conv_part.append(Conv2D(128, 3, padding="SAME"))
```

```
        self.conv_part.append(Activation("relu"))
        self.conv_part.append(MaxPooling2D((2,2)))
        self.conv_part.append(Conv2D(256, 3, padding="SAME"))
        self.conv_part.append(Activation("relu"))
        self.conv_part.append(MaxPooling2D((2,2)))
        self.conv_part.append(Conv2D(512, 3, padding="SAME"))
        self.conv_part.append(Activation("relu"))
        self.conv_part.append(MaxPooling2D((2,2)))

        #An array, where the fully connected part is stored
        self.fconn_part = []
        self.fconn_part.append(Flatten())   #Flatten the input to a batch of vectors
        self.fconn_part.append(Dense(4096, activation='relu'))
        self.fconn_part.append(Dense(1000, activation='relu'))
        self.fconn_part.append(Dense(10, activation='softmax')) #Final layer with clas-
sifying activation function

    #Make a forward pass
    def call(self, x):
        #Forward through convolutional part first
        for lay in self.conv_part:
            x = lay(x)
        #Forward through fully connected part second
        for lay in self.fconn_part:
            x = lay(x)
        #Return the model's predictions
        return x

# Create an instance of the model
model = MDL()


#Initialize the loss object
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
#Initialize a basic Stochastic gradient descent optimizer
optimizer = tf.keras.optimizers.SGD()


#The metrics that will show the loss during training process
train_loss = tf.keras.metrics.Mean(name='train_loss')
#The metrics that will show the batch accuracy during training process
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

#The metrics that will show the loss during testing process
test_loss = tf.keras.metrics.Mean(name='test_loss')
#The metrics that will show the batch accuracy during testing process
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')


#Train the model 1 time using inputs and labels
@tf.function
def train_step(images, labels):
    #Gradient tape is used for automatic derrivation of the whole model
```

```
    with tf.GradientTape() as tape:
        #Do a forward pass. Remember the inputs on each step
        predictions = model(images, training=True)
        #Calculate the resulting loss
        loss = loss_object(labels, predictions)
    #Automatically calculate gradients
    gradients = tape.gradient(loss, model.trainable_variables)
    #Optimize the model using resulting gradients
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    #Get the loss after applying optimization
    train_loss(loss)
    #Get the accuracy after applying optimization
    train_accuracy(labels, predictions)

#Test the model
@tf.function
def test_step(images, labels):
  #Do a forward pass. Memorizing input values is not needed
    predictions = model(images, training=False)
    #Calculate the resulting loss
    t_loss = loss_object(labels, predictions)
    #Get the loss
    test_loss(t_loss)
    #Get the accuracy
    test_accuracy(labels, predictions)
    #Return model's predictions
    return predictions



#Define memory arrays for plotting
accuracy_history_full = []
accuracy_history_epoch = []
xdata_full = []
xdata_epoch = []

#Create plotting canvases
fig, (ax1, ax2) = plt.subplots(1, 2)
fig_imgs, imgs = plt.subplots(IMG_ROWS, IMG_COLS)

#Set the size for image plots
fig_imgs.set_size_inches(20, 20)

#Add titles and labels
fig.suptitle('Horizontally stacked subplots')
ax1.set_title("Overall progress")
ax1.set_xlabel("Batches processed")
ax1.set_ylabel("Accuracy")


#Train the model $EPOCHS times
for epoch in range(EPOCHS):
```

```
    #Clear the epoch accuracy plot
    ax2.clear()
    ax2.set_title("Epoch progress")
    ax2.set_xlabel("Batches processed")
    ax2.set_ylabel("Accuracy")


    #Arrays of accuracy history in epoch
    accuracy_history_epoch = []
    #Indices for accuracy points
    xdata_epoch = []
    #Reset the batch counter
    batch_index = 0
    # Reset the metrics at the start of the next epoch
    train_loss.reset_states()
    train_accuracy.reset_states()
    test_loss.reset_states()
    test_accuracy.reset_states()


    #Let's divide the datasets into subdatasets, len/den each
    #This is needed for memory usage reduction
    for subdataset_index in range(int(len(x_train)/DEN)-1):
        #Get subdatasets, length = DEN
        x_train_sub = x_train[subdataset_index*DEN:(subdataset_index+1)*DEN]
        y_train_sub = y_train[subdataset_index*DEN:(subdataset_index+1)*DEN]

        #Shuffle the inputs and put them in batches of 32 images per batch
        #Merge inputs and labels
        train_ds = tf.data.Dataset.from_tensor_slices((x_train_sub, y_train_sub))
        #Shuffle pairs 10000 times
        train_ds = train_ds.shuffle(10000)
        #Make batches of BATCH_SIZE
        train_ds = train_ds.batch(BATCH_SIZE)

        #Go through every batch
        for images, labels in train_ds:
            #Make a train step
            train_step(images, labels)
            #Update batch counter
            batch_index += 1
            #Update every n batches
            if batch_index % CHECK_EVERY == 0:
                #Get loss and accuracy values from the last batch processed
                loss = train_loss.result().numpy()
                accuracy = train_accuracy.result().numpy() * 100
                #Print the info. Return the carriage, but don't go to the next line
                print("Epoch: ", epoch, "\tCurrent loss: {:.3f}".format(loss), "\tCur-
rent accuracy: {:.3f}%".format(accuracy), "\tImages processed: {:.0f}".format((batch_in-
dex-1) * BATCH_SIZE), end="\r")
                #Append new values to memory
                accuracy_history_epoch.append(accuracy)
                xdata_epoch.append(int(batch_index/CHECK_EVERY))
```

```
                accuracy_history_full.append(accuracy)
                #Append the global indexing array
                if len(xdata_full) == 0:
                    #Begin from 0
                    xdata_full.append(0)
                else:
                    #Append incremented last number
                    xdata_full.append(xdata_full[-1]+1)
                #Plot the global accuracy graph
                ax1.plot(xdata_full, accuracy_history_full, color='r')
                #Plot the in-batch accuracy graph
                ax2.plot(xdata_epoch, accuracy_history_epoch, color='b')
                #Make a small pause to let matplotlib to render
                plt.pause(1)
    #Training is done


    predicts = []   #Array of batched predictions from test dataset

    #Go through every test batch
    for test_images, test_labels in test_ds:
        #Append predictions from a batch
        predicts.append(test_step(test_images, test_labels))


    #Go through every row in canvas
    for idx_row in range(len(imgs)):
        #Go through every column in canvas
        for idx_col in range(len(imgs[idx_row])):
            #Pick a random batch
            batch_idx_pred = random.randint(0, len(predicts)-1)
            #Pick a random image in that batch
            img_id_pred = random.randint(0, len(predicts[batch_idx_pred])-1)
            #Plot a picture to a cell. The calculations are needed to find corresponding
image
            imgs[idx_row][idx_col].imshow(x_test[batch_idx_pred * TEST_BATCH_SIZE +
img_id_pred])
            #Disable axes for the image plot
            imgs[idx_row][idx_col].axis('off')
            #Find the most certain answer in corresponding prediction
            idx_of_max_prob_prediction = np.argmax(pre-
dicts[batch_idx_pred][img_id_pred])
            #Make a guess/target pair
            pair = cifar_labels[idx_of_max_prob_prediction] + "\n" + cifar_la-
bels[y_test[batch_idx_pred * TEST_BATCH_SIZE + img_id_pred][0]]
            #If the prediction is right, print the name of predicted and target classes
in green
            if y_test[batch_idx_pred * TEST_BATCH_SIZE + img_id_pred] ==
idx_of_max_prob_prediction:
                imgs[idx_row][idx_col].set_title(pair, color="green")
            #If the prediction is not right, print the name of predicted and target
classes in red
            else:
```

```
                imgs[idx_row][idx_col].set_title(pair, color="red")

    #Update epoch number in super title
    fig_imgs.suptitle('Test results at epoch ' + str(epoch))
    #Save the image subset plot
    fig_imgs.savefig('epoch' + str(epoch) + '.png')

    #Print the results of an epoch
    print(f'\nEpoch {epoch + 1}, Train Loss: {train_loss.result()}, Train Accuracy:
{train_accuracy.result() * 100}, Test Loss: {test_loss.result()}, Test Accuracy:
{test_accuracy.result() * 100}')
    #Get the original filename using current datetime
    dateTimeObj = datetime.now()
    checkpoint_name = "checkpoint_" + dateTimeObj.strftime("%d_%m_%Y__%H_%M_%S")
    #Save the model using generated name
    model.save('./checkpoints/' + checkpoint_name)
```

3.4.1.2.1 Code explanation

At the beginning the code loads the CIFAR-10 dataset in an appropriate format. Then the code creates a model from the predefined inherited class MDL, which is a child of keras' Model superclass. The model's architecture is predefined inside of an MDL class. Then, the code creates special tools for communicating with the model. After that, the model trains the neural network specified amount of times. During the training process, it is providing the statistical information on the terminal, and in visual form using Matplotlib. The model is saved after passing an epoch, so that it is possible to use the best pretrained model later on.

```
Epoch 32, Loss: 0.014271031133830547, Accuracy: 99.5204086303711, Test Loss: 1.8199031352996826, Test Accuracy: 75.77999877929688
Epoch:  32      Current loss: 0.018     Current accuracy: 99.398%       Images processed: 48792
Epoch 33, Loss: 0.017638172954320908, Accuracy: 99.4000015258789, Test Loss: 1.7533411979675293, Test Accuracy: 75.93000030517578
Epoch:  33      Current loss: 0.015     Current accuracy: 99.547%       Images processed: 48792
Epoch 34, Loss: 0.015259655192494392, Accuracy: 99.54898071289062, Test Loss: 1.753045916557312, Test Accuracy: 76.16000366210938
Epoch:  34      Current loss: 0.012     Current accuracy: 99.643%       Images processed: 48792
Epoch 35, Loss: 0.011692920699715614, Accuracy: 99.64285278320312, Test Loss: 1.8601768016815186, Test Accuracy: 75.7300033569336
Epoch:  35      Current loss: 0.012     Current accuracy: 99.633%       Images processed: 48792
Epoch 36, Loss: 0.011677151545882225, Accuracy: 99.63469696044922, Test Loss: 1.8042412996292114, Test Accuracy: 75.56999969482422
Epoch:  36      Current loss: 0.013     Current accuracy: 99.598%       Images processed: 48792
Epoch 37, Loss: 0.012685966677963734, Accuracy: 99.5999984741211, Test Loss: 1.8528920412063599, Test Accuracy: 76.1300048828125
Epoch:  37      Current loss: 0.014     Current accuracy: 99.520%       Images processed: 48792
Epoch 38, Loss: 0.014313935302197933, Accuracy: 99.52244567871094, Test Loss: 1.7792021036148071, Test Accuracy: 75.63999938964844
Epoch:  38      Current loss: 0.011     Current accuracy: 99.655%       Images processed: 32792
```

FIGURE 14. A depicture of statistical information that is outputted to the terminal.

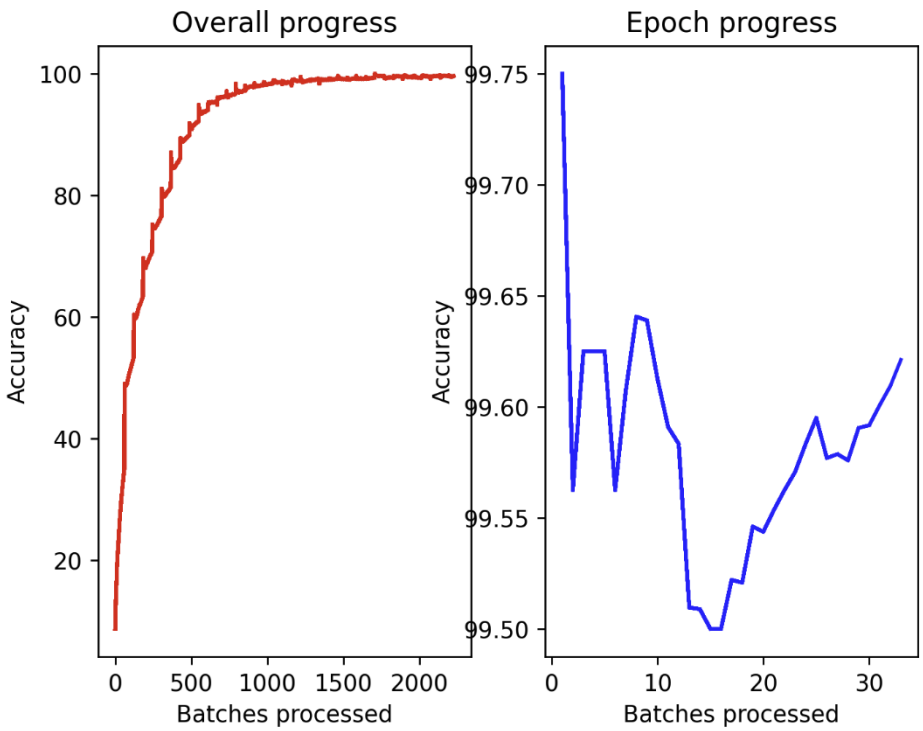

FIGURE 15. A graph of train set accuracy throughout training the model for more than 2000 batches

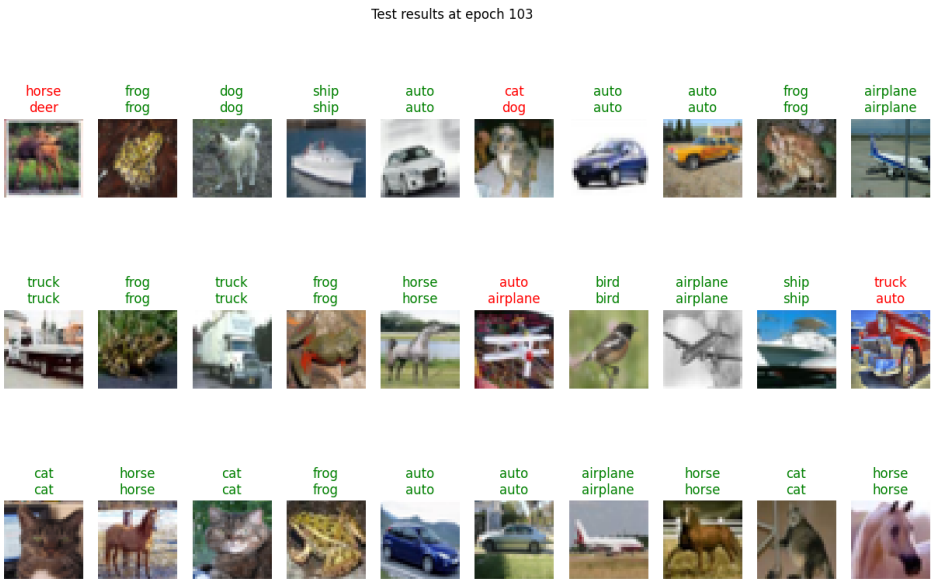

FIGURE 16. Depicture of example test subset results on the epoch #103

3.4.1.2.2 Results

After 20 different architectures tested, the best one was found. Each architecture was trained for 25 epochs 10 times to neglect local minimums. The research started from a sample architecture. Then, the best and average results were collected and analyzed. The decision was then made as of how to

tweak the architecture. The best architecture obtained had a peak test accuracy of 78.82% at 45th epoch.

The biggest problem was overfitting. In most runs, the train set accuracy was over 99%, while final test results were no more than 70%. There are a couple of ways to address this problem: batch normalization, dropout layers, etc. Due to overfitting, the model could be shrinked down to be less than a quarter as big and lose no more than 8% of best test accuracy.

Trained model could be tested using a custom picture. First, it is needed to obtain an image (see Figure 17). Then, crop a needed part of it (see Figure 18). After that, the image should be resized to the same spatial extent as it was trained on (see Figure 19). That is 32x32 pixels. Then a picture may be forwarded through the network:

```
#Load an image
img = tf.io.read_file('./custom_image_resized.png')

#Decode
img = tf.io.decode_png(img)

#Cast image's values as float32
img = tf.tensorflow.cast(img, tensorflow.float32)

#Normalize pixel values
img = img/255

#Add batch dimension
img = img[None, :, :, :]

#Forward the image through the network and print an output
print("Result: ", model_to_use(img, training=False))
```

FIGURE 17. Original picture of a car

FIGURE 18. Cropped picture of a car

FIGURE 19. Resized picture of a car

The resulting output is shown in Figure 20. The pretrained model is predicting 2nd class, which is a car, with 99.97% certainty.

```
Result:  tf.Tensor(
[[5.0123646e-07 9.9873298e-01 2.2520693e-05 9.8991685e-04 1.3914726e-08
  1.4106928e-07 3.5478998e-08 3.1037416e-05 2.3578672e-09 2.2281523e-04]], shape=(1, 10), dtype=float32)
PS C:\Users\ilkute\Desktop\170522\thesis>
```

FIGURE 20. Output of a pretrained network

3.4.2 Object detection

Object detection is a step further from image classification tasks. It is aimed at positions and classes of objects that are present in a picture. The object detection models require a lot of computational power. Due to this reason, the first object detection neural networks have started to be created in the 1990s. (Rowley, Baluja & Kanade 1998, 1(23)).

The ground-braking algorithm was named "Viola-Jones object detection framework". It is able to localize multiple faces on the input image with a detection rate of 95% with a false positive rate at 1 in 14084. In 1998, the execution took only 0.7 seconds to scan a 384 by 288-pixel image. (Viola,

Jones 2001, 4). The operation is such that a window, consisting of set of pretrained detectors, is slided through the whole image. Then, if the output's masking value was bigger than threshold, the detection was made.

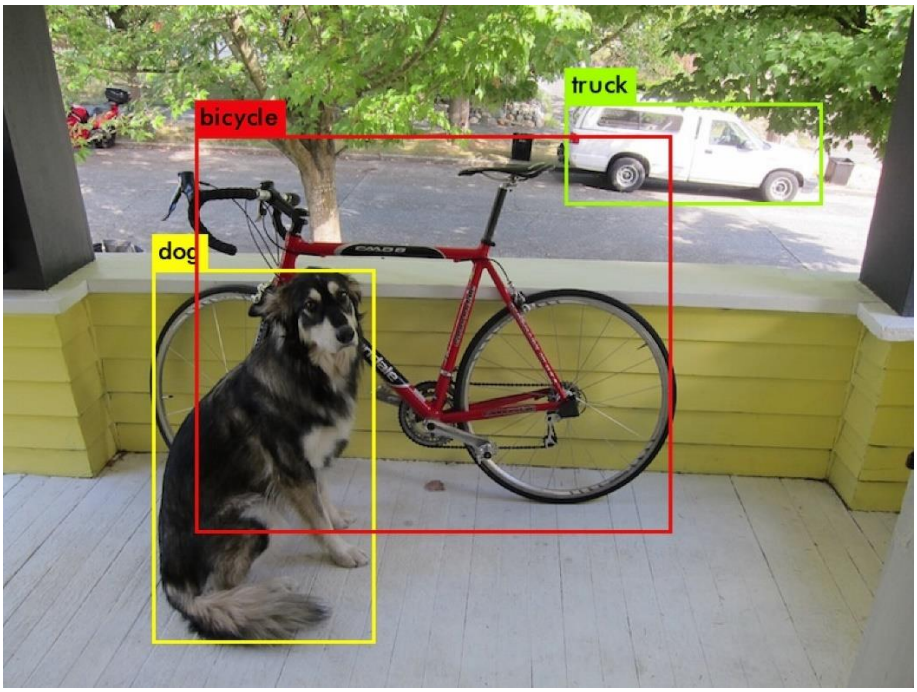

FIGURE 21. A visual representation of an object detection neural network's operation (Redmon 2018)

3.4.2.1 Region Based Convolutional Neural Networks

In 2013, a group of researchers have presented a new concept in object detection models. It was named "region based convolutional neural networks". Its operation was much faster as compared to the previous generation of solutions because of breaking a problem into two parts: region extraction and region classification. The system was called R-CNN – Regions with CNN features. The model uses selective search to first identify regions that have a high probability of containing an object in them. The number of regions generated for a single picture is approximately 2000. Then, all the regions are resized to a 224 by 224-pixel images and fed to a trained convolutional neural network for extracting convolutional features. The convolutional part typically produces 4096 different feature values. Then, all the feature sets are sent to a support vector machine that is classifying regions based on the feature set. After that, a threshold may be applied to remove uncertain regions. As the result, the model returns region coordinates and an associated class probability distributions. (Girshick et al. 2013, 1-3).

The original paper shows that it took about 13 seconds to compute a single image on GPU and about 53 seconds on CPU. The maximum achieved mean average precision on VOC 2007 dataset was 48%. (Girshick et al. 2013, 3).

The precision of object detection solutions may differ a lot from the accuracy of image classification tasks due to not perfectly sized or placed regions. Perfect scoring is not needed in plenty of tasks, so the fast-computing models are very popular in modern development.

3.4.2.2 You Only Look Once model

The YOLOv1 (You Only Look Once version one) architecture was created in 2015. Figure 22 shows that YOLOv1 was orders of magnitude faster as compared to similar models at the cost of insignificant mean average precision decrease. As the name implies, YOLO makes only 1 run through an image.

	mAP	Prediction Time	FPS	Compared to YOLO
R-CNN (VGG-16)	66.0	48.2 hr	0.02 fps	1500x
FR-CNN (VGG-16)	66.9	3.1 hr	0.45 fps	100x
R-CNN (Small VGG)	60.2	14.4 hr	0.09 fps	500x
FR-CNN (Small VGG)	59.2	2.9 hr	0.48 fps	93x
R-CNN (Caffe)	58.5	12.2 hr	0.11 fps	409x
FR-CNN (Caffe)	57.1	2.8 hr	0.48 fps	93x
YOLO	58.8	110 sec	45 fps	-

FIGURE 22. Prediction timing of different architectures on the VOC 2007 dataset (Redmond et al. 2015, 7)

3.4.2.2.1 Operation

First, an algorithm resizes the image to a spatial extent of 448 by 448 pixels. Then, the image is divided into a seven by seven grid. If the center of an object is inside of a grid cell, then that cell is responsible for detecting, providing a bounding box, and classifying that object. Resized image is then sent to a single convolutional neural network. Instead of max pooling, YOLO's CNN uses stride in convolutional layers to decrease the spatial size of data. Since there are 24 convolutional layers, it is needed to reduce the number of channels throughout an operation. Otherwise, the memory consumption would drastically increase. One by one kernel-sized filters are used for reducing a feature space. After the convolutions are done, the data is flattened out and sent to two last fully-connected layers. The first one has 4096 neurons. The second one consists of 1176 neurons. Between the two, there is a dropout layer (Srivastava et al. 2014, 1(1929)-2(1930)) with rate = 0.5 that prevents co-adaptation between these layers. The output from a second dense layer is then shaped in such a way, that it is a 3-dimensional matrix of size 7x7x24, where the first 20 values in an entry are class probability distribution, and the latter 4 are center x, center y, width and height. All the resulting region values are relative to an image and are on a scale of 0-1. The model uses LeakyReLu (Szandała 2020, 14) in every inner layer, and a (linear) logistic activation function in the output layer to produce predictions. (Redmon et al. 2015, 2-4).

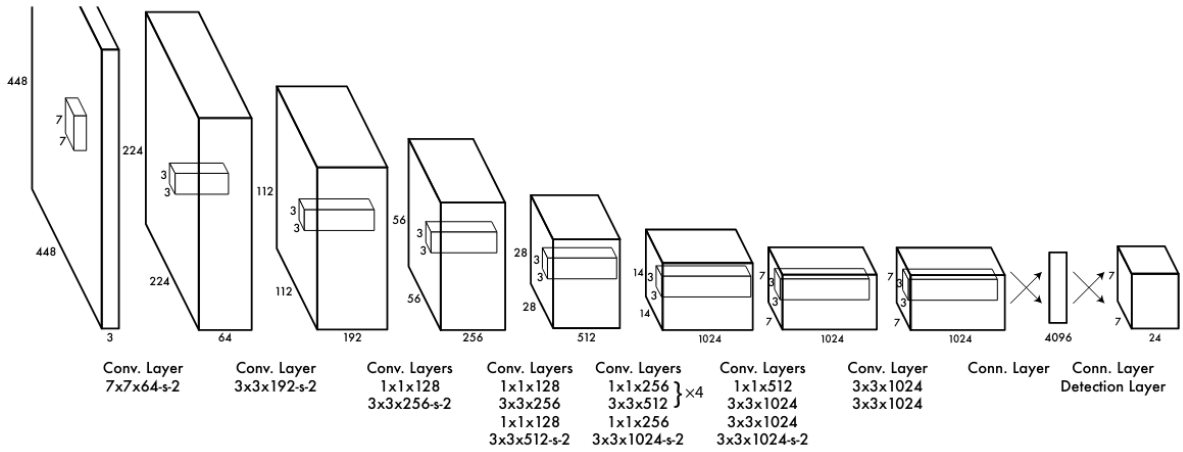

FIGURE 23. A depicture of YOLOv1's architecture. (Redmon et al. 2015, 3)

A sum-square error loss is used in the original model due to its simplicity. However, it was tweaked from the simple form: $loss = \sum_{i=0}^{48}\left(\lambda 1_i^{obj}\left((x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2\right) + \sum_{c \in classes}\left(p_i(c) - \hat{p}_i(c)\right)^2\right)$, where i – cell number, λ – constant scaling factor, x_i- predicted center x value, $\hat{x}_i$- target center x value, y_i- predicted center y value, $\hat{y}_i$- target center y value, w_i- predicted bounding box's width, $\hat{w}_i$- target bounding box's width, h_i- predicted bounding box's height, $\hat{h}_i$- target bounding box's height, $p_i(c)$- predicted probability of class c, $\hat{p}_i(c)$- target probability of class c. $1_i^{obj} = \begin{cases} 1, & \text{if object is present in a cell} \\ 0, & \text{if object is not present in a cell} \end{cases}$. Lambda is used to increase the impact of a bounding box's parameters on a final loss. In original paper, the model used λ = 4. If there is no object in a cell, the model does not need to calculate bounding box's error, but rather focus on a class probabilities. Therefore 1_i^{obj} is used. The squre root is taken from width and height to increase relative influence of errors in small bounding boxes as compared to the bigger ones. (Redmon et al. 2015, 4).

3.4.2.3 Dataset preparation

In this section, there will be shown a way to create a custom object detection dataset. The dataset is going to be in a special YOLO format, but it could be easily converted to different format using specialized software. Special tools will be created and used to fasten the process. The thesis shows a practical way of using neural networks in custom scripts.

3.4.2.3.1 Format

The YOLO model's authors have introduced a new dataset format. Each picture has its own .txt file, in which there is 1 labeled bounding box set of values per line. The format is:

<object-class> <x> <y> <width> <height>

<object-class> <x> <y> <width> <height>

<object-class> <x> <y> <width> <height>

Object class is a class number. X and y are bounding box's center position values relative to an image, e.g. pixel number / total amount of width or height pixels. Width and height are bounding box's width and height values. They are also relative to an image. (Redmon 2018).

Example:

img_0001.png: an image in .png format.

img_0001.txt:

2 0.3 0.6 0.02 0.09

5 0.8 0.1 0.3 0.06

3.4.2.3.2 Original data

The visual dataset is taken from publicly available live camera footage from Helsinki's port harbor. The livestream is hosted on a popular video streaming website. Since the video is too long, it is needed to download only a part of it. A free and open-source tool ffmpeg could be used for extracting and downloading a specified timeframe of a video stream. The first step is to acquire the video stream link using specialized tools. Then, download a 15-minute part of a stream using ffmpeg: *ffmpeg -ss 00:10:00.00 -i "stream link" -t 00:15:00.00 -c copy dataset_video.mp4.* Since the camera is moving, a video needs to be split into multiple parts. This could be done in many ways but using a visual video editor is the simplest solution. Kdenlive is a free and open-source video editing software that will be used here. First, it is needed to import a video. Then, set zone ticks on the time scale (Figure 24). After that, a part of the video could be saved as a separate video (Figure 25).

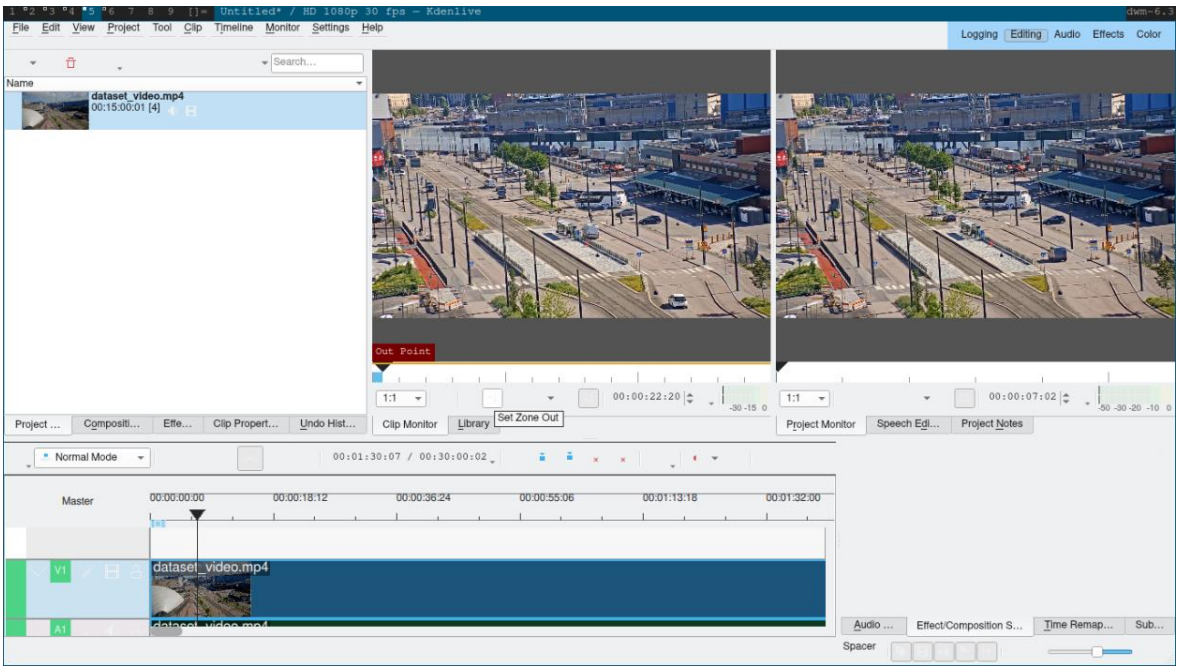

FIGURE 24. Zoning in Kdenlive

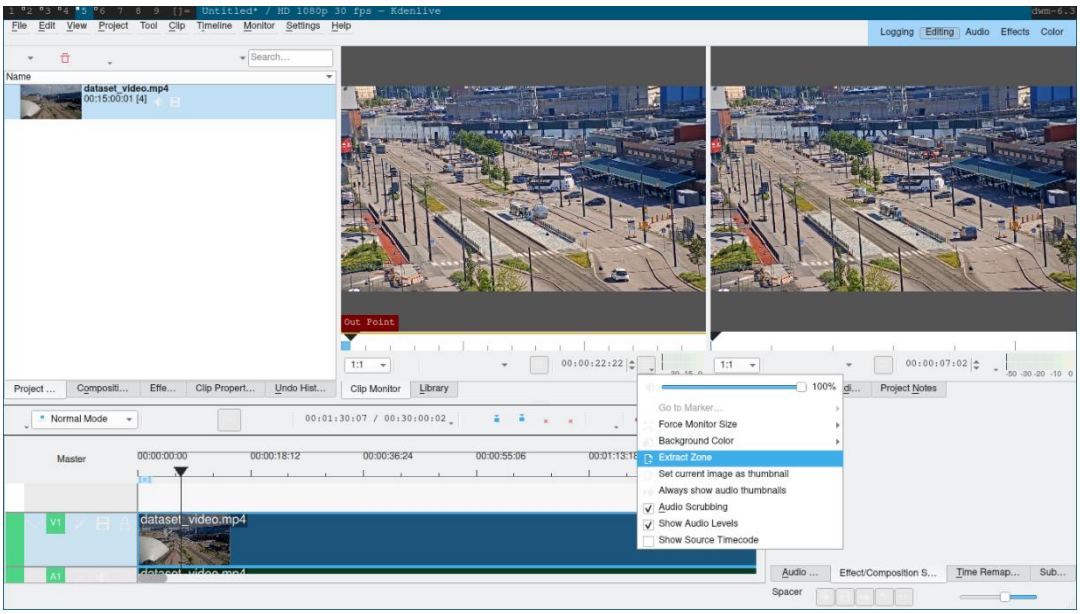

FIGURE 25. Extracting a zone to a separate video

3.4.2.3.3 Automatic annotation

Since humanity have already created and trained huge neural networks, it is possible to use one to automatically annotate the data. The pretrained YOLOv3 model will be used for that. OpenCV's DNN tools will be used as the base system for loading a pretrained YOLOv3 network.

There are a couple of goals that the script will accomplish:

1. Create a dataset that consists of only the needed classes.

2. Detect objects on a picture that are small enough to be not detected via normal pretrained YOLO operation (due to original 416x416 input resizing).

3. Save the results in an appropriate format.

3.4.2.3.3.1 Code

```
import cv2
import numpy as np

#Load a pretrained yolo network
#YOLO-608 works best
#Link: https://pjreddie.com/darknet/yolo/
#Save config as cfg.cfg and weights as yolov3.weights in the folder containing this
script
model = cv2.dnn.readNetFromDarknet("cfg.cfg", "yolov3.weights")
model.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)

#Path to the directory with video file in it
#NOTE: It gets really messy
dir_path = "./frames/"
#Name of the video file
```

```
video_name = "input_video.mp4"

#Array of COCO classes that we are interested in
#Person, car, truck, boat
interested_classes = [1, 3, 8, 9]
#Array of COCO class names for generating new class list
coco_names_indexed = ["person","bicycle","car","motorcycle","air-
plane","bus","train","truck","boat","traffic light","fire hydrant","stop sign","parking
meter","bench","bird","cat","dog","horse","sheep","cow","elephant","bear","zebra","gi-
raffe","backpack","umbrella","handbag","tie","suitcase","frisbee","skis","snow-
board","sports ball","kite","baseball bat","baseball glove","skateboard","surf-
board","tennis racket","bottle","wine glass","cup","fork","knife","spoon","bowl","ba-
nana","apple","sandwich","orange","broccoli","carrot","hot dog","pizza","do-
nut","cake","chair","couch","potted plant","bed","dining table","toilet","tv","lap-
top","mouse","remote","keyboard","cell phone","microwave","oven","toaster","sink","re-
frigerator","book","clock","vase","scissors","teddy bear","hair drier","toothbrush"]

#Counter of processed frames. Used for file naming
frame_number_file = 0

#Define thresholds for the subframe level
probability_threshold_subframe = 0.01
nms_threshold_subframe = 0.2
#Define thresholds for the frame level
probability_threshold_whole = 0.02
nms_threshold_whole = 0.2

#Returns 9 subframes of a frame: left-mid-right, top-mid-bot
def getNineSubframes(frame):
    #Find frame's dimensions
    frame_height, frame_width, frame_channels = frame.shape
    #Array for storing subframes
    subframes = []
    #Now we need to get parts of the image
    #There will be 9 subframes total
    #The subframe step is going to be 1 quarter of a whole image for both width and
height
    for height_pos in np.arange(0.25,1.0,0.25):
        for width_pos in np.arange(0.25,1.0,0.25):
            #Append a new subframe to an array
            subframes.append(frame[int(frame_height * (height_pos-
0.25)):int(frame_height * (height_pos + 0.25)), int(frame_width * (width_pos-
0.25)):int(frame_width * (width_pos + 0.25))])
    #Return the resulting array of subframes
    return subframes

#Function for processing the whole frame
def process_frame(frame, model):
    #Operation:
    #1. Divide each frame into 9 same-sized parts
    #2. Process each subframe
    #3. Mathematically process each bounding box to be relative to the whole frame
    #4. Apply NMS on the resulting data
```

```
    #Frame counter for saving results
    global frame_number_file
    #Output layers of the YOLO network
    global output_layers
    #Save the frame itself
    cv2.imwrite(dir_path + str(frame_number_file) + ".jpg", frame)

    #String for storing the output values. It is going to be printed in a file
    frame_boxes = ""
    #Find frame's dimensions
    frame_height, frame_width, frame_channels = frame.shape
    #Array of frame's subframes
    subframes = getNineSubframes(frame)

    #Conver all subframes into YOLO-specific blobs
    for idx, subframe in enumerate(subframes):
        subframes[idx] = cv2.dnn.blobFromImage(subframe, 1/255, (416, 416), [0,0,0], 1,
crop=False)

    #A holder for the subframes' bbox outputs(same order)
    bboxes_from_subframes = []
    #A holder for the subframes' class outputs(same order)
    classes_from_subframes = []
    #A holder for the subframes' confidences outputs(same order)
    confidences_from_subframes = []

    #Process each subframe, e.g. forward each subframe through YOLO and store the re-
sults
    for subframe in subframes:
        #Feed the blob to the neural network
        model.setInput(subframe)

        #Run forward the neural network and get the outputs from the output layers
        out = model.forward(output_layers)

        #Process the outputs and store classes, bboxes and confidences arrays for the
subframe
        subframe_classes, subframe_bboxes, subframe_confidences = process_subframe(out)

        #Everything is stored in the original format: Left-mid-right, top-mid-bot
        bboxes_from_subframes.append(subframe_bboxes)
        classes_from_subframes.append(subframe_classes)
        confidences_from_subframes.append(subframe_confidences)

    #Now, we have to process bboxes, since they are relative to the subframe, not the
whole frame
    #There are 3 subframes in each row. The width relative offsets are: 0, 0.25, 0.5
    #There are 3 rows of subframes. The height relative offsets are: 0, 0.25, 0.5
    #Each subframe's width and height is 1/2 of the whole frame. Thus, we only need to
scale the bbox's width and height by 1/2
    #Initialize offsets
    cx = 0
```

```
    cy = 0
    #Go through each subframe
    for idx, bboxes in enumerate(bboxes_from_subframes):
        #Go through each subframe's bbox
        for idx1, bbox in enumerate(bboxes):
            #Bbox is : x,y,width,height
            #Scale bbox to be relative to the whole frame
            bboxes[idx1][0] = bbox[0] * 0.5 + cx
            bboxes[idx1][1] = bbox[1] * 0.5 + cy
            bboxes[idx1][2] = bbox[2] * 0.5
            bboxes[idx1][3] = bbox[3] * 0.5
        #No need to save new bbox since we were working with pointers
        #Shift the width offset
        cx += 0.25
        #If the width offset is bigger than 0.5, change to the next row
        if cx > 0.5:
            #Restart at the beginning
            cx = 0
            #Change the height offset
            cy += 0.25

    #Now it's time to apply non-maxima surpression. It will delete bboxes of objects
that were detected on multiple subframes
    #But before that, we need to flatten the bboxes, classes and confidences arrays to
not to have a subframe dimension
    #Array of frame's bounding boxes
    global_bboxes = []
    #Go through each subframe
    for subframe_bboxes in bboxes_from_subframes:
        #Go through each bbox
        for bbox in subframe_bboxes:
            #Append bbox to the global bbox array
            global_bboxes.append(bbox)

    #Array of frame's classes
    global_classes = []
    #Go through each subframe
    for subframe_classes in classes_from_subframes:
        #Go through each class
        for box_class in subframe_classes:
            #Append class to the global class array
            global_classes.append(box_class)

    #Array of frame's confidences
    confidencess = []
    #Go through each subframe
    for subframe_confidencess in confidences_from_subframes:
        #Go through each confidence
        for confidence in subframe_confidencess:
            #Append confidence to the global confidences array
            confidencess.append(confidence)

    #Apply the non maxima surpression on all of the bounding boxes.
```

```
    #This will remove all the excessive bboxes that were foung on the sides of subframes
    nms_indexes = cv2.dnn.NMSBoxes(global_bboxes, confidencess, probability_thresh-
old_whole, nms_threshold_whole)

    #Go through each index that was chosen via NMS
    for i in nms_indexes:
        #Get the bounding box
        box = global_bboxes[i]
        #Get the class
        box_class = global_classes[i]
        #Convert the class to custom index
        box_class = interested_classes.index(box_class)

        #Add new entry to the string that will be written as an annotation to the frame
        frame_boxes += str(box_class) + " "
        frame_boxes += str(box[0]) + " "
        frame_boxes += str(box[1]) + " "
        frame_boxes += str(box[2]) + " "
        frame_boxes += str(box[3]) + " "
        frame_boxes += '\n'

        #I also want to show the bounding boxes for visual inspection during processing
        #First step is to scale them(values are in 0-1 format, need to multiply by orig-
inal w and h)
        #Reformat center coordinates from relative format to pixel format
        cx = int(box[0] * frame_width)
        cy = int(box[1] * frame_height)
        #Reformat width and height from relative format to pixel format
        w = int(box[2] * frame_width)
        h = int(box[3] * frame_height)
        #Get top left corner
        #To get to the left, we need to substract width/2 from the center coordinate
        x = int(cx - w/2)
        #To get to the top, we need to substract height/2 from the center coordinate
        y = int(cy - h/2)
        #Get bottom right corner
        #To get to the right, we need to add width/2 to the center coordinate
        x1 = int(cx + w/2)
        #To get to the bottom, we need to add height/2 to the center coordinate
        y1 = int(cy + h/2)
        #Draw a bounding box using resulting coordinates
        cv2.rectangle(frame, (x,y), (x1, y1), (0,255,0), 2)

    #Now i want to save the file that contains generated annotations
    #Open/create a new txt file with a processed frame's number as name
    f = open(dir_path + str(frame_number_file) + ".txt", 'w')
    #Write all the detections to the file
    f.write(frame_boxes)
    #Close the file
    f.close
    #Increment processed frames counter
    frame_number_file += 1
```

```
    #Show the image with all the resulting bounding boxes
    cv2.imshow("test", frame)

#A function for processing subframes. Returns bboxes, corresponding classes and confi-
dences
def process_subframe(out):
    #Array of subframe's bboxes
    bboxes = []
    #Array of subframe's bboxes' confidences
    confidences = []
    #Array of subframe's bboxes' classes
    classes = []

    #Go through each YOLO's output layer
    for layerOut in out:
        #Go through each detection
        for detection in layerOut:
            #Find the most probable class
            max_prob = 0
            max_class = 0
            #We only want to know certain classes
            for class_interested in interested_classes:
                #We need to add 4 to the index, since first 5 values are the bounding
box properties(cx,cy,w,h,probability that there is an object at all)
                #Indexsing of arrays start from 0, and indexing of COCO classes starts
from 1, which gives index of 5 for the first class(person)
                if(detection[class_interested + 4] > max_prob):
                    #Update if new max
                    max_prob = detection[class_interested]
                    max_class = class_interested

            #If threshold is passed
            if max_prob > probability_threshold_subframe:
                #Append a new class entry
                classes.append(max_class)
                #Append a new bbox entry(in the subframe-relative format)
                bboxes.append([detection[0], detection[1], detection[2], detection[3]])
                #Append a new confidence entry
                confidences.append(max_prob)

    #Apply NMS on the subframe level
    nms_indexes = cv2.dnn.NMSBoxes(bboxes, confidences, probability_threshold_subframe,
nms_threshold_subframe)
    #Holders for the remaining bboxes, classes and their confidences
    nms_bboxes = []
    nms_classes = []
    nms_confidences = []

    #Pick only the ones that have passed NMS
    for i in nms_indexes:
        #Append them to corresponding arrays
        nms_bboxes.append(bboxes[i])
        nms_classes.append(classes[i])
```

```
        nms_confidences.append(confidences[i])

    #Return resulting arrays
    #We also need to return the confidences for further NMS process
    return nms_classes,nms_bboxes, nms_confidences

#Write a new class names file
f = open(dir_path + "classes.txt", 'w')
for label in interested_classes:
    #Insert a corresponding indexed label name
    f.write(coco_names_indexed[label-1] + "\n")
f.close()

#Get a video capture
cap = cv2.VideoCapture(dir_path + video_name)

#Get output layer names. They may differ from model to model.
layers = model.getLayerNames()
output_layers = []

#Get output layers' indexes from the yolo's built-in method
for i in model.getUnconnectedOutLayers():
    #Append the output layer. Built-in method's indexing starts from one, so need to
substract 1
    output_layers.append(layers[i-1])

while cap.isOpened():
    #Read a new frame
    ret, frame = cap.read()

    #Check if video is over
    if not ret:
        print("The file was processed.")
        break

    #Process the frame
    process_frame(frame, model)

    #Check if interrupted
    if cv2.waitKey(1) == ord('q'):
        print("Keyboard interrupt. Exiting...")
        break

#End the job
cap.release()
cv2.destroyAllWindows()
```

3.4.2.3.3.2 Code explanation

The code first breaks input video into frames. Then, it is breaking a frame into 9 equally sized sub-frames. Each subframe is one half of frame's width and height in spatial dimensions. The middle row

and column subframes are overlapping half of the corner ones. This is done to prevent missing detections on the sides of subframes. After that, each subframe is processed using pretrained YOLOv3 darknet model. Then, all the collected bounding boxes are mathematically processed in such a way, that their spatial properties are relative to the whole frame. The next thing that the script does is deleting bounding boxes of objects that were detected on multiple subframes simultaneously, e.g. duplicates. The script also generates a file with verbal class names that were defined. The line number is an index of a class. Example:

classes.txt:

person

car

truck

The code breaks image to subframes in order to increase the resolution of input image. The original image size is 1920x1080 pixels. When it is resized to 416x416 pixels, a lot of objects are squished to the point, where pretrained neural network would not correctly detect them. Fully covering a picture using a 1/3-sized subframes would result in 21 + 4 = 25 subframes, which would drastically increase computational time.

3.4.2.3.3.3 Results

On average, it takes about 1600 milliseconds to process a single frame on Intel Core i7-11800H processor, and 350 milliseconds on an NVIDIA GeForce RTX 3050 TI Laptop GPU. As could be seen from Figure 26, the automatic annotation is not perfect. There are many artefacts and missing labels. Due to this reason, it is needed to manually correct the resulting labels.

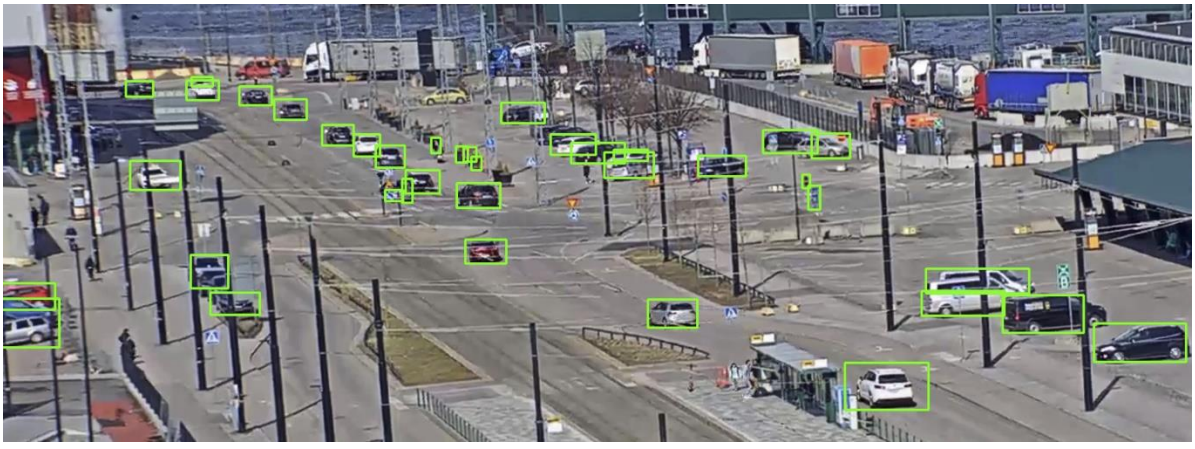

FIGURE 26. Results of an automatic annotation.

3.4.2.3.4 Manual annotation using LabelImg

LabelImg is a free and open-source software that is made for annotating data. It could be used for object detection dataset creation. It is capable of exporting the results in a number of different formats: PASCAL VOC, YOLO and CreateML (Lin 2015). The YOLO format is the needed one. Another reason to use LabelImg is the fact that it can open a folder with images and load all the predefined labels. The custom workflow consists of a couple of steps:

1. Check the correctness of automatically labeled bounding boxes. Delete excessive ones.

3. Resize present bounding boxes.

4. Manually add missing labels.

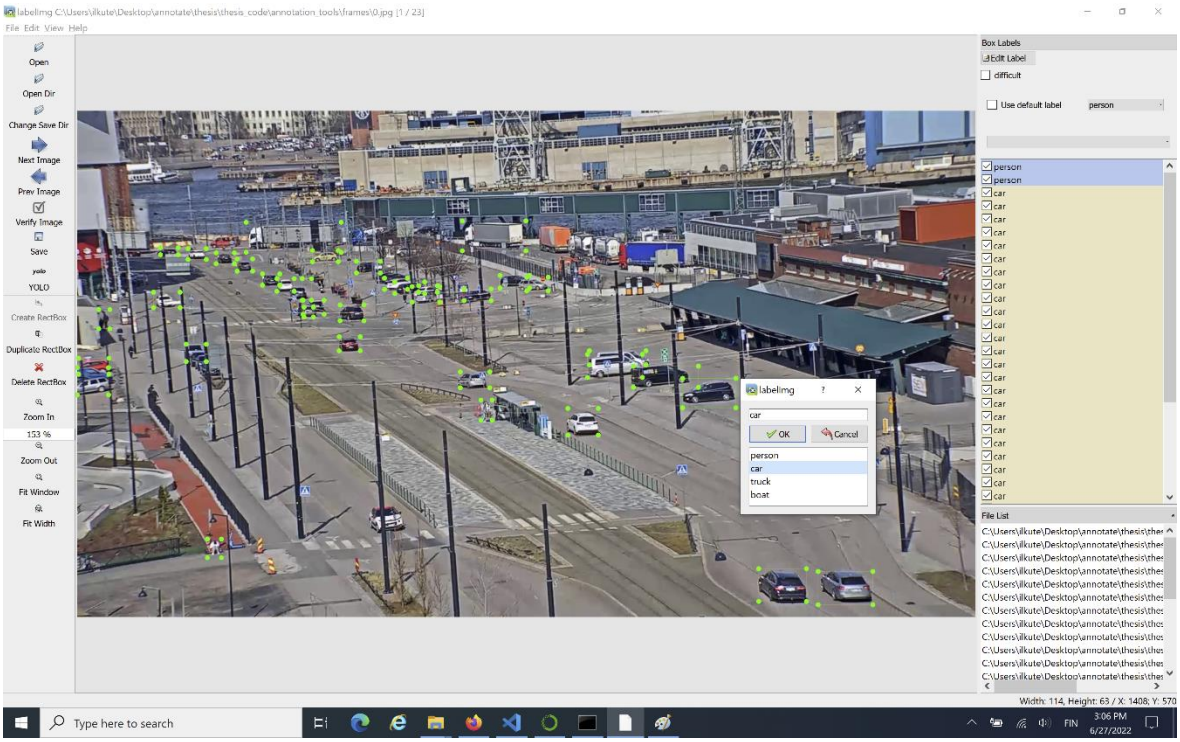

FIGURE 27. Labeling process in LabelImg

3.4.2.3.5 Static bounding boxes

3.4.2.2.5.1 Process

There are many steady objects (parked cars, moored ships, sitting people, etc.) in a picture, so it is possible to ease labeling process by propagating static bounding boxes. For doing so, it is needed to annotate only steady objects in the first frame. Then, rename the resulting file to static_labels.txt. After that, run the script that will propagate first frame's labels to all the present label files.

3.4.2.2.5.2 Code

static_label_propogation.py:

```
from os import listdir
#USAGE: Save the static bboxes as dir_path/static_labels.txt

#Path to the frames/labels directory
dir_path = "./frames/"
#Load the static labels
f = open(dir_path + "static_labels.txt", "r")
#Read the contents
bboxes = f.read()
f.close()

#Get all file names in the directory
all_files = listdir(dir_path)
#Array of label text files
text_files = []
#Go through each filename in the directory
for filename in all_files:
    #If file has a name ending with .txt
    if filename.endswith(".txt"):
        #Add filename to the text files array
        text_files.append(filename)

#Go through each file
for frame_labels in text_files:
    #We don't need to change classes file
    if frame_labels == "classes.txt":
        continue
    #Path to the frame label file
    fname = dir_path + frame_labels
    #Append static labels at the end of the file
    f = open(fname, "a")
    f.write(bboxes)
    f.close()
```

3.4.2.2.5.3 Code explanation

The code is fairly simple: it takes the contents of a static_labels.txt file and appends it to all of the present .txt files in a frames directory. The only exception is a classes.txt file which stores indexed class' names.

4 DISCUSSION

The thesis has shown the basic processes of modern vision-based neural networks. It has started from mathematically computing an abstract artificial neural network's operation. Then, the same logic was implemented in code using a Python programming language and a NumPy library. The custom model was able to learn to differentiate 9 dissimilar handwritten digits. It took a lot of effort to implement it.

The second part of the thesis have presented a walk-through of convolutional neural network processes. It got started from explaining convolutional tools' workflow in an abstract and mathematical way. Then, the author used a modern library TensorFlow for creating a custom image classification model for the CIFAR-10 dataset. During the creation, an author got an understanding of how to optimize model's architecture in such a way, that it is giving the best accuracy/performance results. The program was also using a Matplotlib library for visual observation during the training process. Then, an author explained a different type of problem: object detection. He decided to not to implement and train the network itself, but rather develop a special workflow for creating custom datasets. Author used pretrained YOLO model for automating a labeling process. This could save a lot of time during the process of creating a custom dataset. Since the input image has a very high resolution, a pretrained model was not capable of detecting many objects. The author made an elegant work-around for this problem. As a result, the author was able to create a dataset for training a specific neural network that would, for example, detect and differentiate a car from a truck represented by only a little number of pixels.

In my opinion, neural networks are becoming an integral part of human life. In the last 30 years, neural networks have made a huge leap from being able to detect human faces to safely driving a car in a traffic. Neural networks could be used for almost anything: medicine, autopilots, advisors, seismological predictors, weather forecasts, stocks trading, and many more. So, being able to create working solutions would give me a great basis for future life. I may continue working on my custom library later on. Implementing convolutional layer would be a good challenge, which would open an opportunity to create and test a lot of different architectures.

Although my signature programming language is C, I have decided to use Python for this thesis. I have faced a lot of high-level interpreted language's sugar, and it is motivating me to learn more different languages.

REFERENCES

Dennis DeCoste 1997, The Future of Chess-Playing Technologies and the Significance of Kasparov Versus Deep Blue

Kyriaki Sidiropoulou, Eleftheria Kyriaki Pissadaki and Panayiota Poirazi 2006. EMBO Reports vol. 7(9); 2006 Sep. Inside the brain of a neuron.

Jürgen Schmidhuber 2014. Deep Learning in Neural Networks: An Overview.

Chigozie Enyinna Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall 2018, Activation Functions: Comparison of Trends in Practice and Research for Deep Learning

Tomasz Szandała 2020, Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks.

Eli Bendersky 2016. The Softmax function and its derivative. Eli Bendersky's website. https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative. Accessed 10.06.2022.

Lei Feng, Senlin Shu, Zhuoyi Lin, Fengmao Lv, Li Li, Bo An 2020. International Joint Conference on Artificial Intelligence. Can Cross Entropy Loss Be Robust to Label Noise?

Yann LeCun, Leon Bottou, Yoshua Bengio, Patrick Haffner 1998. Gradient-Based Learning Applied to Document Recognition

Ilango Gogul, Sathiesh Kumar 2017. Flower Species Recognition System using Convolution Neural Networks and Transfer Learning.

Matteo Carandini 2006. The Journal of Physiology vol. 577(Pt 2). What simple and complex cells compute.

Xu Kang, Bin Song and Fengyao Sun 2019. A Deep Similarity Metric Method Based on Incomplete Data for Traffic Anomaly Detection in IoT.

Vincent Dumoulin and Francesco Visin 2018. A guide to convolution arithmetic for deep learning.

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng 2016. TensorFlow: A System for Large-Scale Machine Learning.

Alex Krizhevsky 2009. Learning Multiple Layers of Features from Tiny Images.

Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, Hervé Jégou 2021. Going deeper with Image Transformers.

Hamner Ben 2017. Popular Datasets Over Time. Kaggle. https://www.kaggle.com/code/benhamner/popular-datasets-over-time/data?select=popular_data_references_by_year.png. Accessed 14.06.2022.

Henry A. Rowley, Shumeet Baluja, and Takeo Kanade 1998. Neural network-based face detection. IEEE Transactions on Pattern Analysis and Machine Intelligence (Volume: 20, Issue: 1, Jan 1998).

Paul Viola, Michale Jones 2001. Rapid Object Detection using a Boosted Cascade of Simple Features.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik 2013. Rich feature hierarchies for accurate object detection and semantic segmentation.

Joseph Redmon 2018. YOLO: Real-Time Object Detection. Object detection image example. https://pjreddie.com/darknet/yolo/. Accessed 20.06.2022.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov 2014. Journal of Machine Learning Research 15. Dropout: A Simple Way to Prevent Neural Networks from Overfitting.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi 2015. You Only Look Once: Unified, Real-Time Object Detection.

Joseph Redmon 2018. YOLO: Real-Time Object Detection. Training YOLO on VOC. https://pjreddie.com/darknet/yolo/. Accessed 20.06.2022.

TzuTa Lin 2015. LabelImg. GitHub. https://github.com/tzutalin/labelImg. Accessed 20.06.2022.

APPENDIX 1: A CODE FOR GENERATING ACTIVATION FUNCTION PLOTS

```
import numpy as np
from matplotlib import pyplot
import math

def relu(x):
    if x > 0:
        return x
    else:
        return 0

def linear(x):
    return x*3

def sigmoid(x):
    return 1/(1 + math.e ** (-1 * x))

def softmax(x):
    result = []
    for i in x:
        result.append(i/sum(x))
    return result

#x = np.arange(-5, 5, 0.1)
#y = list(map(sigmoid, x))
#pyplot.plot(x,y)
#pyplot.xlabel("input")
#pyplot.ylabel("output")
#pyplot.title("Sigmoid")
#pyplot.grid()

x = [0.6, 0.88, 9, 2, 0.11, 0.4, 5]
y = softmax(x)
space = [1,2,3,4,5,6,7]
pyplot.bar(space,y)
pyplot.xlabel("entry")
pyplot.ylabel("output")
pyplot.title("Softmax of [0.6, 0.88, 9, 2, 0.11, 0.4, 5]")
pyplot.grid(axis='y')

pyplot.show()
```