Fachhochschule Dortmund

Student Research Project

Introduce AI into kinematics

Kai Lawrence

June 5, 2020

Contents

1 Introduction

The first work, which is today assigned to the field of artificial intelligence (AI), was done in 1943 by McCulloch and Pitts [MP43]. In their work they were inspired by neurons in the brain and created first mathematical formulations which later had a great influence on the development of neural networks. The neurons are seen as switches whose activation is influenced by other neurons. In this way complex systems can be created, which are able to solve different tasks. The term artificial intelligence itself was coined by McCarthy in 1956, when he organized a two-month workshop in Dartmouth [McC55].

After the initial high expectations of artificial intelligence, without delivering the promised results, the field received less attention. A famous quote from Herbert Simon in 1957 states:

> It is not my aim to surprise or shock you – but the simplest way I can summarize is to say that there are now in the world machines that think, that learn and that create. Moreover, their ability to do these things is going to increase rapidly until – in a visible future – the range of problems they can handle will be coextensive with the range to which the human mind has been applied.

However, contrary to these exaggerations, progress has been made over the decades. After another period of high investment in the 1980s, without achieving the ambitious goals, the so-called ”AI winter” set in, which was characterized by a lack of academic and economic interest and lack of funding.

Recently, however, research and application of these techniques have been attracting more attention. This is due to a number of factors that are prevalent today. One is the sheer amount of computing power that modern computers can have. Even home computers today are capable of training models of remarkable complexity.

Furthermore, the amount of data available through the Internet facilitates the training of AIs for various tasks. In order to train a model for a specific task, appropriate data are needed to make correct predictions about data never seen before (generalization). Since data is available in large quantities, there are accordingly several databases for research, training and evaluation. A list of databases can be found at `https://aka.klawr.de/srp#1`.

That is why more and more attention is being paid to artificial intelligence in research, and more and more practical areas regard this technology as a solution to a wide range of

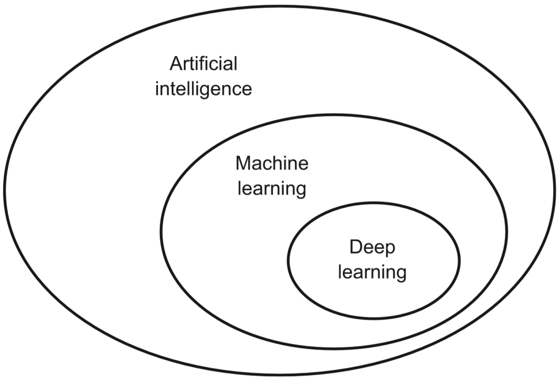


Figure 1: Artificial intelligence, machine learning and deep learning [Cho17, p.4]

problems. Whether the recent enthusiasm is just exaggerated expectations, or whether the ”last invention of mankind” [Goo65] remains to be seen.

The term artificial intelligence is used to describe machine processes that perform tasks that are otherwise characteristic of human intelligence. This paper is mainly intended to deal with a subfield of artificial intelligence, the so-called ”deep learning”. This connection is shown in figure 1.

In this project I want to introduce machine learning into the field of kinematics for two-dimensional sketching and prototype development with `deepmech`. The first solution implemented for this project takes images as input and outputs the respective type of handwritten mechanical symbols. The models created with the presented approach are small and mostly independent of the used programming language, which is made possible by using the popular Keras [Cho19] library on TensorFlow [Goo19b].

Simple demonstrations are performed using JavaScript to connect to an emerging area of kinematics using web technologies such as `mec2` [Gös19c] and `mecEdit` [Uhl19a]. For the training of the current model the Python implementation of TensorFlow is used, because the better utilization of the GPU with CUDA [nvi19] as backend allows a faster training of the Keras models. Thanks to the convertibility of model descriptions for use in different programming languages, this allows seamless transitions between different approaches.

Before showing how deepmech was developed, the basics of machine learning will be covered, with some simple examples as an introduction, introducing the general terminology used in this paper. In chapter 3 the concepts are expanded by introducing neural networks, which are the key to translating the workings of machine learning into practical models. Another topic to be covered is data generation. Data is an important part of the training of statistical models, so data generation and augmentation is covered in

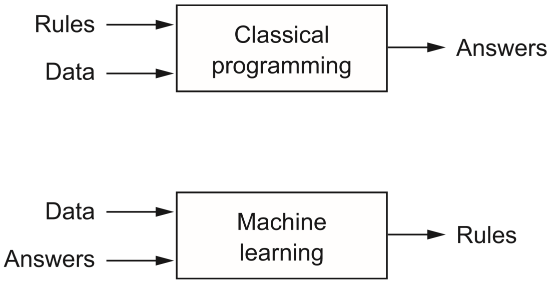


Figure 2: Machine learning, a new programming paradigm [Cho17, p.5]

chapter 4. Then the first working model is shown. All main components of the process are examined in detail to get a better understanding of each function. Chapter 6 deals with optimizing the model to improve accuracy and minimize the size of the model that will be loaded in future applications. Finally, a conclusion is given in which the results are evaluated.

2 What is machine learning?

2.1 Machine learning in general

Usually the task of programming is to apply rules to certain problems in order to obtain solutions. With the help of machine learning we want to create a model that uses solutions for certain problems to find out the corresponding rules. The relationship is shown in figure 2.

The promise of machine learning is that the rules learned can be much more complex than it is possible to program by hand. A good example is image recognition (which will be the focus of this project), where the content of images is to be classified. The relations between the pixels have to be found out, which leads to a rather complex model. This cannot be programmed in the traditional way even for the simplest tasks.

The universal approximation theorem [Cyb89; HSW89] states that for any input x there is a function h which approximates the mapping y, while the mapping is only the input to the correct output (often labeled by a human). This relationship is described in equation (1).

$$\forall \epsilon > 0 : \exists h(\theta, x) : \forall x \in I : |h(\theta, x) - y(x)| < \epsilon \tag{1}$$

The mathematical description of h is called a model (e.g. $h(x) = \sum_{i=1}^{n} \lambda(\theta^T * x)$). $\lambda \in \mathbb{R}$ is a constant and θ is a tensor that defines the parameters of h, which are made up of so-called weights and biases, which will be explained later.

The universal approximation theorem makes no statements about the form of θ; therefore h is called a hypothesis because it states that for a certain composition of θ this equation is true. This term is summarized in the equation (2).

$$\forall \epsilon > 0 : \exists \theta \in X : \forall x \in I : |h_\theta(x) - y(x)| < \epsilon \tag{2}$$

A loss function is used for model valuation; it compares the result of the hypothesis with the data provided. Probably the most prominent loss function is the `mean square error`. It takes the sum of the deviation of n examples squared and divides it by $2 * n$.

$$L_{mse}(\theta, x) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x) - y(x))^2 \tag{3}$$

The task of machine learning is to determine a θ for a model that confirms the equations 1 and 2, which is done by iterative updating of θ. Before training, θ is randomly initiated and then iteratively adjusted to make ϵ converge toward zero. This minimizes the value of L.

In order for the value of the loss function to gradually decrease, θ is adjusted using an optimization function. This is usually done with a gradient descent algorithm or a variant of it. The gradient descent is performed by calculating the gradient of the loss function and subtracting it from the corresponding parameters shown in equation (4).

$$\theta_{i+1} := \theta_i - \eta \nabla_\theta L(\theta, x) \tag{4}$$

Where η is a constant called the learning rate, which helps the loss function converge to 0 using the gradient descent algorithm. It is one of many hyperparameters that are not automatically adjusted during training, but are determined beforehand. Optimal adjustment of hyperparameters is an important task that is difficult to automate and is therefore the focus of much research[1]. If the equation (2) applies to the model in question, it is then sufficiently suitable for the given input.

These concepts are presented now using a simple example.

[1] There are algorithms for adjusting the learning rate, namely `AdaGrad` [DHS10] and derived algorithms and is the main topic of the chapter 6

2.2 Simple linear example

As an illustrative example, a model is created that translates Fahrenheit into Celsius[2]. The original equation is given by the linear function $y = mx + b$ as $F = C * 1.8 + 32$, with F as degrees Fahrenheit and C as degrees Celsius.

A model to learn this relationship is defined in the listing 1 [3] .

Listing 1: Celsius to Fahrenheit

```
const model = {
    w: [Math.random(), Math.random()],
    h(x) {
        return this.w.reduce((pre, cur, idx) => pre + x ** idx * cur, 0);
    }
}

function y(C) { return C * 1.8 + 32; }

function mse(h, y) { return (h - y) ** 2 / 2; }

function sgd(model, x) {
    model.w =  model.w.map((weight, idx) =>
        weight - (model.h(x) - y(x)) * x ** idx);
}

for (i = 0; i < 200; ++i) {
    const C = Math.random() * 100;
    sgd(model, C / 100);
    if (!(i % 20)) console.log('loss: ', mse(model.h(C), y(C)));
}

console.log('weights: ', model.w);
```

The described model is implemented by `w` and the hypothesis function by `h`. Since it is known that the relationship is linear, the model represents a one-dimensional polynomial, with the first index being dimensionless (and, as mentioned above, emulating the bias) and the second being used as input x. We will add a dimension later to demonstrate the behavior with polynomials that do not directly represent the target function.

`y` provides the correct, predefined solution. Please note that `y` is only used during training and is omitted if the model parameters are correctly set after training.

In this example the loss function is described by the *mean square error* [4].

To minimize the loss function gradient descent described in equation (4) is used.

[2] Please note that using a learning algorithm here is very inefficient and is in contrast to Maslow's hammer (`https://aka.klawr.de/srp#2`).

[3] Also available at `https://aka.klawr.de/srp#3`

[4] Abbreviated as *mse*, which is shown in equation (3)

In listing 1 equation (4) is implemented as `sgd` with $\theta_0 = $ `b` and $\theta_1 = $ `m` as shown in equation (5).

`sgd` is short for *stochastic gradient descent*[5]. It represents the optimization function that updates the parameters of the model by calculating the gradient of loss for each parameter and then updating it.

$$\begin{aligned} \theta_0 &:= \theta_0 - \frac{\partial L}{\partial \theta_0} = \theta_0 - (h(x) - y(x)) \\ \theta_1 &:= \theta_1 - \frac{\partial L}{\partial \theta_1} = \theta_0 - (h(x) - y(x)) * x \end{aligned} \tag{5}$$

Listing 2: Output of C to F converter.

```
loss:  343041.666256673
loss:  16242.7707476548
loss:  88.27595280422442
loss:  10.74432403839251
loss:  1.1607137779951606
loss:  0.252186283500194
loss:  0.00040450626910586374
loss:  0.000042470333032O6121
loss:  7.158429555712516e-7
loss:  1.6948019668728757e-10
weights:  [ 31.99999973758735, 1.8000003977458756 ]
```

The expected behavior for the loss function is to decrease with each iteration. This is the case in this example, showing that the deviation of the hypothesis from the actual result decreases. After 200 iterations, the parameters `m` and `b` of the `model` are displayed and show that they are actually close to the expected result. By increasing the number of iterations, the result can be increased up to the point where they are rounded by the JavaScript compiler to represent exact results (using Node.js[6]).

Provided that this hypothesis is initially well suited to the task, it is important to note that by modifying the hypothesis to represent a polynomial of higher degree (e.g. $y = nx^2 + mx + b$) the additional parameters converge towards 0, which effectively gives the same result as before [7].

The experiment can be followed at `https://aka.klawr.de/srp#4`.

[5] A distinction is made between batch, mini-batch and stochastic gradient descent; using a complete data set, a defined subset or individual data on the individual iterations of the training process

[6] As far as can be assumed when calculating exact results using floating point numbers.

[7] But to achieve similar results, the iteration number must be increased by at least a factor of 20

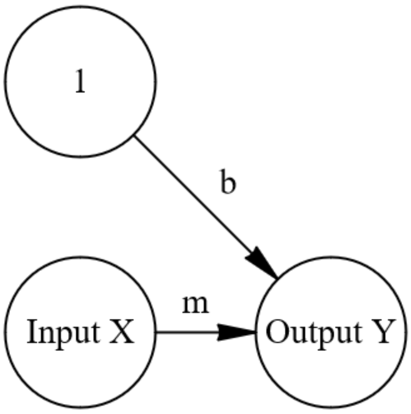


Figure 3: This example can be visualized as shown here. The bias can be interpreted as additional input, which always has the value 1. The input is multiplied by the parameters marked with the corresponding arrows, which in turn are summed up (resulting in $y = mx + b$).

3 Neural Networks

As already mentioned, the name Neural Network has become widely accepted, although it is neither neural nor a network. They are used to create more complex mappings than the previous models were able to do. In the following we will discuss what the motivation behind this concept is and how networks are trained.

3.1 Deep learning

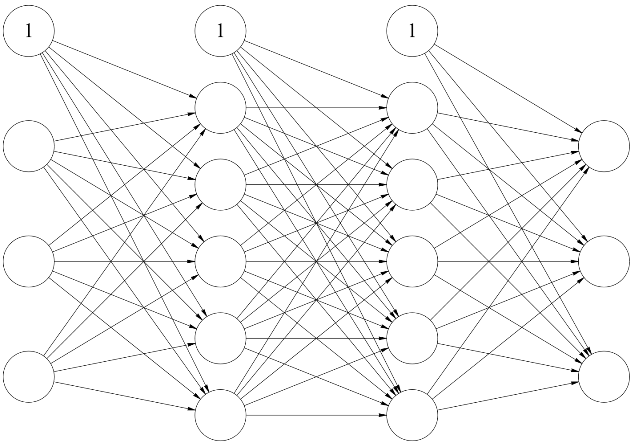
Although the universal approximation theorem shows that all continuous functions can be described with only one layer, in practice it is much more reasonable to add several layers to our model to keep the required resources low. Rolnick and Tegmark [RT17] show that in single-layer models the number of neurons grows exponentially with the number of variables of the input, whereas multi-layer neural networks grow only linearly.

Each node in figure 4 can be addressed using the notation z_i^l. l is the number of the layer in which the node is located and i is the index of the node. Each weight $\theta_{i,j}^l$ can also be addressed; l is the layer to which the weight refers, i is the index of the node, which is multiplied by the weight to influence the value of the node in the layer l with the index j. So we can describe each node as the result of the following equation[8]:

$$z_j^l = \sum_{i=0}^{n} z_i^{l-1} \theta_{i,j}^l \tag{6}$$

[8] In literature the bias is often added separately. According to the convention of prefixing the input vector x with a 1, it is always included as index 0 of each layer. Please note that, mathematically this is the same.

Figure 4: A neural network with three input nodes, two hidden layers with five nodes each and an output layer with three nodes. The bias is added to each layer by prepending a node with value one. Every node is, like the previous example calculated by the sum of the product of the previous layer and their respective weight.



n is the number of nodes in the respective layer. Equation (6) can be vectorized to:

$$z^l = \theta^l z^{l-1} \tag{7}$$

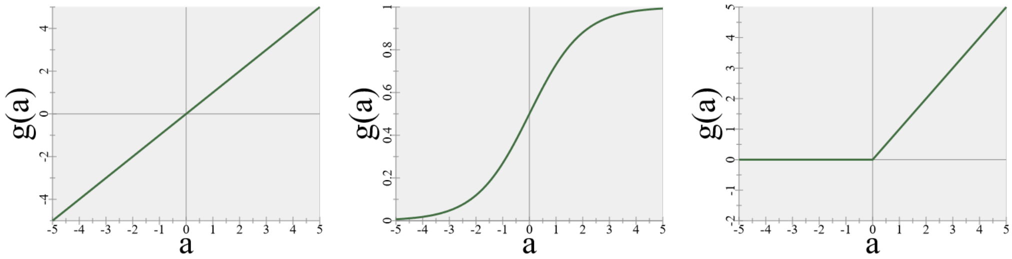
3.2 Activation functions

One problem with the values of the individual cells is that they can be of any size. Numerically, this is very difficult to control, so the nodes themselves are given to another function first; this function is aptly called an activation function.

Activation functions are used in different ways and give more possibilities to adjust yet new parameters to increase model performance. Previously we intrinsically assumed the linear function shown in figure 5a with the mapping $g(a) = a$. The logistic sigmoid is often used in practice, because it creates a mapping of arbitrary numerical input and maps it into the interval $[0, 1]$. The rectified linear unit is much easier computationally and is recommended for use with most feedforward neural networks[GBC17, p.169][GBB11].

Mathematical representations are shown in equation (8)

$$\begin{aligned} Sigmoid : f(x) &= \frac{1}{1 + \exp^{-x}} \\ ReLU : f(x) &= max(0, x) \end{aligned} \tag{8}$$



(a) Linear (b) Logistic sigmoid (c) ReLU

Figure 5: Activation functions

The use of the activation functions can also vary; for example, it is common for the logistic sigmoid to have a threshold to not fire at all if a certain value (e.g. 0.5) is not reached.

With the introduction of activation functions, the equations (6) and (7) become equal:

$$a_j^l = \sigma(z_j^l) = \sigma(\sum_{i=0}^{n} z_i^{l-1}\theta_{i,j}^l) \tag{9}$$

With the corresponding vectorized equation:

$$a^l = \sigma(z^l) = \sigma(\theta^l z^{l-1}) \tag{10}$$

There are a variety of different activation functions, these two being the most prominent. The selection of the activation function and possibly a corresponding threshold value is another hyperparameter that must be selected before the training.

3.3 Backpropagation

With gradient descent (4) not all parameters in all layers can be updated, because it only considers updating the parameters of the last level if it is considered independent of previous layers. Since the last layer is a function of the previous layer, the chain rule can be applied and thus apply the calculated error of the last layer to the preceding one and update the weights and biases. This applies to every layer, except the input layer (which does not need to be updated in the end). The following equations are derived using [Stu18, p.733], [GBC17, p.197] and [Nie15, ch.2]:

$$\mathit{\Delta}^L = \nabla_a L \odot \sigma'(z^L) \tag{11}$$

$$\Delta^l = ((\theta^{l+1})^T \Delta^{l+1}) \odot \sigma'(z^l) \tag{12}$$

$$\theta_{i+1} := \theta_i - \eta \Delta \tag{13}$$

Equation (11) describes the error calculated in the output layer. We define our network with L layers, so the output layer is always described by the L index. $\odot$ is the Hadamard operator. It is used to multiply tensors element by element, which is useful in this case as it prevents the vectors from having to be transformed into diagonal matrices. Equation (11) is a direct result of applying the multivariate chain rule to the derivative of the loss function performed in the last layer. As mentioned above, the gradient descent is performed by the derivative of the loss function in equation (5). Since an activation function is implemented between the input used in the loss function and the output of the previous layer, the chain rule must also be applied to the latter.

$$\frac{\partial L}{\partial z_j^L} = \frac{\partial L}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \tag{14}$$

By pointing out that $\frac{\partial L}{\partial a_i^L}$ is only a non-vectorized form of $\nabla_a L$ and $\frac{\partial a_i^L}{\partial z_i^L}$ of $\sigma'(z^L)$ the equivalence becomes clear.

Equation (12) can be rewritten as follows:

$$\Delta_i^l = \frac{\partial L}{\partial z_i^l} = \sum_{j=0}^{n} (\frac{\partial L}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l}) \tag{15}$$

n is the number of nodes in the next layer. From $\frac{\partial L}{\partial z_i^{l+1}} = \Delta_i^{l+1}$ follows:

$$\frac{\partial L}{\partial z_i^l} = \sum_{j=0}^{n} (\Delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial z_i^l}) \tag{16}$$

The following equation is also known:

$$\begin{aligned} z_j^{l+1} &= \sum_{i=0}^{n} \theta_{i,j}^{l+1} a_i^l = \sum_{i=0}^{n} \theta_{i,j}^{l+1} \sigma(z_i^l) \\ \frac{\partial z_j^{l+1}}{\partial z_i^l} &= \theta_{i,j}^{l+1} \sigma'(z_j^l) \end{aligned} \tag{17}$$

So we can substitute the terms in equation (16):

$$\frac{\partial L}{\partial z_i^l} = \sum_{j=0}^{n} (\mathit{\Delta}_j^{l+1} \theta_{i,j}^{l+1} \sigma'(z_j^l)) \tag{18}$$

Which again is just a non-vectorized form of the equation (12).

Equation (12) is then executed on each layer until $l = 0$ is reached and a $\mathit{\Delta}$ is calculated for each parameter in θ, which are applied by equation (13). This type of backward propagation gave this algorithm the prominent name of ”backpropagation” (short for ”backward propagation of error” [RHW86]).

3.4 Generalization

The task of any algorithm for machine learning is to generalize. After the model has been trained with data, it should also achieve good results with data it has never seen before. This distinguishes machine learning from optimization problems (which could be solved with the Newton-Raphson method, for example).

However, a generalization is not guaranteed. In particular the phenomena of over- and underfitting pose problems. For example, a model that has been trained for a classification task might make correct predictions for training examples, but not work reliably with new data (overfitting). This indicates that the model has, so to speak, memorized the training examples. There are some things which can be done to solve this issue. Giving the model more data for training is a solution shown by Banko and Brill [BB01] and commented by Halevy, Norvig and Pereira in their article ”The Unreasonable Effectiveness of Data” [HNP09]. Adding a lot of data is not always possible because generating data can be costly, so some compromises may have to be made to solve this problem.

Another possibility is to reduce the number of features in the training data; apparently unintuitive at first glance, it becomes clear that an abundance of features does not add much information to an image. For example, to recognize a handwritten digit, many more parameters have to be adjusted if a high-resolution image with hundreds of thousands of pixels is fed into the model, where a pixel count of less than a thousand would already be sufficient [Nie15].

On the other hand, the problem of underfitting is as problematic as overfitting. If the model is already not getting sufficient results from the training data, the problem is called underfitting. If this is the case, it is likely that the model or parameter will need to be adjusted, or that the training data is unsuitable for the problem. Furthermore, for the equations (1) and (2) to be true for a reasonable model, a rule of thumb is that mapping the input to the expected output should be at least relatable to a human.

In order to train a model for the recognition of mechanical symbols using these techniques, the corresponding data must first be generated.

4 Data Generation

Before models for the detection of mechanical symbols can be created, it is necessary to collect data suitable for this task.

The aim is to read handwritten mechanical symbols. The data for training the model are necessarily created by myself at this point. Since the data generation can be time consuming, a way had to be found to create the data as time efficient as possible. Fortunately I had access to a tablet which allows to draw directly on the display. To generate the data quickly, some requirements had to be taken into account:

1. A drawing context is created.
2. The context is variable in size and color.
3. The context should be able to recognize mouse events.
4. The script should have access to the file system to safe data on the hard drive with no extra work required.

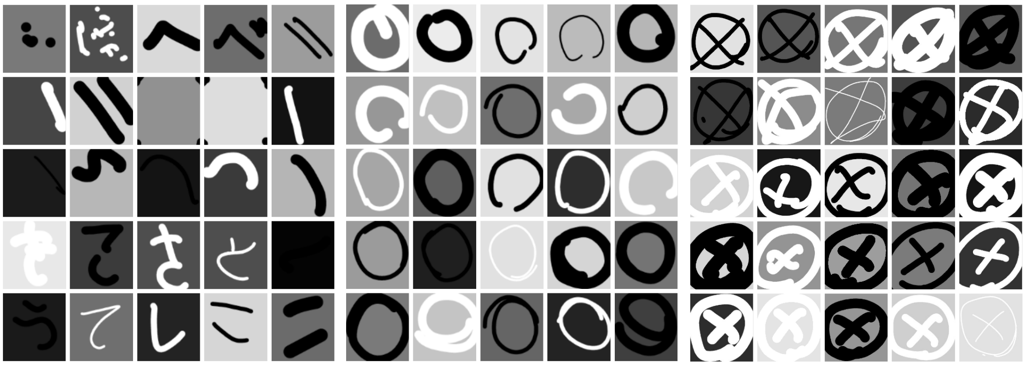
A Python script is able to meet these requirements. The library `OpenCV` [Ope19] allows to create a window with corresponding requirements. The library `opencv-python` [Hei19] is used here implementation of `OpenCV`. It meets all these requirements because it provides the creation of a window with programmable context and functions to handle mouse events[9].

Since the model should be able to recognize symbols of any color on any background, the context background is a random value in grayscale, as is the color of the line drawn on the background. The thickness of the line used for drawing is also random, thus mimicking the appearance of a symbol of different sizes[10].

After the parameters used to draw the context have been set, the function `cv2` is used to `namedWindow` created the function `cv2`. For each interaction with the canvas, the

[9] The data generation should be done using JavaScript in a web context, but the final requirement to secure data does not seem to be easy to meet. Node.js has access to the filesystem, but is inherently "headless" and therefore does not allow any context to be used for drawing.

[10] This should serve to predict images of different resolutions in a later process



(a) Non hits

(b) Nodes

(c) Bases

Figure 6: Some examples of the data created with the described method. Only these three classes were created for the first tests. The images are created centered in a square, so that later any images can be scanned, using squares of different sizes as templates.

`setMouseCallback` function is applied to the generated `namedWindow`, calling a `draw` function that allows to detect and react to mouse events.

Drawing is done by functions triggered by events called `cv2.EVENT_LBUTTONDOWN` and `cv2.EVENT_LBUTTONUP`, which set a boolean `drawing` flag to either `True` or `False`. The drawn image is saved with `cv2.Event_RBUTTONUP`, which names the image after the number of previously automatically drawn images.

The names of the different classes are set at the beginning of the script, with an interactive prompt asking the user which class to fill next, with the options *"x" for base, "o" for link, "n" for no match.* Consideration of *not found* is especially important because later, when searching for any image, most responses are likely to contain no symbol at all.

With this script 500 symbols are created for each class. It is expected that these images will be augmented later to counteract overfitting, but for the task of distinguishing three different classes, this set should be sufficient[11].

5 First model

The Jupyter notebook for this chapter can be viewed at `https://aka.klawr.de/srp#6`.

[11] The code can be viewed at `https://aka.klawr.de/srp#5`

5.1 Project structure

The project structure is an important part to test models efficiently and evaluate them against other models to determine the best performing model. deepmech is built using a modified version of the Cookiecutter Data Science [dri19] structure. It can be viewed at https://klawr.github.io/deepmech.

The root folder contains files such as the license for the entire project. The project is licensed under the MIT license, which allows anyone to modify, distribute or use it for private and commercial use, but with no liability or warranty on my part. In the root folder there is another file requirements.txt, which is part of the installation process and allows to replicate all the code handled in this project.

The ”Cookiecutter Data Science” structure allows a clear division of the data into raw data (which must never be touched), intermediate data (raw data which is modified, supplemented or altered in some way) and processed data (which is to be fed into the model). This data is stored in the data directory.

Logs are created during training with TensorBoard[12]. These logs are distinguished by a timestamp of the respective training run and are located in the directory logs.

The currently used models are located in the models directory. The models in this directory can change at different stages of the project, so the respective reports have their own model directories to store the intermediate models.

The reports directory contains a folder for each report created in this project, including this article[13]. The notebook folder in this report contains Jupyter notebooks [Jup19] used in different chapters. The models are also trained in jupyter notebooks. Jupyter notebooks are interactive Python code development editors that provide an easy way to efficiently write, test and comment code (with markdown). All notebooks with corresponding code are located in the notebooks directory at the root of this project[14]. The respective directory also contains the code used, the trained models, images and the documents created for the creation of this report.

src contains code (except demos used in the reports) that are used in the development. They range from scripts that create appropriate environments for training the models to data generation and augmentation.

[12] TensorBoard is the visualization toolkit of TensorFlow. The visualizations created with TensorBoard are shown later.

[13] Provided with the name srp, for Student Research Project.

[14] https://aka.klawr.de/srp#7

5.2 Loading data

Before the training of the model can begin, the data must be loaded. As mentioned, the data in this project is stored in the `raw` directory of the `data` directory. It is good practice to never work directly with the raw data, so the beginning of each training session includes a preparation phase in which the raw data is copied to the `processed` data directory (with possible extensions or changes in intermediate steps).

Accordingly, the training of a model in this project always starts with the following code:

Listing 3: Loading Data.

```
from os.path import join

raw = join('data', 'raw')
interim = join('data', 'interim')
processed = join('data', 'processed')

from src.utils import reset_and_distribute_data

reset_and_distribute_data(raw, processed, [400, 0, 100])
```

For the first model, the raw data itself is sufficient; therefore, it is simply copied by `reset_and_distribute_data` from `src/utils.py` to the directory of the processed data. The third parameter of `reset_and_distribute_data` is an array of three numbers describing the distribution of training, validation, and test data. In this example, the validation data is omitted because no adjustments are made to the hyperparameters.

Listing 4: Loading data with generator.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

def create_generator(data_dir, batch_size):
    datagen = ImageDataGenerator(rescale=1./255)
    full_path = join(processed, data_dir)
    return datagen.flow_from_directory(
        full_path,
        target_size=(32, 32),
        batch_size=batch_size,
        class_mode='binary')

train_generator = create_generator('train', 20)
test_generator = create_generator('test', 10)
```

After the corresponding directories are filled with data, the `ImageDataGenerator`[15] is used to allow the model to load the data later during the training process. The

[15] `https://aka.klawr.de/srp#8`

generators also define the size of the loaded image, which can be set to 32 pixels wide and 32 pixels high[16] (as opposed to the original image size of 512x512) and a batch size[17] of 20. The `class_mode` is set to `binary` because the input consists of 1D Numpy arrays. These generators also recognize the structure of the raw data (distributed in respective directories using their label as directory name) and thus assign the appropriate labels to them.

5.3 Creation of the model

Listing 5: Definition of first model.

```
from tensorflow.keras import layers
from tensorflow.keras import models

model = models.Sequential()
model.add(layers.Flatten(input_shape=(32, 32, 3)))
model.add(layers.Dense(32,'relu'))
model.add(layers.Dense(32,'relu'))
model.add(layers.Dense(3, 'softmax'))

model.summary()
```

All models used in this project are sequential models. Sequential models propagate the result of each layer in one direction to the next layer.

`Keras` requires that the first layer has a `input_shape`. The input shape of the first model must match the actual size of the input data. All other layers are able to derive the shape of the previous layer, so no further definitions are required.

Layers can be added by using a provided `add` function which accepts layers as parameters which are subsequently added to the model.

Two layers are added, both of which are dense layers with 32 nodes each. The dense layers are defined by connecting all nodes of the previous layer to the nodes of their own layer using weights and biases and behave like the ”traditional” layers used in this project. The two hidden layers are activated with the `ReLU` activation, as suggested in [GBC17, p.168].

The last layer has three nodes in relation to three classes to which the data can be assigned. The activation `softmax` is used, which adjusts the values of the layer so that they add up to the value one. Therefore, the value of the node in the initial layer can be

[16] The size of the images is determined manually, assuming that as long as a human is able to recognize the characteristic features of the image, the algorithm can learn them.

[17] batch size is the number of images used between each backpropagation cycle. If the batch size matches the record size, it is called batch gradient descent, if the batch size is one, it is called stochastic gradient descent. Everything in between is called a mini-batch gradient descent.

considered a measure of the confidence of the model's evaluation for a given input. The node with the highest value is accordingly assumed to be the assumed correct result.

A summary of the model is given as:

Listing 6: Summary of first model.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 3072)              0
_________________________________________________________________
dense (Dense)                (None, 32)                98336
_________________________________________________________________
dense_1 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 99
=================================================================
Total params: 99,491
Trainable params: 99,491
Non-trainable params: 0
_________________________________________________________________
```

Here we see that the θ in the model has 99491 parameters that can be adjusted. This number is the sum of the parameters given by the 3 layers of the model. The first layer has the flattened 3072 $(32 \times 32 \times 3)$ nodes. The second layer multiplies this number $(+1$ for the bias) by 32 and therefore has an additional 98336 nodes. The third layer is the product of $32 + 1$ nodes in the second layer and 32 nodes in the third layer, which gives us 1056 parameters to adjust. The last layer again has $(32 + 1) \times 3$ nodes.

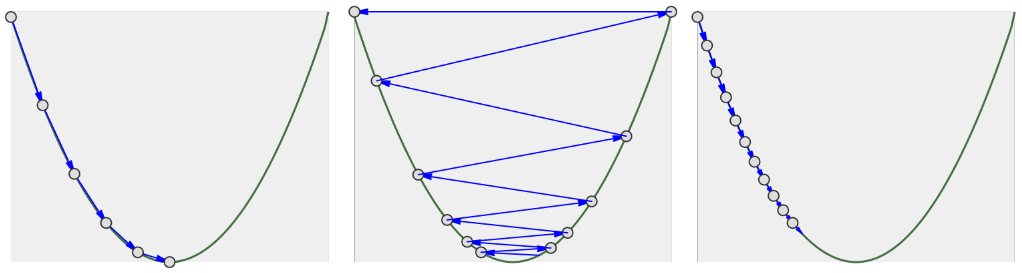
5.4 Optimizer

In order to train our model, an optimizer must be defined. We will start with a SGD optimizer, which is the abbreviation for `Stochastic Gradient Descent`. Stochastic gradient descent is already mentioned in chapter 2.2, where it was implemented very rudimentarily. In the training of this model, the Keras implementation of `SGD` is used. The Keras implementation of SGD differs somewhat from the general definition, since it does not inherently assume a batch size of one, but rather the batch sizes defined in the respective generator, so that the user can decide whether to use full batch, mini batch, or stochastic gradient descent[18].

Listing 7: Definition of an optimizer using Keras' SGD function.

```
optimizer = SGD(lr=0.01, momentum=0.9, nesterov=True)
```

[18] The batch size used here is 20.



(a) Learning rate just right (b) Learning rate too high (c) Learning rate too small

Figure 7: If the learning rate is just right, the value of the loss function approaches a local optimum in an acceptable number of steps. If it is too high, it diverges. If the learning rate is too low, the values converge, but too many steps are needed to achieve this. Please note that these pictures show the parameter space in a very simplified way, since the function shown is completely convex and only two-dimensional.

The function `SGD` requires three parameters: The learning rate `lr`, `momentum` and `nesterov`. These values in turn are hyperparameters that must be determined before training.

5.4.1 Learning rate

The learning rate is one of the most influential hyperparameters and probably the most significant, since it is used in almost every optimizer. It is a scalar value that determines the step size of the steps performed. The update function of the parameter θ is as already mentioned:

$$\theta_{i+1} := \theta_i - \eta\varDelta \tag{13 revisited}$$

.

Where η is the value of the learning rate.

If the learning rate is set too low, the loss function needs many iterations to converge to a sufficiently low value. On the other hand, a learning rate set to a high value cannot converge at all.

5.4.2 Momentum

Introduced in 1964 by Polyak [Pol64], momentum is used to keep the gradient constant over several steps. Momentum takes its name as an analogy to the physical effect according to Newton's second law of motion [New87, p.12]. The direction of the gradient should be consistent over several steps, and it would be strange if the direction made sudden shifts during training. This is useful if the data is ”noisy” or some of the training examples are wrong, since their negative effect is counteracted by the momentum. Therefore another parameter ν is introduced into the equation (13):

$$\begin{aligned} \nu_i &:= \alpha\nu_{i-1} - \eta\mathit{\Delta} \\ \theta_{i+1} &:= \theta_i + \nu_i \end{aligned} \tag{19}$$

In contrast, α is another parameter that can be set to regulate the influence of η on the parameters of the next iteration. Thus, if a training example would change the ”direction” of the next step with a high deviation from the median of the last steps, the effect would be reduced and the convergence rate might be improved.

In 2013 Sutskever, Martens, Dahl and Hinton [Sut+13] introduced another variant of the momentum inspired by Nesterov’s Accelerated Gradient [Nes83]. This variant updates the previously discussed equation to change the parameter for calculating $\mathit{\Delta}$ using ν. Please note that $\mathit{\Delta}$ in equation (19) and (13) is itself a function of the θ parameter (see equation (12)).

$$\begin{aligned} \nu_i &:= \alpha\nu_i - \eta\mathit{\Delta}(\theta_i + \alpha\nu_i) \\ \theta_{i+1} &:= \theta_i + \nu_i \end{aligned} \tag{20}$$

The idea of momentum is that the direction vector points in the right direction, whereas the use of the Nesterov Accelerated Gradient is probably more accurate when one starts measuring the next error, which is slightly shifted in the respective direction [Gér19, p.353] [GBC17, p.291].

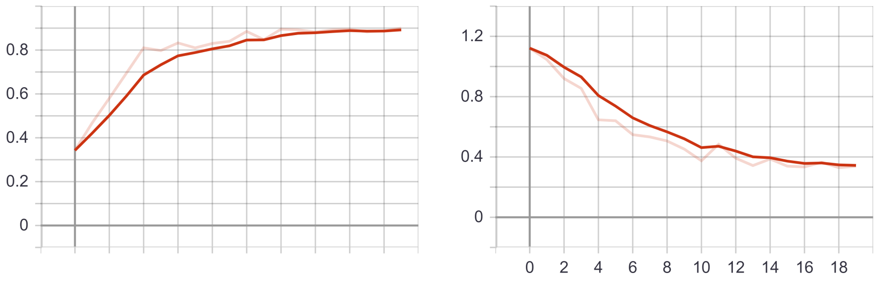
5.4.3 Actual training

After the optimizer is defined, the model is compiled.

Listing 8: Model compile function.

```
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=optimizer,
    metrics=['acc'])
```

`sparse_categorical_crossentropy` is used as a loss function, as recommended for multiclass categorization problems that are characterized by integers [Cho17, p.84]. It



(a) Accuracy

(b) Loss

Figure 8: The callbacks of the training algorithm provide logs that can be used by TensorBoard to provide useful graphics for evaluating the model. The vertical axis represents the value of each epoch (shown on the horizontal axis).

measures the distance between two probability distributions given by the last layer of the model by a `softmax` activation and the labelled data. The SoftMax activation is used to normalize the output by dividing each value by the sum of all values in each layer. This is useful because the labeled data assigns the value one to the correct node and zero to all others, making them numerically comparable and minimizing loss for a meaningful result.

The actual training then takes place using a `fit` function in the model. The training data is provided by generators; the function in this case is `fit_generator`. Therefore, the training is performed by calling the following function:

Listing 9: Initiating training of the model.

```
history = model.fit_generator(
    train_generator,
    steps_per_epoch=20,
    epochs=20,
    callbacks=callbacks)
```

Here it is defined that 20 images are fed into the model as a batch at each iteration, which is repeated for 20 epochs. After each epoch the optimizer updates the parameters (the θ of the model), which is happening 20 times in this run.

5.4.4 Results

As expected, the model starts with an accuracy of about 33% by trying to classify the training data into three classes. As the data suggests, the model quickly learns some relationships between the input data and the label, as it already has an accuracy of almost 70% after only four epochs.

Listing 10: Results of the first model (shorted).

```
Epoch 1/20
20/20 [=====...=====] - 2s 79ms/step - loss: 1.1220 - acc: 0.3425
Epoch 2/20
20/20 [=====...=====] - 1s 55ms/step - loss: 1.0457 - acc: 0.4675
Epoch 3/20
20/20 [=====...=====] - 1s 55ms/step - loss: 0.9199 - acc: 0.5800
Epoch 4/20
20/20 [=====...=====] - 1s 46ms/step - loss: 0.8548 - acc: 0.6950
...
Epoch 17/20
20/20 [=====...=====] - 1s 45ms/step - loss: 0.3340 - acc: 0.8950
Epoch 18/20
20/20 [=====...=====] - 1s 48ms/step - loss: 0.3648 - acc: 0.8825
Epoch 19/20
20/20 [=====...=====] - 1s 52ms/step - loss: 0.3291 - acc: 0.8875
Epoch 20/20
20/20 [=====...=====] - 1s 47ms/step - loss: 0.3388 - acc: 0.9000
```

After 20 epochs the model has an accuracy of about 90%, which is already a very good start for a first model with non-optimized hyperparameters. Important for a working model is how well the model classifies data it has never seen before. To verify this, a second generator is created: the `test_generator`, which is used to evaluate the model. By outputting `model.evaluate_generator(test_generator)` we get `[0.6071663084129493, 0.7866667]`, which indicates that the loss is `0.60`, as opposed to `0.33` for the training data, and has an accuracy of 78% on the data it has never seen before.

This result is quite remarkable, but obviously the model performs worse on new data, indicating that it overfitted on training data. Regardless, the result shows that by adjusting the hyper parameters, the result should be able to be increased to a satisfactory level.

The model is then saved at `models/symbol_classifier/first_model.h5` and can be loaded into Keras for further review.

6 Hyper parameter tuning

Unfortunately there is no obvious optimal model for every problem. A model maps the input to the output, but to reliably determine which information in the data should be discarded (and thus not contribute to generalization) and which should be emphasized, appropriate assumptions must be made.

In 1996 David Wolpert pointed out that there is no reason to prefer one model over another if no assumptions are made about the data set. This line of argument is now known as the *No Free Lunch Theorem* [Wol96]. Some of these assumptions are made in advance (for example, the use of feed-forward neural networks), but others are subject to optimization.

Hyperparameter optimization is an optimization problem in which the loss (or other metrics such as the accuracy) of the model for the given training data is to be minimized. The hyperparameters used in the first model, which could be the subject of optimization, are among others:

1. Training data:
   a) Number of training examples
   b) Image size
   c) Batch size
   d) Feature reduction
2. Model:
   a) Number of layers
   b) Layer type
   c) Layer size
   d) Loss function
3. Optimizer:
   a) Type of optimizer
   b) Learning rate
   c) Momentum
   d) Nesterov
4. Training:
   a) Steps per epoch
   b) Number of epochs

Some hyperparameters need to be tested and are not obvious, but others can be improved without obvious negative side effects, so they are discussed here first. Then, algorithms for testing different models with different hyperparameters to improve the model will be studied.

6.1 Data augmentation

The first thing that can be improved is the size of the dataset we use to train the model. In previous tests, the models saw each image about 20 times during training (batches of size 20, 20 steps per epoch, and 20 epochs divided by 400 training samples), which is most likely the cause of the overfitting that occurs. So increasing the size of the training batch seems to be a reasonable approach.

The augmentation of the data works in this case by changing some properties of the image that are useful for the purpose. In the case of symbol recognition, the symbol can be rotated or mirrored so adding training examples without adding much redundancy to the dataset and in this very case not invalidating the data.

The data extension is done by adapting the `create_generator` function previously used:

Listing 11: Data augmentation.

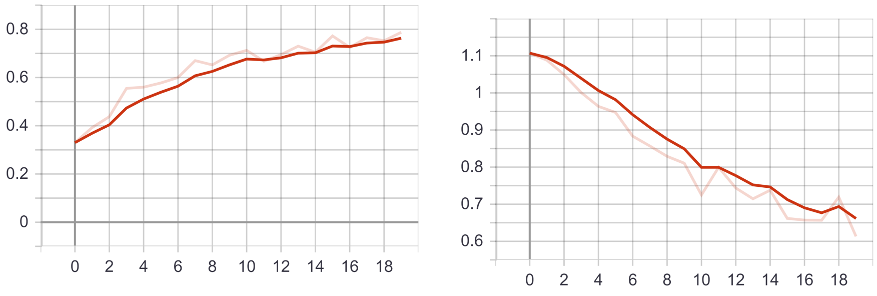
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

def create_generator(data_dir, batch_size, datagen):
    full_path = join(processed, data_dir)
    return datagen.flow_from_directory(
        full_path,
        target_size=(32, 32),
        batch_size=batch_size,
        class_mode='binary')

train_datagen = ImageDataGenerator(
        rescale = 1./255,
        rotation_range=360,
        horizontal_flip=True,
        vertical_flip=True)

test_datagen = ImageDataGenerator(rescale = 1./255)

train_generator = create_generator('train', 20, train_datagen)
test_generator = create_generator('test', 10, test_datagen)
```

Instead of `datagen = ImageDataGenerator(rescale = 1./255)` this object is specified as parameter, because the training set makes random changes to the loaded data. Namely a random rotation and the possibility to be mirrored horizontally and vertically. Please note that no changes are applied to the test data.

Applying these changes results in a loss of 0.61 and a 78.75% accuracy of the training data. The loss and accuracy of the test set is 0.63 and 78% respectively.

It is obvious that the accuracy of the training data actually decreases here. This is not surprising, as the model is no longer given data twice, i.e. there is no memorization, but as the test accuracy has not changed, the desired effect seems to have occurred. The model accuracy of the training set also seems to increase at a lower rate, so that more epochs or larger batches could already improve the performance.

Please note that the training set has practically grown by a factor of 1440 (360 degrees + rotations in two axes), which makes it possible to feed considerably more data into the model during training without having to expect too much overfitting.



(a) Accuracy

(b) Loss

Figure 9: Compared to figure 8, accuracy and loss increases more slowly. Further epochs may improve the results, as the model does not seem to have fully converged after 20 epochs.

The goal of reducing overfitting seems to be solved in this regard; the difference between training and test accuracy is negligible[19].

6.2 Feature reduction

Another thing that can improve the performance of the model is to reduce the size of the input. The phenomenon known as *curses of dimensionality* describes the possibilities of different configurations that are possible with an increasing number of variables, and is first discussed by Richard Bellman, who first refers to problems in dynamic programming [Bel57, p.ix].

The data is already greatly reduced by loading the data as images with a width and height of 32 pixels, although the data is raw as 512 by 512 images. This reduction has been done manually beforehand to keep the data relatively small, but the symbols are still clearly distinguishable.

Although the raw data contains only grayscale colors, the images are loaded with three color channels (hence the input size `[32, 32, 3]`). So one way to reduce the input size is to project the data onto a color dimension, as suggested by Géron [Gér19, p.215]. The call of the `flow_from_directory` is thus supplemented by a further parameter `color_mode='grayscale'` with the 3-dimensional `color_mode='rgb'` (which is set as default) to one dimension. Practically nothing should change, except the input size, which is adjusted to `[32, 32, 1]`.

[19] link to the respective notebook: `https://aka.klawr.de/srp#9`

The resulting model accordingly looks as follows:

Listing 12: Summary of improved model.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None , 1024)             0
_________________________________________________________________
dense (Dense)                (None , 32)               32800
_________________________________________________________________
dense_1 (Dense)              (None , 32)               1056
_________________________________________________________________
dense_2 (Dense)              (None , 3)                99
=================================================================
Total params: 33 ,955
Trainable params: 33 ,955
Non-trainable params: 0
_________________________________________________________________
```

The number of trainable parameters is almost divided by a factor of three, reducing the size of the model from 807kb (from the previous models with 99491 parameters) to 295kb. Nevertheless, no accuracy is lost[20].

6.3 Optimizers

So far only the optimizer of stochastic gradient descent has been considered. The stochastic gradient descent has several parameters that can be optimized, e.g. the learning rate, the momentum and whether Nesterov is used or not, as discussed in chapter 5.4.

Several attempts have already been made to automate the search for these hyperparameters. In this project some alternative optimizers are investigated, but not explained in too much detail. Please refer to the given citations for further insights.

With stochastic gradient descent, the learning rate is the same for all parameters of every layer in the model. An adjustment for each parameter or even for individual dimensions of the layers can be helpful, but to do this manually would require a disproportionate effort.

`AdaGrad` [DHS10] changes the learning rate for each parameter. This is done by adjusting the learning rate (starting with a uniform value) for each individual parameter using the size of the gradient for the respective parameter. While this approach works well for some problems, it does not make sense for some problems. Nevertheless, it serves as a baseline for many other optimizers developed after it, three of which have been tested in the tuning algorithm used in this project.

[20] Link to the respective notebook: `https://aka.klawr.de/srp#10`

AdaDelta [Zei12] is a modified algorithm based on AdaGrad that tries to solve AdaGrad's idea of constantly reducing the learning rate by dividing the learning rate again by an exponentially decreasing average (decay) of the gradients. This leads to a much more stable optimizer that promises to work well in practice.

RMSProp [Geo12] works similar to AdaDelta with a slightly different update rule. It is considered to be ”generally a good enough choice, whatever your problem”. [Cho17, p.77].

Adam [KB14][RKK18] is the last optimizer included in this project, which is another method to apply an adaptive learning rate for each parameter. Adam integrates the aforementioned idea of momentum into an algorithm similar to that of RMSProp.

There are other optimizers worth mentioning (namely AdaMax or Nadam), which may be the subject of further investigation.

It is remarkable that the adaptive methods explained here may not work with every model, so that it is always worth trying the basic gradient descent with nesterov[Gér19, p.358].

6.4 Convolutional layers

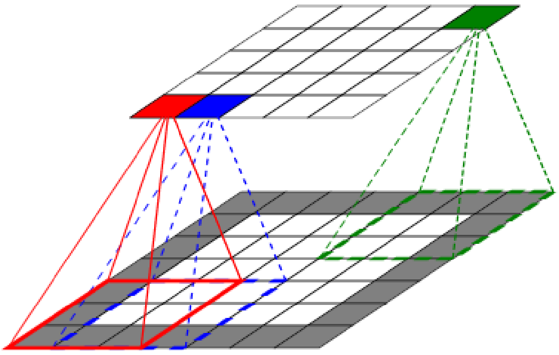
Convolutional layers are widely used for image recognition and it is therefore useful to integrate them into this model. They were introduced by Yann LeCun in 1998 in a work [Lec+98] to recognize handwritten numbers.

In this project only two-dimensional convolutional layers are used, because the input is two-dimensional (after the images have been transferred in grayscale). The difference between the convolutional layers and the previously described ”Dense Layers” is that the trained parameters do not represent the connections of all nodes of the previous layer to a next one, but a kernel that slides over the input layer and acts as a filter that builds up the output layer. This also means that the input layer does not need to be flattened before it is fed into the model.

When training a convolutional layer, the trained parameters are the filters that have a given shape. A two-dimensional convolutional layer with eight filters, an input depth of one and a kernel size of four times four (and one for the bias) has 136 (8 x (1 x 4 x 4 + 1)) parameters, which is significantly less than with previous approaches and does not depend on the size of the input layer, so that it is possible to recognize patterns in images of any size without having to enlarge the model itself.

The corresponding equation for a single activation is therefore given by [Gér19, p.

Figure 10: The bottom layer is the input, which is represented by a two-dimensional layer of size five by five (filter size is one), padded with a padding of one (drawn in gray) to ensure that the size remains the same when using a kernel of size three by three (highlighted by the red, green and blue rectangles). The stride used is one, i.e. the receptive field propagates one input node at each step. Here the process starts at the red rectangle, blue is the second step and green the last (25 steps in total).



453]:

$$z_{i,j,k} = b_k + \sum_{u=1}^{h} \sum_{v=1}^{w} \sum_{k'=1}^{n'} x_{i',j',k'} \times w_{u,v,k',k} \tag{21}$$

Where z is the activation value, the indices i and j represent the position of the node in the output layer and k the index of the filter. The sum of all input nodes is the sum of the nodes in the respective directions given by iteration over three axes using the height (h), width (w) and filter size (n') of the previous layer. For each of these combinations, the respective connection weight $w_{u,v,k',k}$ is multiplied by the input $x_{i',j',k'}$ at the respective position in the previous layer, where $i' = i \times s_h + u$ and $j' = j \times s_w$ respectively, and s_h and s_w are the steps taken at each step in each direction. The steps reduce the size of the following layer. Additionally, each filter has its own bias b_k, which is added to the activation function.

In order for the output to be the same size as the input, numbers must be added to the input, since a kernel that is greater than one in any direction cannot cross an edge for the input (and the step must be 1 by 1). So padding is used to solve this problem by (typically) adding zeros around the input, resulting in a technique called zero padding.

By using convolutional layers instead of dense layers, the model size is further reduced (295kb to 232kb) and accuracy is increased by a few percent. Adding pooling reduces the size further to 149kb and further improves the accuracy[21]. Pooling is a technique where a kernel (similar to the core of Convolutional Layers) scans over the input layer. In the linked notebook, a kernel was used in pairs, taking the maximum value of the kernel and passing only the maximum value, thus halving the output size in width and height[22].

6.5 Tuning algorithms

Taking into account all possibilities to optimize the model before training, some approaches have been developed to automatically optimize the hyperparameters. One way of setting the hyperparameters is to test each one with different values, but freezing the values of all other hyperparameters. This is then repeated for all hyperparameters. This procedure ensures that the optimal set of hyperparameters is used, but can take an unacceptable amount of time, since the number of models created grows exponentially with the number of hyperparameters to be adjusted. With two hyperparameters, this would mean checking each value on a two-dimensional grid, so the method is called ”grid search”[23].

Another approach frequently used in practice is the random search. The random search works on the principle of randomly assigning hyperparameters and thus testing different models without any advantage of a combination. Of course, this (most likely) does not lead to an optimal result, but in practice this approach is often used because it usually gives an acceptable result by testing a wide range of possible configurations. A notebook was created to show the implementation of `keras tuner` in this project at `https://aka.klawr.de/srp#13`.

Some more sophisticated methods are based on the random search approach, in which the results of the random search are repeatedly examined and then the hyperparameters are adjusted in favor of those that promise to yield better models. This approach is called zooming, because after each iteration you zoom into the most promising range of hyperparameters.

Other approaches try to completely automate this process. Popular algorithms are based on Bayesian optimization, but these are not covered in this project and are therefore omitted.

The tuner used in this project is called `Hyperband` [Li+18], which is implemented in the library `keras-tuner`. [Goo19a]. Essentially, `Hyperband` trains models using random

[21] Link to the respective notebooks: `https://aka.klawr.de/srp#11` and `https://aka.klawr.de/srp#12`
[22] This is appropriately called `MaxPooling`.
[23] If only one hyperparameter is adjusted, the method is called ”linear search”.

searches, but restricts training to a wide range of hyperparameters in a few epochs. This is done by training the models for a few epochs and keeping only the most promising models for further iterations. This method promises a much better use of resources and will most likely be able to present good hyperparameters.

The final model tuner is designed to accept a Hyperparameter object that is supplied to a function that returns the respective model:

Listing 13: Function to find optimized model.

```
def create_model(hp):
    model = models.Sequential()
    model.add(layers.Conv2D(2**hp.Int('2**num_filter_0', 4, 6),
        (4,4) ,activation='relu', input_shape=(32, 32, 1)))

    for i in range(hp.Int('num_cnn_layers', 0, 3)):
        filter = 2**hp.Int('2**num_filter_' + str(i), 4, 7)
        model.add(
            layers.Conv2D(filter, (4,4), activation='relu',padding='same'))
        if hp.Boolean('pooling_' + str(i)):
            model.add(layers.MaxPooling2D(2, 2))

    model.add(layers.Flatten())
    for i in range(hp.Int('num_dense_layers', 1, 3)):
        nodes = 2**hp.Int('2**num_nodes_' + str(i), 4, 7)
        model.add(layers.Dense(nodes, activation='relu'))

    model.add(layers.Dense(3, 'softmax'))

    optimizers = {
        'adam': Adam(),
        'sgd': SGD(lr=hp.Choice(
            'learning_rate', [0.001, 0.003, 0.007, 0.01 0.03]),
            momentum=hp.Float('momentum', 0.6, 1, 0.1),
            nesterov=hp.Boolean('nesterov')),
        'rms': RMSprop(lr=hp.Choice(
            'learning_rate', [0.001, 0.003, 0.007,0.01, 0.03]))
    }

    model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=optimizers[hp.Choice('optimizer', list(optimizers.keys()))],
        metrics=['acc'])

    return model
```

This function extends 5 by implementing hyperparameters using hyperparameter of `keras tuner`. This will be implemented later using a custom class[24]:

[24] The results are logged with `TensorBoard`, whose API is not yet supported by `keras tuner` itself. So the tuner is inherited from a custom class

Listing 14: Definition of model tuner.

```
tuner = customTuner(
    create_model,
    hyperparameters=hp,
    objective='acc',
    max_trials=100,
    executions_per_trial=1,
    directory=log_dir,
    project_name=timestamp)
```

The training that was previously performed by the `fit` function is now performed by the `search` function of the tuner:

Listing 15: Searching the best model.

```
tuner.search(
    train_dataset,
    validation_data=validation_dataset,
    epochs=30,
    steps_per_epoch=100,
    validation_steps=100,
    verbose=0,
    callbacks=callbacks)
```

Another note to take here is that the generators used in 6.1 have been replaced by custom functions to augment data. The data is then preprocessed using protocol buffers(`protobufs`)[25]. By pre-processing the image data in advance, the loading time of the data is significantly reduced. The code used for this can be viewed here: `https://aka.klawr.de/srp#15`.

After checking the data, an accuracy of 96.7% on the test data was achieved with a model size of 4.3mb.

6.6 Increasing the training set

After the model was adapted, another approach was tried to increase the accuracy. The raw training data was more than tripled (to 1000 examples for each symbol in the training data) and then expanded with 32 repetitions, resulting in a data set of 96000 different images.

Using this data set, almost every model tested previously achieved an accuracy of over 99%. This situation presents the minimization of the model as the most important optimization goal. The final model is manually reduced and tuned by reducing layers and hyper parameters. The final model has a size of 207kb with an accuracy of 99.3% and a loss of 0.04 on the training set, which is satisfactory.

[25] The final notebook using these can be found at `https://aka.klawr.de/srp#14`

The model is summarized as:

Listing 16: Definition of final model.

```
model = models.Sequential()
model.add(layers.Conv2D(16, (4,4), activation='relu', padding='same',
    input_shape=(32, 32, 1)))
model.add(layers.MaxPooling2D(2,2))
model.add(layers.Conv2D(32, (4,4), activation='relu', padding='same'))
model.add(layers.MaxPooling2D(2,2))
model.add(layers.Flatten())
model.add(layers.Dense(3, 'softmax'))

optimizer = Adam()
model.compile(loss='sparse_categorical_crossentropy',
    optimizer=optimizer,metrics=['acc'])
```

The code for the model creation can be viewed at `https://aka.klawr.de/srp#16`.

7 Conclusion

After investigating the possibilities of machine learning and its impact on the different branches of technology, the idea of extending this to the field of kinematics seems reasonable. In this project the basics for the application of machine learning were learned from scratch, starting with a simple statistical model capable of classifying handwritten mechanical symbols.

The data suggest that a sufficient algorithm has been found that has an accuracy of over 99% over the test data. After removing the training configuration, it has a size of 76kb as `Keras` model. Converting the model into a format readable by `tensorflow.js`[26] and assembling it into a readable `IOHandler`[27] does not significantly increase the size (81kb).

Applications for testing the models were made available in the repository:

A Python script for testing purposes to draw images and the model predicting the drawn symbol, is located in the `code` directory as `symbol_classifier_test.py`[28].

A web page that can be loaded directly into a web browser[29]. The file is located in the same directory as the Python script as `symbol_classifier_test.html`[30]. This test

[26] `https://www.tensorflow.org/js/`

[27] The loader.js can be found at `https://aka.klawr.de/srp#15`

[28] `https://aka.klawr.de/srp#16`

[29] The test works on Firefox and Google Chrome. Microsoft Edge does not yet work, but this will probably change when Microsoft moves to a Chromium-based build.

[30] `https://aka.klawr.de/srp#17`

comes close to its intended purpose by uploading images into the application and then providing automatic predictions to the user.

Later projects will follow to provide functionality for detecting the position of recognized symbols in a given image. Furthermore, possible connections of the symbols to each other in the form of ”constraints” will be recognized. After complete mechanism representations can be derived, they can be imported into existing web applications like `mec2` [Gös19c; Gös19b; Gös19a] or `mecEdit`. [Uhl19a; Uhl19b].

List of Figures

Listings

Link Index - `https://aka.klawr.de/srp`

References

[BB01] Michele Banko and Eric Brill. “Scaling to Very Very Large Corpora for Natural Language Disambiguation”. In: *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*. ACL ’01. Toulouse, France: Association for Computational Linguistics, 2001, pp. 26–33. DOI: 10.3115/1073012.1073017. URL: https://doi.org/10.3115/1073012.1073017.

[Bel57] Richard E. Bellman. *Dynamic Programming*. PRINCETON UNIV PR, Oct. 11, 1957. 342 pp. ISBN: 069107951X. URL: https://www.ebook.de/de/product/34448612/richard_e_bellman_dynamic_programming.html.

[Cho17] Francois Chollet. *Deep Learning with Python*. Manning Publications, Oct. 28, 2017. 384 pp. ISBN: 1617294438. URL: https://www.ebook.de/de/product/28930398/francois_chollet_deep_learning_with_python.html.

[Cho19] François Chollet. *Keras*. 2019. URL: https://keras.io/ (visited on 10/09/2019).

[Cyb89] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[DHS10] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research 12* (2010). URL: `http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf`.

[dri19] drivendata. *Cookiecutter Data Science.* 2019. URL: `https://drivendata.github.io/cookiecutter-data-science/` (visited on 10/23/2019).

[GBB11] Xavier Glorot, Antoine Bordes, and Y. Bengio. “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the 14th International Conference on Artificial Intelligence and Statisitics (AISTATS) 2011* 15 (Jan. 2011), pp. 315–323.

[GBC17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* The MIT Press, Jan. 3, 2017. 800 pp. ISBN: 0262035618. URL: `https://www.ebook.de/de/product/26337726/ian_goodfellow_yoshua_bengio_aaron_courville_deep_learning.html`.

[Geo12] Kevin Swersky Geoffrey Hinton Nitish Srivastava. “Neural Networks for Machine Learning”. 2012. URL: `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[Gér19] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow.* O’Reilly UK Ltd., Oct. 1, 2019. 819 pp. ISBN: 1492032646. URL: `https://www.ebook.de/de/product/33315532/aurelien_geron_hands_on_machine_learning_with_scikit_learn_keras_and_tensorflow.html`.

[Goo19a] Google. *Keras Tuner.* 2019. URL: `https://keras-team.github.io/keras-tuner/` (visited on 11/28/2019).

[Goo19b] Google. *Tensorflow.* 2019. URL: `https://www.tensorflow.org/` (visited on 10/09/2019).

[Goo65] Irvin John Good. *Speculations Concerning the First Ultraintelligent Machine.* 1965. URL: `http://acikistihbarat.com/dosyalar/artificial-intelligence-first-paper-on-intelligence-explosion-by-good-1964-acikistihbarat.pdf` (visited on 10/16/2019).

[Gös19a] Stefan Gössner. “Ebene Mechanismenmodelle als Partikelsysteme - ein neuer Ansatz”. In: *13. Kolloquium-Getriebetechnik - Tagungsband.* Ed. by Burkhard Corves, Philippe Wenger, and Mathias Hüsing. Cham: Springer International Publishing, 2019, pp. 169–180. ISBN: 978-3-8325-4979-4.

[Gös19b] Stefan Gössner. “Fundamentals for Web-Based Analysis and Simulation of Planar Mechanisms”. In: *EuCoMeS 2018*. Ed. by Burkhard Corves, Philippe Wenger, and Mathias Hüsing. Cham: Springer International Publishing, 2019, pp. 215–222. ISBN: 978-3-319-98020-1.

[Gös19c] Stefan Gössner. *mec2*. 2019. URL: `https://github.com/goessner/mec2`.

[Hei19] Olli-Pekka Heinisuo. *opencv-python*. 2019. URL: `https://github.com/skvark/opencv-python` (visited on 10/22/2019).

[HNP09] A. Halevy, P. Norvig, and F. Pereira. “The Unreasonable Effectiveness of Data”. In: *IEEE Intelligent Systems* 24.2 (Mar. 2009), pp. 8–12. DOI: `10.1109/MIS.2009.36`.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (Jan. 1989), pp. 359–366. DOI: `10.1016/0893-6080(89)90020-8`.

[Jup19] Jupyter. *Jupyter Notebook*. 2019. URL: `https://jupyter.org/` (visited on 10/23/2019).

[KB14] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: `1412.6980 [cs.LG]`.

[Lec+98] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: `10.1109/5.726791`.

[Li+18] Lisha Li et al. “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization”. In: *Journal of Machine Learning Research* 18.185 (2018), pp. 1–52. URL: `http://jmlr.org/papers/v18/16-558.html`.

[McC55] John McCarthy. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. 1955. URL: `http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html` (visited on 10/09/2019).

[MP43] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: `10.1007/BF02478259`. URL: `https://doi.org/10.1007/BF02478259`.

[Nes83] Y. E. Nesterov. “A method for solving the convex programming problem with convergence rate $O(1/k^2)$”. In: *Dokl. Akad. Nauk SSSR* 269 (1983), pp. 543–547. URL: `https://ci.nii.ac.jp/naid/10029946121/en/`.

[New87] Isaac Newton. *Philosophiae naturalis principia mathematica.* Jussu Societatis Regiae ac Typis Josephi Streater. Prostat apud plures bibliopolas, 1687. DOI: 10.5479/sil.52126.39088015628399.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning.* 2015. URL: http://neuralnetworksanddeeplearning.com (visited on 10/17/2019).

[nvi19] nvidia. *CUDA*. 2019. URL: https://www.geforce.com/hardware/technology/cuda (visited on 10/09/2019).

[Ope19] OpenCV. *OpenCV.* 2019. URL: https://opencv.org/ (visited on 10/22/2019).

[Pol64] Boris Polyak. "Some methods of speeding up the convergence of iteration methods". In: *Ussr Computational Mathematics and Mathematical Physics* 4 (Dec. 1964), pp. 1–17. DOI: 10.1016/0041-5553(64)90137-5.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature, Volume 323, Issue 6088, pp. 533-536 (1986)* 323 (Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0.

[RKK18] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. "On the Convergence of Adam and Beyond". In: *International Conference on Learning Representations.* 2018. URL: https://openreview.net/forum?id=ryQu7f-RZ.

[RT17] David Rolnick and Max Tegmark. "The power of deeper networks for expressing natural functions". In: *Neural Networks* (May 16, 2017). arXiv: http://arxiv.org/abs/1705.05502v2 [cs.LG].

[Stu18] Peter Norvig Stuart Russell. *Artificial Intelligence: A Modern Approach, Global Edition.* Addison Wesley, Nov. 28, 2018. ISBN: 1292153962. URL: https://www.ebook.de/de/product/25939961/stuart_russell_peter_norvig_artificial_intelligence_a_modern_approach_global_edition.html.

[Sut+13] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning.* Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 2013, pp. 1139–1147. URL: http://proceedings.mlr.press/v28/sutskever13.html.

[Uhl19a] Jan Uhlig. "Entwicklung einer modularen Web-App zur interaktiven Modellierung und impulsbasierten Analyse beliebiger planarer Koppelmechanismen". MA thesis. Fachhochschule Dortmund, 2019.

[Uhl19b] Jan Uhlig. *mecEdit*. 2019. URL: `https://mecedit.com` (visited on 12/11/2019).

[Wol96] David H. Wolpert. "The Lack of A Priori Distinctions Between Learning Algorithms". In: *Neural Computation* 8.7 (1996), pp. 1341–1390. DOI: `10.1162/neco.1996.8.7.1341`. eprint: `https://doi.org/10.1162/neco.1996.8.7.1341`. URL: `https://doi.org/10.1162/neco.1996.8.7.1341`.

[Zei12] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). arXiv: `1212.5701`. URL: `http://arxiv.org/abs/1212.5701`.