University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Modelling Mouse Movements With Machine Learning

Anith Manu Ravindran

April 6, 2020

Abstract

In order to better understand human mouse dynamics, this thesis presents a modeling of human mouse motion using neural networks. A trained neural network generates human-like horizontal movements step by step. I use feed-forward neural networks and recurrent neural networks. Input values for the network are acceleration, speed, distance to target and width of target. This generates the acceleration for the next step as output. I explain the choice of network design and illustrate result movements with different plots. I compare the movements of the networks with each other and with human ones.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.0.1 Motivation

The mouse is one of the biggest switches between humans and computers. Many PCs are operated and controlled with the computer mouse. Still, human mouse dynamics are not fully understood. A human-like mouse movement model may help comprehending this topic. This in turn could help to develop a more effective mouse design or adapt tasks for the mouse. Nevertheless, the modeling of human mouse dynamics has not yet been intensively researched. Therefore, this work is intended to close part of this gap and to show another method for realistic modeling of human mouse dynamics. I have chosen machine learning with neural networks as the modeling method. Machine learning is a widespread tool worldwide. It is used successfully in more and more areas today to simulate human behavior that is too complex for conventional algorithms. One of the most well known types of machine learning is the training of artificial neural networks. The way the nets learn to move could allow conclusions to be drawn about human movements. Hence, I have chosen this specific method for this thesis.

1.0.2 Topic

Therefore, the topic of this thesis is the modeling of mouse movements using neural networks. As a basis I use a dataset containing details of horizontal human movements. The goal is to train neural networks in such a way that they can produce human-like horizontal movements. The computer-generated mouse movement should be as closely resembling as possible to a characteristic human movement.

1.0.3 Structure of the thesis

The rest of the thesis is structured as follows. First, I outline neural networks in general and feed-forward networks in particular. Their structure and functionality is explained. Then I will introduce and describe recurrent neural networks. On this basis I present the used

data set and show the tasks for networks as well as their training. Subsequently I evaluate the gathered results. Afterwards, I discuss limitations and present related work. Finally, I describe future work before concluding the work.

Chapter 2

Neural networks

In this chapter, I provide the technical background required to understand the functionality of neural networks. I explain their structure and inner composition. Then I show how the networks learn.

2.1 Neurons

The functioning of neural networks is characterized by the work of the individual neurons. Therefore, I introduce them now.

2.1.1 Biological neurons

Artficial neuronal networks are inspired by their biological counterparts. Hence, I briefly present them first. To put it simply, biological neurons consist of dendrites, cell body and axon, as shown in Figure 2.1. "The dendrites receive signals from other neurons and pass them over to the cell body." [6, p.4] Dendrites and cell body constitute the input surface of the neuron. The signal then travels through the cell body to the axon. The axon transmits it through the synapse to the dendrites of bordering neurons. The same process happens in these neurons then. The amount of the forwarded signal depends on its intensity, the strength of the connection between the communicating neurons and the threshold of the receiving neuron. A neuron has a large number of dendrites and synapses. So it can receive and send many signals at the same time. These signals can support or interfere each other [6, 5, p.4f].

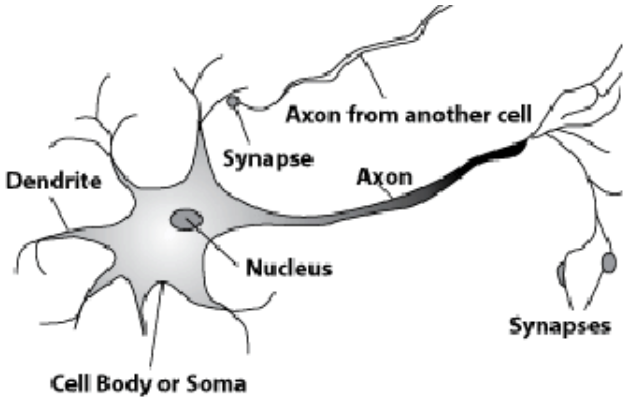


Figure 2.1: Biological Neuron with dendrites, cell body and axon [4, p.3]

2.1.2 Artificial neurons

Artificial neurons function in a similar manner. The dendrites and axons are imitated by connections between individual neurons. Connection weights imitate the synapses. The threshold of a neuron approximates the activity in the cell body. Also, both types learn in the same way. They incrementally strengthen or weaken the weights or synapses. [6, p.5f]

An artificial neuron j receives the inputs $x_1, ..., x_n$. Then it processes them with the weights of the neuron w_{ij} to form a net input ξ, as shown in Equation 2.1. Positive weighted connections reinforce a signal, while negative weights inhibit neuron activity [6, p.16]. The bias b_j represents the bias of a neuron. It is added to the net input, as it is considered to be the weight of an additional edge [23, p.165].

$$\xi_j = b_j + \sum_{i=1}^{N} x_i w_{ij} \tag{2.1}$$

ξ_j is the input for the activation function φ. φ calculates the corresponding activity y_j of the neuron j, as can be seen in Equation 2.2. The whole input processing procedure is outlined in Figure 2.2 [4, p.3].

$$y_j = \varphi(\xi_j) \tag{2.2}$$

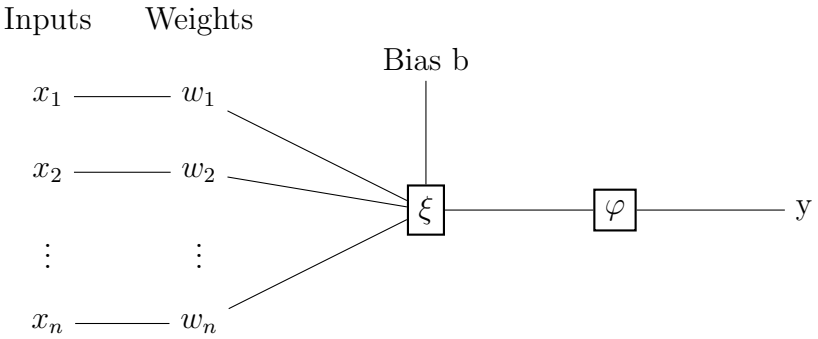


Figure 2.2: Input processing in an artificial neuron

2.1.3 Activation functions

Activation functions assign a corresponding output to a given input. Many different functions are available for this purpose. Among them are for example sigma σ or the hyperbolic tangent *tanh*. A variation of σ is σ_{hard}. Their respective function is shown in the following equations. Also, their plots are shown in Figure 2.3 [18, p.7].

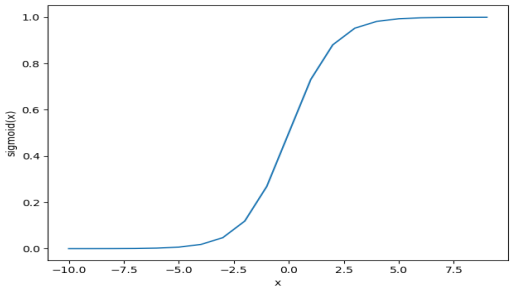
$$Sigmoid : \sigma(x) = \frac{1}{(1 + e^x)} \tag{2.3}$$

$$Hard - Sigmoid : \sigma_{hard}(x) = \begin{cases} 0, & \text{if } x < -2.5 \\ (0.2 * x) + 0.5, & \text{if } -2.5 <= x <= 2.5 \\ 1, & \text{if } x > 2.5 \end{cases} \tag{2.4}$$

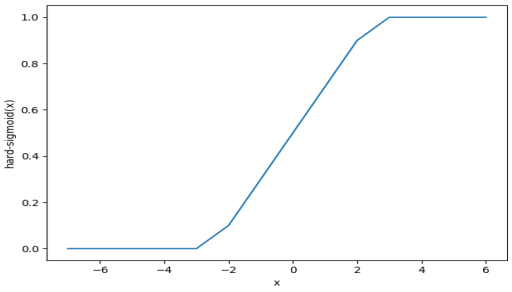
$$\tanh(x) = \frac{e^x - e^(-x)}{e^x + e^(-x)} \tag{2.5}$$

In this thesis, I use the hard sigmoid from Equation 2.4 and *tanh* from Equation 2.5 as activation functions for recurrent networks. For feed-forward neural networks (FFNNs) I use the rectifier linear unit (ReLU), because it improves the performance for many deep neural networks [18, p.7]. ReLU is the most widely-used activation function for deep learning models [22, p.1]. It is defined as just returning the parameter in case it is positive or zero if the parameter is negative [12] as can be seen in Figure 2.3d or Equation 2.6.
For the output layer, all models in this thesis use a linear function $\varphi(x) = x$ as activation function.

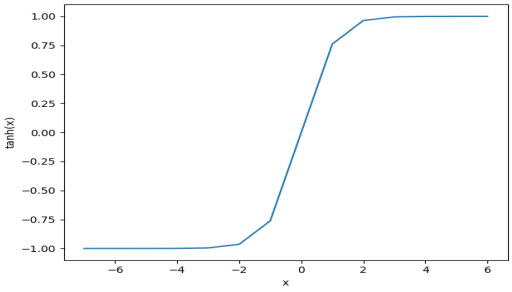
$$ReLU : \varphi(x) = max(0, x) \tag{2.6}$$



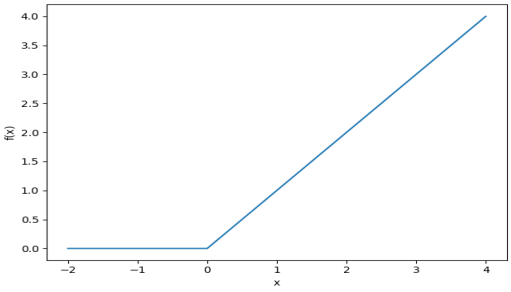
(a) Sigmoid $\sigma(x)$



(b) Hard-sigmoid $\sigma_{hard}(x)$



(c) Tanh $\tanh(x)$



(d) ReLU $\varphi(x)$

Figure 2.3: Plots of different activation functions

2.2 Artificial neural networks

An artificial neural network (ANN) consists of many of the neurons presented above. They are arranged into layers. The first layer is called the input layer and the last one the output layer. Layers in between are called hidden layers. Signals are sent from the input layer to the output layer through the hidden layers. At each layer, the signal can be processed and modified. A model that work this way is called FFNN or multilayer perceptron (MLP). Not all neural networks are MLPs. For some, output of neurons is fed back to the same layer or previous layers [4, p.3].

The MLPs in this thesis are fully-connected neural networks because each neuron is connected to all neurons in the adjacent layers. An example network can be seen in Figure 2.4.

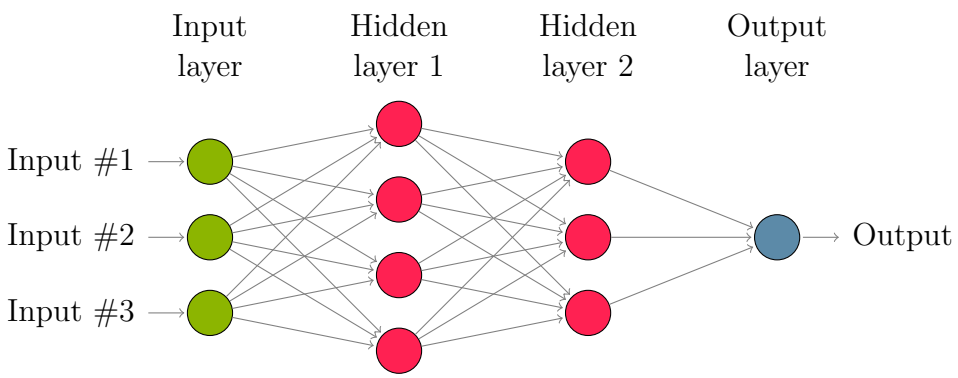


Figure 2.4: Example of a fully connected MLP with two hidden layers

2.2.1 Backpropagation

Backpropagation is a popular learning algorithm for neural networks [17]. Backpropagation calculates the derivatives of the loss function $\mathcal{L}$ corresponding to the various weights of the network. These are then adapted according to the gradient descent. The procedure is as follows [23, p.166].

(i) Compute feed-forward propagation

(ii) Compute backpropagation error of output layers

(iii) Iterate backwards through the network and calculate backpropagation errors

(iv) Update weights

First, an output is calculated forward through the network. In this process each node calculates an output y_j. Both y_j and ξ_j, that is required to get y_j, are stored. The nodes of the output layer produce an output $\hat{y_k}$ for every node k of the output layer. Subsequently, a loss function value $\mathcal{L}(\hat{y}, y_{true})$ is calculated for every node k in the output layer. Afterwards the backpropagated error δ_k is calculated for each output node k [18, p.9]:

$$\delta_k = \frac{\partial \mathcal{L}(\hat{y_k}, y_k)}{\partial \hat{y_k}} * \varphi_k`(\xi_k) \tag{2.7}$$

These values are used to calculate the δ_j values of all nodes in the previous layer. Other layers are calculated in this manner. This is done layer by layer from back to front. w_{jk} describes the weight of the connection from node j to node k [18, p.9f].

$$\delta_j = \varphi`(\xi_j) \sum_k \delta_k * w_{jk} \tag{2.8}$$

As ξ_j is calculated and stored during the forward pass, δ_j during the backward pass. Hence, $\partial \mathcal{L}$ can be calculated according to a respective weight w_{ij} this way [18, p.9f]:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \delta_j * y_i \tag{2.9}$$

After the backpropagated error has been calculated throughout the network, the weights are updated in the negative gradient direction. The learning constant γ sets the step length of the correction. The bias is treated and adjusted like the normal weights. Weight corrections are obtained by the means of following formula [23, p.169]:

$$\Delta w_{ij} = -\gamma y_i \delta_j \tag{2.10}$$

The backtracking algorithm allows the network to adjust its weights with a given input and correct output. This is how learning works at the level of detail. Now I show the different generalized learning methods.

2.2.2 Training

There are three fundamental ways to train an ANN. The network can learn supervised, unsupervised or reinforced. With supervised training, both training data and the associated solution are transferred to the network. The network then adapts its weights to these given data. In this thesis, all models are trained supervised. With unsupervised training, the network only receives the training data. It then tries to determine underlying patterns of this data. In reinforcement learning, the network receives neither input nor output data. Instead, a function determines how good or bad an output value of the network is. The network is then rewarded or punished accordingly [19, p.13ff].
In this chapter I introduced the structure and functionality of neural networks (NNs) in general and MLPs in particular. But there are also other types of NNs.

Chapter 3

Recurrent neural networks

In this chapter I present recurrent neural networks as an alternative to MLPs. I show the general principle as well as the technical details of the most used construction.

3.1 From feed-forward to recurrent

I start with the differences to the MLPs, and then I get to the functionality of the features of the recurrent neural networks (RNNs). I illustrate these with a simple example.

3.1.1 Recurrent edge

Recurrent neural networks are an extension of conventional feedforward neural networks. Unlike MLPs, the signal does not simply flow from the input layer through the hidden layers to the output layer. Rather, the output of a recurrent neuron can be fed back to the same neuron or to neurons in previous layers. Hence, the signal can flow both forward and backward. Thus, as a feature, RNNs can remember already processed inputs. Thereby, they use both current and former inputs for making a decision. An example for a simple RNN can be seen in Figure 3.1 on page 11 [6, p.14].

3.1.2 Simple recurrent network

For a simple recurrent network with only one hidden layer as in Figure 3.1 the values can be computed in the following manner. At time t, the hidden layer vector $h^{(t)}$ is calculated with current data point $x^{(t)}$ and hidden node values from the previous state $h^{(t-1)}$. W^{hx} is the matrix of conventional weights between the input and the hidden layer. W^{hh} is the matrix of recurrent weights between the hidden layer at adjacent time steps. b represents the bias [18, p.10f].

$$h^{(t)} = \varphi(W^{xh}x^{(t)} + W^{hh}h^{(t-1)} + b_h) \tag{3.1}$$

The output $\hat{y^{(t)}}$ can be obtained with the hidden node values of the current time step $h^{(t)}$ [18, p.10f].

$$\hat{y}^{(t)} = \varphi(W^{hy}h^{(t)} + b_y) \tag{3.2}$$

This process of expanding the recurrent network of figure 3.1 along the time steps can be seen in figure 3.2. The same principle is also used to complement the backpropagation algorithm. This extended version is called backpropagation through time (BPTT). BPTT was introduced by Werbos in 1990 [26].

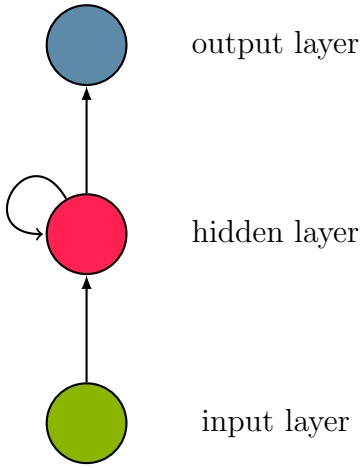


Figure 3.1: A simple recurrent network with one hidden layer and one node per layer.

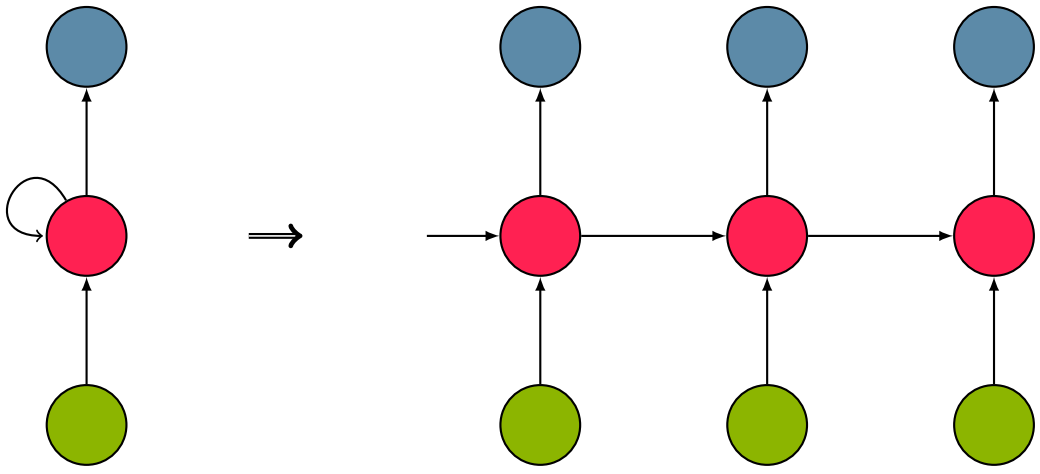


Figure 3.2: Unfolding of a simple recurrent network along the time steps

3.2 LSTM

One of the most well known RNN architectures is the long short-term memory model (LSTM). The LSTM was introduced by Hochreiter and Schmidhuber in 1997 [13].

The name can be explained as follows: Simple recurrent nets have a long-term memory due to their weights. Additionally, they have short-term memory in the form of ephemeral activations, which pass from each node to successive nodes. The LSTM introduces an intermediate type of storage via the memory cell [18, p.17].

3.2.1 Memory cell

The hidden nodes are replaced with memory cells in a LSTM layer or network. The composition of a memory cell c is described as follows. The input node g_c is activated through the input of the current or the previous time step. The input node is followed by the input gate i_c. i_c can increase or decrease the value of a node. In the case of $i_c = 0$, the value of the node is completely ignored. The internal state s_c is a linear activated node. It is connected to itself by a recurrent edge with a fixed weight. As the weight is fixed across time steps, error can flow without vanishing or exploding [18, p.18].

$$s^{(t)} = g^{(t)} * i^{(t)} + s^{(t-1)} \tag{3.3}$$

In 2000, the forget gate f_c was introduced by Gers et al. [10]. This allows the network to learn to change the internal state. That changes the equation for updates of the internal state to the following [18, p.18].

$$s^{(t)} = g^{(t)} * i^{(t)} + f^{(t)} * s^{(t-1)} \tag{3.4}$$

Afterwards, the value of the internal state is multiplied with the output gate o_c [18, p.18].

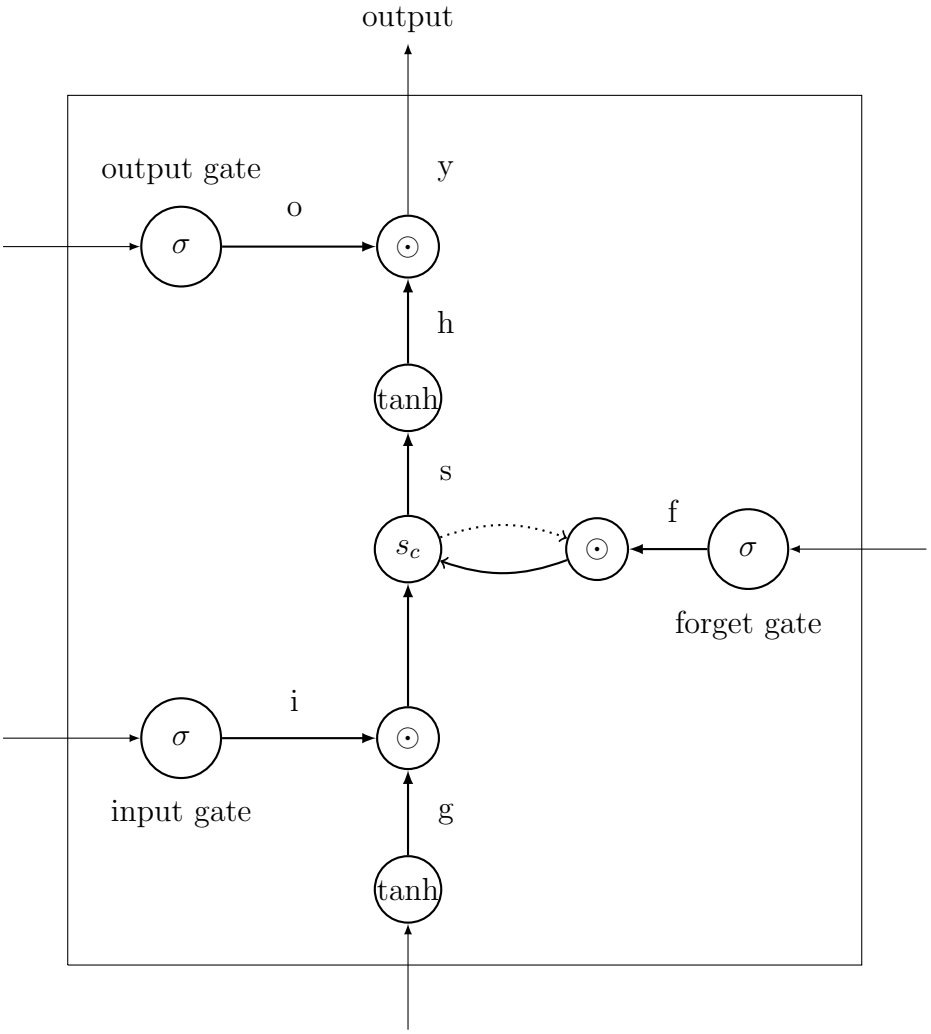


Figure 3.3: Memory cell of a LSTM with forget gate. Dotted line represents a recurrent edge, normal lines within in the cell are conventional edges. All inputs from outside the memory cell are both recurrent and conventional edges. Based on the work of Greff et al. [11, p.2]

.

3.2.2 Forward pass

The inner function of the memory cell can be summarized by these equations [18, p.20].

$$g^{(t)} = \tanh(W^{xg}x^{(t)} + W^{hg}h^{t-1} + b_g) \tag{3.5}$$

$$i^{(t)} = \sigma(W^{xi}x^{(t)} + W^{hi}h^{t-1} + b_i) \tag{3.6}$$

$$f^{(t)} = \sigma(W^{xf}x^{(t)} + W^{hf}h^{t-1} + b_f) \tag{3.7}$$

$$o^{(t)} = \sigma(W^{xo}x^{(t)} + W^{ho}h^{t-1} + b_o) \tag{3.8}$$

$$s^{(t)} = g^{(t)} \odot i^{(t)} + s^{(t-1)} \odot f^{(t)} \tag{3.9}$$

$$h^{(t)} = \tanh(s^{(t)}) \odot o^{(t)} \tag{3.10}$$

In these equations *tanh* from Equation 2.5 is used as an activation function for non-recurrent edges. Similarly, the σ_{hard} function from Equation 2.4 is used for recurrent steps in a LSTM. Both functions are displayed in Figure 2.3 on page 7.

3.2.3 BPTT in a LSTM

The BPTT algorithm can be be calculated with these equations. $\Delta^{(t)}$ is the vector of deltas passed down from the layer above [11, p.2].

$$\delta_y^{(t)} = \Delta^{(t)} + W^{hg}\delta_g^{t+1} + W^{hi}\delta_i^{t+1} + W^{hf}\delta_f^{t+1} + W^{ho}\delta_o^{t+1} \tag{3.11}$$

$$\delta_o^{(t)} = \delta_y^{(t)} \odot \tanh(c^{(t)}) \odot \sigma`(o^{(t)}) \tag{3.12}$$

$$\delta_s = \delta_y^{(t)} \odot o^{(t)} \odot \tanh`(s^{(t)}) \tag{3.13}$$

$$\delta_f^{(t)} = \delta_c^{(t)} \odot c^{(t-1)} \odot \sigma`(f^{(t)}) \tag{3.14}$$

$$\delta_i^{(t)} = \delta_c^{(t)} \odot g^{(t)} \odot \sigma`(i^{(t)} \tag{3.15}$$

$$\delta_g^{(t)} = \delta_c^{(t)} \odot i^{(t)} \odot \tanh`(g^{(t)}) \tag{3.16}$$

If there is a layer below that also needs training, the deltas for the inputs need to be computed as follows:

$$\delta_x^{(t)} = W^{xg}\delta_g^{(t)} + +W^{xi}\delta_i^t + W^{xf}\delta_f^t + W^{xo}\delta_o^t \tag{3.17}$$

Lastly, the gradients for the weights are computed with these equations with $\star \epsilon g, i, f, o$.

$$\delta W_\star = \sum_{t=0}^{T} (\delta_\star^t \otimes x^{(t)}) \tag{3.18}$$

Same for the recurrent weights:

$$\delta W_\star = \sum_{t=0}^{T-1} (\delta_\star^{t+1} \otimes y^{(t)}) \tag{3.19}$$

$$\delta b_\star = \sum_{t=0}^{T} \delta_\star^t \tag{3.20}$$

With the knowledge of how FFNNs and RNNs work and learn, I show in the next chapters how they can be used for the problem of this thesis.

Chapter 4

Realization

After the technical background I now explain the structure and training of the various used neural networks. I also discuss the reasons for the selected design of the models.

4.1 Used Tools

For this purpose I first present the tools and data used in this thesis.

4.1.1 Dataset

The publicly available "Pointing Dynamics Dataset: Processed Data" [20] offers measurement data for horizontal mouse movements. The dataset contains the data of experiments involving 12 persons, of whom 11 are male and one female. [21, p. 12]
Their task is specified as follows:

> "Two one-dimensional targets are displayed. The x-dimension of the mouse is used to move a white crosshair pointer (1 px wide) that only moves horizontally. The task is to click on the targets serially. The current target is shown in red, while the other target is shown in grey. A new trial starts as soon as the participant clicks on the previous target. Missed trials are annotated in the dataset. Distance and width are varied between blocks, but kept constant within each block. One condition of the experiment is a combination of distance and width, where all distances and widths are in screen (not mouse) coordinates. We cover 4 different ID (2,4,6,8) with two different distances. One distance is 212 mm, with target widths of 0.83, 3.32, 14.1, and 70.6 mm, respectively. The other distance is 353 mm, with target widths of 1.38, 5.54, 23.5, and 118 mm, respectively. Each combination is repeated for 102 trials, and the order of conditions is counterbalanced using a Latin square." [21, p. 12]

22 of these 102 trials are for preparation and are not included in this dataset. Therefore, the set contains 80 trials per individual task. Lastly, pointer position was filtered with a Savitzky-Golay filter with a 4th degree polynomial and a window size of 101 samples. This is to reduce the noise in the signal in the derivatives' acceleration and velocity. [21, p. 13] The first included trial for target width 0.83mm is shown in Figure 4.1.

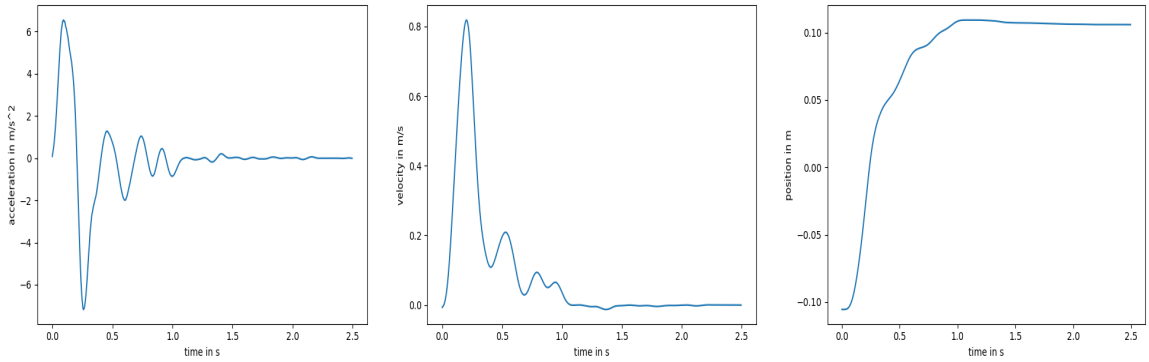


Figure 4.1: Human movement data for first target with 0.83mm width. Left: Acceleration. Mid: Velocity. Right: Position of cursor

4.1.2 Keras and Tensorflow

The networks are implemented and trained using Python and Keras [7]. Keras is an open-source library for Python, that has been written by François Chollet. Keras allows for a simple implementation of neural networks using backend-frameworks like TensorFlow [2] or Theano [3]. For this thesis, Keras version 2.2.4 with Python version 3.5 is used as front-end with Tensorflow version 1.12 as back-end.

4.2 Implementation

After introducing the tools used, I now present the basic structure of the models, their training, and special features of the implementation. I start with the preparation of the already presented data for the training.

4.2.1 Training data

I divide the previously mentioned dataset into a training and a test data set. Thus, after training the networks can be tested with data they have never seen before. As the original dataset contains the data of mouse movements executed by twelve persons, the movements belonging to persons two till twelve constitute the training set. Movements by person one represent the test set.

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Test Set	Training Set										

Additionally, the data can be broken down further. Movements per person can be partitioned into their respective ID. ID is the logarithm of the distance to the middle of the target times two and divided by the width of the target. ID was proposed by Paul Morris Fitts in 1954 [9].

$$ID = log_2(\frac{2D}{W}) \tag{4.1}$$

Motions with ID 2 varies from others with greater IDs because such a low difficulty allows for huge variance in mouse movements. Accordingly, neuronal networks show great difficulty to learn such behavior and predict correctly afterwards. Hence, I have omitted this ID from the datasets.

4.2.2 Principle

A network shall predict a one-dimensional horizontal human mouse movement toward a target. To achieve this, the movement can be broken down into small time steps and be predicted step by step. In this process, one (time)step takes 2 milliseconds. For each step, a neuronal network is fed an input vector. The composition of this input vector varies partially in this thesis. The network then processes the input to an acceleration value as output value. This output represents the predicted acceleration of the next time step. The respective velocity and position values can be gained by integrating the acceleration. I use the Euler method [8] for integration in this thesis.

4.2.3 Custom loss function

While using inherent keras loss functions like mean squared error provides good results for both acceleration and velocity patterns, positional values are neglected. There does not seem to be sufficient motivation to actually hit the pursued target or move the pointer relatively near to it. Instead, the networks reproduce a big acceleration rise at start, but fail to account for the later small correction movements that are both necessary for hitting the target and typical for human movement. Hence, a custom loss function is needed that includes errors of velocity and position besides acceleration.

The predicted velocity can be gained by integrating the predicted acceleration. Likewise, the predicted positional value can be calculated by integrating the predicted velocity. Though, for this procedure both the velocity and position of the last time step are required. These values are also part of the input for the network. Hence, the input vector must be supplied also to the custom loss function. This is possible with the functional Keras API. However, Keras forces a loss function to have exactly two parameters, the true value and the prediction

of the network. Putting the actual loss function inside a wrapper function allows additional parameters to be delivered.

Finally, the magnitude of the acceleration, velocity, and positional values differs. The amplitude of the acceleration is larger than the one of the velocity, which in turn is larger than the positional one. Thereby, an acceleration error weighs considerably more than errors of the other two. Especially the positional errors would be neglected by the network. But these are precisely the ones that need to be corrected. Therefore, the mean squared errors of velocity and position are being multiplied by 10 and 10000, respectively.

```
def custom_loss_wrapper(input_tensor):

    def custom_loss(y_true, y_pred):

        last_velocity = input_tensor[:, 0, 1]
        last_position = input_tensor[:, 0, 2]

        # calculate velocity and position
        velocity_pred = last_velocity + y_pred * 0.002
        velocity_true = last_velocity + y_true * 0.002
        position_pred = last_position + velocity_pred * 0.002
        position_true = last_position + velocity_true * 0.002

        # calculate mean squared error for acceleration, velocity and position
        acceleration_error = K.mean(K.abs(y_pred - y_true), axis=-1)
        velocity_error = K.mean(K.abs(velocity_pred - velocity_true), axis=-1)
        position_error = K.mean(K.abs(position_pred - position_true), axis=-1)

        # error is sum of acc, vel and pos error
        errors = acceleration_error + 10 * velocity_error + 10000 * position_error

        return K.mean(errors)
    return custom_loss
```

4.2.4 Selection of features

The features are the values that are fed to the network for each step. As they determine the precise constitution of the input given to the network, they are critical to the performance of the network.
Looking at the given dataset, I found the relevant columns to be acceleration (sga), velocity (sgv), position (sgy), ID, width (widthm), target and time.

Additional information can be given. For example, you can assign the test persons to their movement as added input. This would increase the success rate during training. But in this way the network learns only the assignment of a movement to a test subject, not the movement itself better.
But even some of this information can be simplified or omitted. Instead of giving the network both the current cursor position and the position of the target center, this information can be summarized outside the network as distance to the target center.
Supplying the target values in addition to the distance does not offer the network any new information. But it still has an effect. It reduces the success rate, even if only slightly. To pass on the time progress since the beginning of the movement has not proved to be advantageous. I assume that the movements vary too much in time for this value to represent a benefit. Above all, relatively late clicks, i. e., large time values, are likely to confuse the net.
ID and width, however, are necessary to identify the type of targets. If you train a net with given targets, these two values are indispensable. However, such a broad spectrum of possible motion sequences poses a problem for the net. For example, some targets at difficulty level 8 are not even one millimeter large. Great precision is therefore required in order to hit these targets. The movements of smaller IDs differ significantly from those necessary for this. That's one of the reasons why I did not take a closer look at ID 2 in this thesis. But also movements of difficulty level 4 show big differences to level 8. Large enough that the level 8 objectives are no longer met. The cursor is near the target, but not on it at the time of the test. Therefore, the models are trained and tested for only one ID at a time. Due to this restriction, the ID in the input is also omitted, since this value would be the same for all targets that the model sees.
However, the width of the target is kept. One could go even further and train the models only for a specific width with a specific ID. This would probably improve the accuracy even more. On the other hand, the performance range of the resulting model is severely limited. Thus, I have excluded such training from my experiments.

4.2.5 Comparing models with different features

Now the result between models with different selected features is illustrated briefly. One model uses all seven features mentioned above. It is a fully connected feed-forward neural network. It has two hidden layers. The first layer has 48 nodes, the second 35. ID belongs to the input, so the model is trained with IDs 4, 6 and 8. The net was trained for 4 epochs with batch size 256. The data was shuffled. 15 percent of the data was used for validation. The loss function was the custom loss function mentioned above. Adaptive moment estimation [16] (Adam) was used as the optimization function.
The other network was trained with four features. The input for this model does not contain the ID for the target. Hence, this model war only trained with targets of one ID. In this case, the model is trained with ID 8 targets. All the specific training data can be found in Table 4.1 on page 24. Figure 4.2 shows the difference between the models. It also

illustrates the problem of models that are trained with different target IDs and have more input features trying to hit very small targets. In the case of figure 4.2, the target is only 0.83 mm small. Such models mimic a human movement and move the cursor close to the target. But they can only hit it rarely. Generally speaking, the bigger the targets become, the higher the hit rates. But even relatively large targets with ID 4 benefit from fewer input values and more specific training.

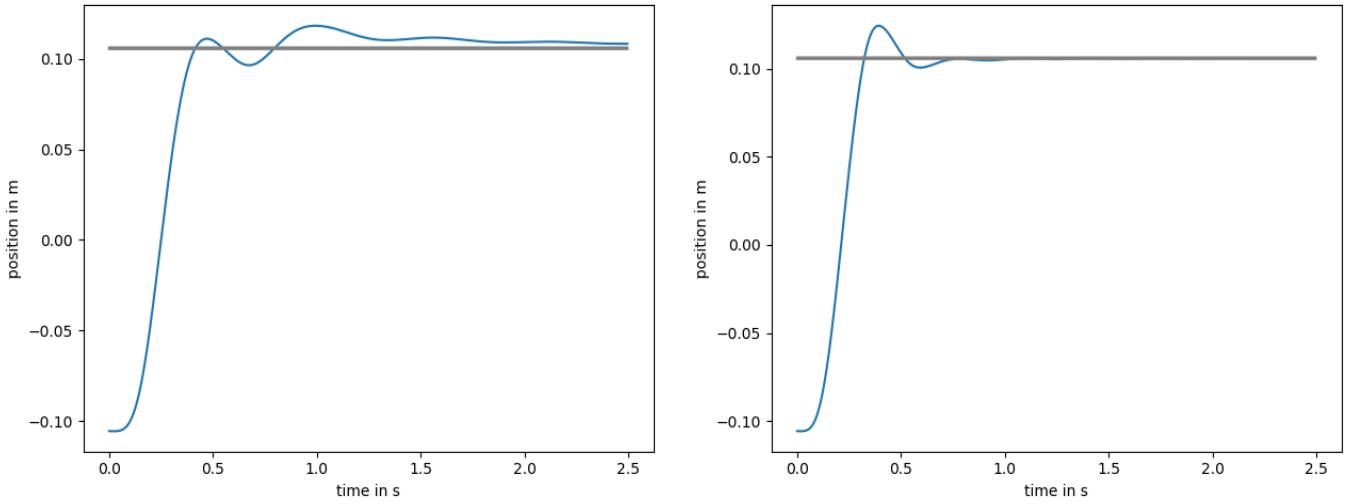


Figure 4.2: Model generated movement in blue tries to hit the target in gray. Target width is 0.83mm, index of difficulty is 8. On the left is the FFNN with seven features, on the right the FFNN with 4 features.

4.3 Models

I now move from the basic structure to the different, specific structures of the models used. The training of the individual models also differs and is now presented.

4.3.1 Preprocessing input data

As already mentioned, the position is given as the distance to the target center. In addition, I cut off the reaction time for each person at the first goal of each size. Because this complicates learning the movements for the network. On top of that, the cut-off reaction time can be manually added again before generated movements. I use the acceleration value of 0.05 as the start of the surge and thus the start of the actual movement. Further recommended data preprocessing methods [17, p.8f] such as standardizing or min-max normalizing the input data to an interval from zero to one worsened the results in my experiments.
Lastly, the input data for the FFNN networks was shuffled prior to training. Shuffling is advisable because networks learn learn faster with an unexpected sample [17, p. 7]. And

randomly shuffled data provides more unexpected samples compared to data that is very similar from step to step.

4.3.2 Models

Following the above outlined principle and using the introduced custom loss function, I have built some models. For each remaining ID in the datasets, I have built both a FFNN and a RNN. The precise structure of these models varies from one case to another. A model structure that works fine for one ID can perform considerably worse for another.
As input, these models take the acceleration, velocity and distance from the last time step combined with the width of the current target in this order. Hence, for all models the first input layer consists of four neurons. Likewise, all models have one neuron as last output layer after their individual hidden layers. Table 4.1 on page 24 shows the composition of the individual layers. Figure 4.3 shows the structure of the FFNN for ID 6.
As optimization function all models use Adam. The learning rate was set as $\alpha = 0.001$. The exponential decay rates for the moment estimates β_1 and β_2 are set as $\beta_1 = 0.9$ and $\beta_2 = 0.999$, as recommended in [16, p.2].
Lastly, as each model is only build for one ID, the training data consists of all targets with this ID of the test persons two to twelve.
For example, the FFNN for ID 6, referenced in 4.3, is coded with the following code. In this code segment, features the number of columns of the input.

```
input_tensor = Input(shape=(features,)) # shape(4, )
hidden_layer_1 = Dense(50, activation='relu')(input_tensor)
hidden_layer_2 = Dense(38, activation='relu')(hidden_layer_1)
out = Dense(1, activation='linear')(hidden_layer_2)
model = Model(input_tensor, out)
```

Similarly, its RNN counterpart for ID 6 is created with this code:

```
input_tensor = Input(shape=(lookback, features))    # shape(1,4)
hidden_layer_1 = LSTM(50)(input_tensor)
out = Dense(1, activation='linear')(hidden_layer_1)
model = Model(input_tensor, out)
```

4.3.3 Model features

Many techniques have been proposed in order to improve deep learning. One example for such a technique is batch normalization. Batch normalization can increase the learning rate and improve the results [14]. My experiments show that the models used in this thesis do not benefit from this technique. Other techniques that do not seem to provide benefits in my experiments are dropout, added Gaussian Noise between layers or kernel regularizers.

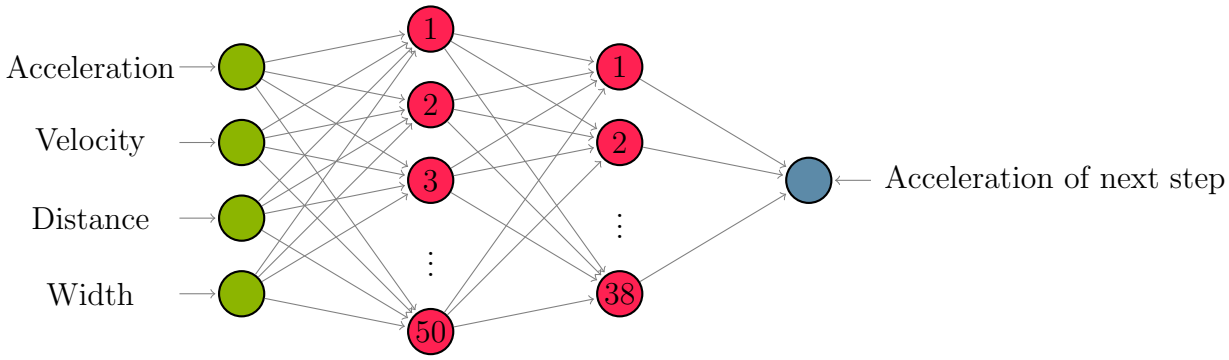


Figure 4.3: Architecture of the FFNN for tasks of ID 6

4.3.4 Training

Processing the whole given training data once is called an epoch. But a network cannot usually handle the entire dataset at once. Therefore it is given a number of training examples, called the batch size. The number of batches that are needed to fulfill an epoch, is called an iteration. [24] The previously mentioned shuffling process for FFNNs gets repeated after each epoch.

The FFNN accepts a two-dimensional array as input. The columns of this array represent the features. Recurrent networks require a different input format. The format of the input data indicates the number of time steps that are included. For this purpose, the input needs to be reshaped into a three-dimensional array. The first dimension specifies the amount of data, the second the time steps and the last one the amount of features. [1]

```
RNN input shape: (batchsize, timesteps, input dimension)
```

For this thesis, time steps for all recurrent models are set to one. Experimenting with different step sizes always showed a worse model performance. The larger the step size, the worse the performance.

Hence, the data was reshaped from (length of data, features) to (length of data, 1, features). Features are in this case acceleration, velocity and position from the last time step combined with target width.

Table 4.1 specifies the precise parameters used in the training for each model.

Table 4.1: Model structures and specific training parameters

Model Type	ID of data	Hidden Layers	Epochs	Batch size	Data shuffled?
FFNN	8	50	200	320	Yes
RNN		50	200		No
FFNN	6	50 - 38	50		Yes
RNN		50	150		No
FFNN	4	50 - 38	8		Yes
RNN		60	150		No

4.3.5 Prediction

As previously mentioned, the not yet used data of person one serves as reference when judging the predictions of the networks. Though, the network is not supposed to clone the reference movement. On the contrary, minor differences are to be expected as the network has never seen the reference data while training. The reference data only assists in the assessment whether the movement looks like a human movement.

For prediction, the networks use the reference data. The starting acceleration, velocity and position is the same starting values as in the reference data. The target width equals the one from the reference data in all steps. Given this data, the first step can be predicted by the networks. The acceleration value of the previous step is used each step in addition to the respective velocity, and position values gained by integrating the acceleration. In this manner, the networks predict recursively whole movements and subsequent sequences of movements from the same starting point as the reference data.

Chapter 5

Result

After introducing the specific models, I present the results of my experiments and evaluate the model-generated movements. For this purpose, I first determine the evaluation bases.

5.1 Model Fitness Criteria

When looking at the results, I use the following measure to judge the results. I use the error sum of squares (SSE) as error measure. SSE calculates the sum of the squared difference of the predicted values when compared to their counterparts of the original data. The SSE values can mislead though. For example, the prediction curve might be generally correct but lag behind a few milliseconds. The SSE value would cumulate in this scenario. Hence, additional graphical analysis is required in order to avoid false conclusions.

$$SSE = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{5.1}$$

The first fitness criterion is the pointer position. Still, as the models might move correctly right before the target and then stop or overshoot, target hit frequency are displayed as well. Additionally, acceleration and velocity are useful in order to judge whether the model produces human-like movements.

5.1.1 Test data

As in chapter 4 outlined, the test data set contains all movements from one person. The models have not seen movements from this person before in order to get a realistic estimation of their efficiency. They were only tested against movements of the same ID they were trained with. For each target, the first line was omitted. Acceleration, velocity, and position from this line function as part of the first input data for the models.

5.2 Comparison between models

With established fitness criteria I now show different evaluations and comparisons between the models. I start with the target hit frequency.

5.2.1 Target hit frequency

The models are given 80 targets to hit for each target. There are two targets per ID. A target is considered as hit when the cursor is on the target when the timer is reset to zero. That is the point when the original human mouse click happened. This method of counting target hits excludes the times when the model would need more time steps to hit the target. Also, the model might hit the target sooner than human of the reference data and instead of staying on the target move the cursor away again. Table 5.1 on page 26 displays the individual target hit frequency of each model.

Lastly, it is important to remember that the models are supposed to learn human behavior. This includes failing sometimes.

Table 5.1: Target hit frequencies

ID	8		6		4	
Target in mm	0.83	1.38	3.32	5.54	14.1	23.5
FFNN	69	79	63	48	60	38
RNN	70	70	63	71	60	39

5.2.2 Error rates

After establishing the target hit frequencies, I outline the individual error rates of these models. Hitting the target is part of the task, but not the whole task. The models are supposed to hit the targets in a human-like movement. In order to judge the movements in this regard, both the error rates and visual analysis is required. Table 5.2 on page 26 shows the positional SSE values. The models were tested against the median of moves trying to hit the target on the right. In this process, all the moves were padded with their last value to match the length of the movement with the longest duration.

Table 5.2: Positional SSE

ID	8		6		4	
Target in mm	0.83	1.38	3.32	5.54	14.1	23.5
FFNN	0.62	0.67	0.01	2.94	0.01	2.73
RNN	0.12	0.05	1.10	2.36	0.01	3.56

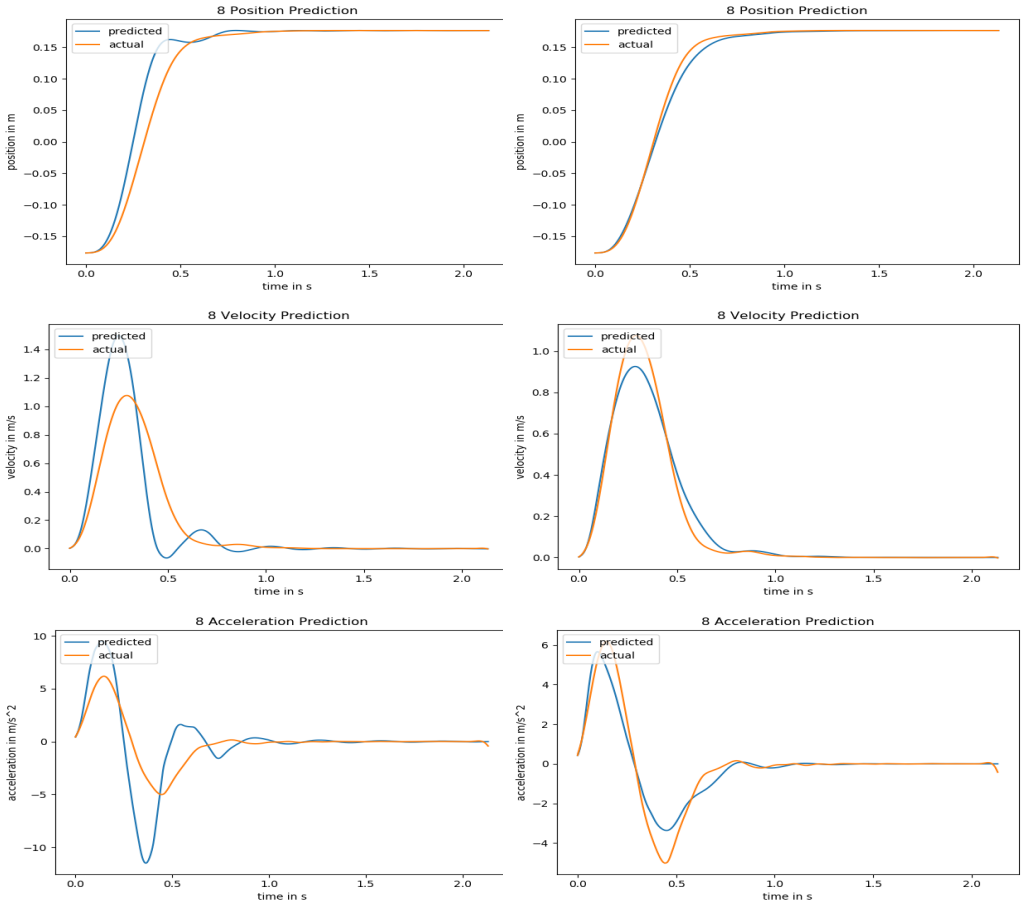
Table 5.3: SSE of acceleration

ID	8		6		4	
Target in mm	0.83	1.38	3.32	5.54	14.1	23.5
FFNN	2310.01	5382.70	477.15	1238.34	126.74	897.55
RNN	719.37	345.40	1253.91	47453.55	148.53	1279.19

Interestingly, in all these cases a higher positional SSE also translates into a higher acceleration SSE and vice versa.

While the RNN model beats its FFNN counterpart for targets of difficulty 8, it loses the comparison for ID 6. ID 4 is close, but also a slight win for the FFNN.

In addition, the following pictures illustrate the comparison of the networks with the mean original movement for ID 8.



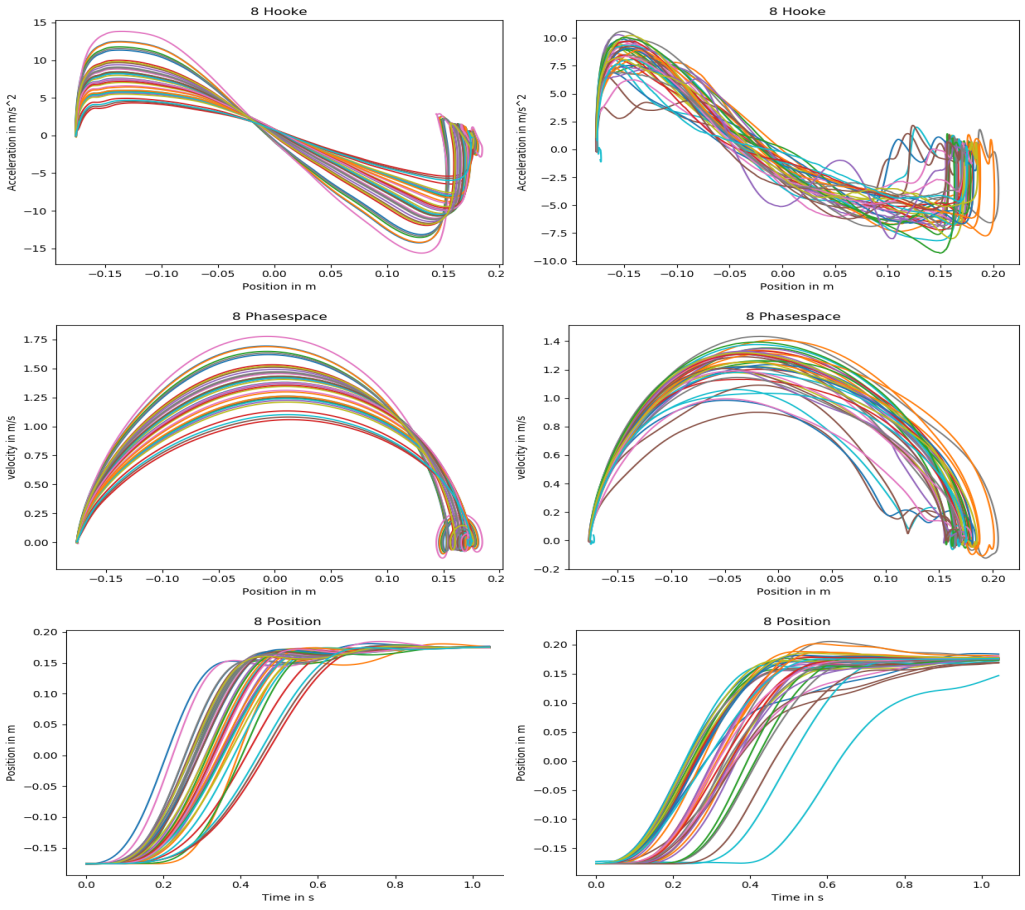
(a) DNN (b) RNN

Figure 5.1: Model predicts cursor position in blue. Median of ID 8 movements with target width 1.38mm of P1 in orange.

The SSE values indicate that the RNN for ID 8 provides movements that resemble the reference data more closely. Figure 5.1 confirms this.

5.3 Comparing human and model movements

After the fitness of the different model types has been shown, the similarity of the movements to the human ones can now be tested. This is best done visually, because deviations immediately catch the eye. For this purpose I use different plots. Hooke plots show the acceleration on the position. Phasespace plots show the velocity on the position. And the position plot against time is also displayed again. Movements are cut down to the shortest movement. All test movements with target 1.38mm are shown.



(a) DNN movement (b) Human movement

Figure 5.2: All 1.38mm target movements compared between the ID 8 FFNN (left) and the original human movement (right). Top: Hooke plot shows acceleration on position. Mid: phasespace plot shows velocity on position. Lower: Position plotted against time

As can be seen in Figure 5.2, human movements vary distinctly more. Generated movements all follow the same basic structure. Characteristics of human movements such as overshooting with subsequent correction are visible in both the hooke and phasespace plot. The models have learned this human characteristic. The large acceleration surge at the beginning of a movement can also be seen in the hooke plot. But some of the generated movements have a smaller surge than can be seen in the reference movements. Here the model covers a larger rise spectrum. Most human movements have a similarly large surge, with some outliers. Finally, strong outliers that clearly miss the target or alternately show rising and falling accelerations are found only in the human plots.

Chapter 6

Limitations

After presenting the results of my experiments, I now go into some limitations of my work. The data set only includes people who regularly work with the mouse, i. e., are experienced in using the mouse [21]. Accordingly, the networks have not learned to imitate the movements of inexperienced mouse users.
In addition, the networks are only trained and tested for targets with ID 4 to 8. Targets with smaller or larger IDs are not covered in this thesis. ID intermediate values, for example 5 or 7, are not taken into account either. A test would only have shown whether the models hit the targets, not whether the generated movement resembles a human movement. The human reference data are missing for this. After all, modeling a human movement is the subject of this thesis.
A problem with this topic is to determine what exactly a human movement is. Because one characteristic of human movement is that it varies slightly. This is also visible in Figure 5.2 on page 28. The network generated movements are more similar. Outliers such as delayed movements are less common. In order to make the movements more human-like, a short delay before the start could be inserted for some randomly selected movements in order to simulate a reaction time.

Chapter 7

Related Work

Following the presentation of some limitations, I now describe some existing papers that are similar to the topic of this thesis.
I have not found any other work that has already dealt with the modeling of mouse movements using neural networks. But there are those who have dealt with the classification of a user by his mouse movements with the tool of neural networks.
Shen et al. [25] propose a user authentication approach that is based on a fixed mouse-operation task. The mouse data generated in the process provide information about the specific user. The mouse data generated in the process provide information about the specific user. As tools for classification, neural networks, a one-class support vector machine and the k-nearest neighbors algorithm are used and evaluated. The one-class Support Vector Machine performs best.
Jorgensen and Yu [15] review existing authentication approaches based on mouse dynamics and notice some limitations with this approach to authentication. Among these methods are also neural networks. According to the authors, this approach has, among other things, the problem of a verification time of at least 17 minutes.

Chapter 8

Conclusion

8.1 Future work

After presenting this thesis, I show some possible approaches that build on this work. For future work the potential of the models can be increased by adding the vertical layer. After all, most actual mouse movements are two-dimensional. This may require deeper models. Above all, however, a two-dimensional data set is required for training. Alternatively, using unsupervised learning might generate even more interesting results. That way, the models do not learn how to imitate people. Instead, they invent their own movement technique. Whether the models act exactly like humans or to what extent the self-learned movements differ from human ones could increase the understanding of mouse dynamics.

8.2 Conclusion

I created two neural network models for each ID 4, 6, 8 of the basis dataset. Then I trained them to generate human-like horizontal movements. In the process, the biggest problem is actually hitting the target. I managed to overcome this by making the models simpler. Particularly reducing the number of input features proved to deliver better results. Also more specific training and application of the model for only one ID contributes to higher positional accuracy.
All models successfully mimic human movements. Typical features of a human motion such as shooting over the target and then correcting the position emulate the models. However, some small differences remain between the given and the generated movements. Overall, feed-forward neural networks and recurrent neural networks compete closely, though in the end RNNs can achieve better results.

Bibliography

[1] Rnn. https://keras.io/layers/recurrent/.

[2] Tensorflow. https://www.tensorflow.org/.

[3] Theano. http://www.deeplearning.net/software/theano/.

[4] Abdulwahed Aboukarima, Hussien Elsoury, and M Menyawi. Artificial neural network model for the prediction of the cotton crop leaf area. *International Journal of Plant & Soil Science*, 8:1–13, 01 2015.

[5] Michael A. Arbib. *The Handbook of brain theory and neural networks*. The MIT Press, 2003.

[6] Imad Basheer and M.N. Hajmeer. Artificial neural networks: Fundamentals, computing, design, and application. *Journal of microbiological methods*, 43:3–31, 01 2001.

[7] François Chollet. Keras: The python deep learning library. https://keras.io/.

[8] Leonhard Euler. *Institutionum calculi integralis*. Imper. Acad. Imperialis Scient., 1845.

[9] Paul M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 1954.

[10] Felix Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12:2451–71, 10 2000.

[11] Klaus Greff, Rupesh Kumar Srivastava, Jan KoutnÃk, Bas R. Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28, 03 2015.

[12] Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, and H. Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Macmillan Magazines Ltd*, 2000.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[14] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[15] Zach Jorgensen and Ting Yu. On mouse dynamics as a behavioral biometric for authentication. pages 476–482, 01 2011.

[16] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *ICLR conference*, 2015.

[17] Yann Lecun, Leon Bottou, Genevieve Orr, and Klaus-Robert Müller. Efficient backprop. 08 2000.

[18] Zachary C. Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning, 2015.

[19] Pierre Lison. An introduction to machine learning. *Language Technology Group (LTG), 1*, 35, 2015.

[20] Jörg Müller, Antti Oulasvirta, and Roderick Murray-Smith. Control theoretic models of pointing. `http://joergmueller.info/controlpointing/`.

[21] Jörg Müller, Antti Oulasvirta, and Roderick Murray-Smith. Control theoretic models of pointing. *ACM Transactions on Computer-Human Interaction Vol. 24*, 2017.

[22] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017.

[23] Raúl Rojas. *The Backpropagation Algorithm*, pages 149–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[24] Sagar Sharma. Epoch vs batch size vs iterations, Sep 2017.

[25] C. Shen, Z. Cai, X. Guan, Y. Du, and R. A. Maxion. User authentication through mouse dynamics. *IEEE Transactions on Information Forensics and Security*, 8(1):16–30, Jan 2013.

[26] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.