Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Self-supervised prediction of sensory-motor trajectories with deep neural nets

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Felix Stiehler
geboren am: 11.02.1987
geboren in: Potsdam

Gutachter/innen: Prof. Dr. Verena Hafner
Prof. Dr. med. Felix Blankenburg

eingereicht am: ..... verteidigt am: .....

Contents

Acknowledgements

I would like to thank Verena Hafner for supervising the thesis and providing a lot of conceptual freedom. I would also like to thank all members from the adaptive systems group for their great support. Big special thanks to Oswald Berthold, Guido Schillaci, Andreas Gerken, Antonio Pico Villalpando and Juan Manuel Acevedo Valle for always having an open ear.

I would also like to thank my father and grandmother for supporting me throughout this journey and my girlfriend Svenja for all the ongoing support.

We investigated multiple deep learning methods for the prediction of future spatiotemporal touch input made on a touchscreen by a human experimenter. We recorded seven different datasets, which were all made by repeating a predefined geometric shape of varying length, complexity and noise. The task was to predict a 40 point long continuation of a 30 point long input. Two main models emerged, which both made use of a special kind of recurrent neural network, called Long Short-Term Memory (LSTM), but differ in their respective output. One, called the *Basic LSTM*, is simply trained to predict the next concrete point. The other one takes the form of a Mixture Density Network (MDN), which combines a regular neural network with a mixture density model. We call this architecture the *LSTM-MDN*. Both approaches have their strengths and weaknesses for different types of datasets, with their main discrepancy being in how they are able to cope with big jumps in the touch trajectories. While the LSTM-MDN was able to incorporate them in the predictions, the Basic LSTM was not. However, this came at the price of worse predictions for simpler trajectories and the occasional occurrence of completely wrong predictions. To analyse this more complex behavior on the output layer, we investigated the dynamics of the probabilistic encoding of the LSTM-MDN for different prediction scenarios in detail. Quantitative results for all main models and datasets are reported. We also show results of experiments for dimensional reduction of the touch datasets with a deep autoencoder network.

1. Introduction

To predict well just feels right. Whether it is on a small scale when you know exactly how each object looks and feels in your room or when you know how the seasons will change the world around you over the course of a year. To predict well means to be in sync with the world around you and it is something that we are much better at than any computer program or robot out there, at least when it comes to flexibility and adaptability. This is in part due to the fact that the design of artificial intelligence historically almost always focused only on specific tasks, and did rarely trust on the emergence of complicated and surprising behavior from learning with a more general goal. Recently, there have been strong trends towards that with techniques like reinforcement learning [61], which leave an agent a great amount of freedom in how to learn how to reach an arbitrary goal. Learning, that has just the prediction of the next incoming raw sensor information at its core, seems to be a promising avenue to explore.

Prediction, no doubt, is powerful and the theory of predictive coding [8] states, that everything we do, including all thoughts and emotions, is ultimately rooted in our drive to minimize the error of our predictions of what comes next through our numerous sensory channels. Letting robots learn with this principle in mind could lead to novel

behaviors, perhaps even some that resemble basic behavioral patterns of mammals, but it is not an easy feat to find the right settings for such a learning experiment.

One of the problems is that the learning has to be powerful enough to encode complex spatiotemporal statistics (for senses with a spatial dimension like vision and touch) in order to make close-to-accurate predictions. In this work, we focused on a small step towards the aforementioned goal: we explored ways how to accurately predict raw touch input made by a human experimenter on a touchscreen and report results for various prediction methods. We focused on producing the most overall convincing (and not just quantitatively optimal) predictions. Multiple different touch trajectories were recorded that each follow a specific pattern. While they are similar in the general generation scheme, they differ greatly in complexity and noise. We found, for example, that it makes a big difference for the learning if a touch trajectory contains big jumps, which turned out to be difficult to predict.

To achieve high prediction accuracies, we decided to use techniques from the vast field of deep learning [18], which gained a lot of popularity in recent years. It was used to drive improvements on numerous fronts, spanning from all kinds of qualitative problems like image classification [33] to quantitative ones such as speech synthesis [67].

We think that the results of this work can be applied to a real-time setting in a robot with artificial skin without big changes to the model itself, at least for inference. It should also not be very complicated to add additional sensor modalities into the prediction algorithm. Although our datasets all follow a predefined pattern, we think that our models have shown enough flexibility in how they encode the different statistics of each dataset that they can readily be used for arbitrary touch input in real-time. We tested all our models for a consecutive prediction of 40 points after receiving a 30 point starting sequence, which should span more time than a typical time gap between two touch interrupts of a robot. However, it is not possible to explicitly predict and learn on different time scales. When deployed in a real-time setting, the model could have access to a longer history with the same data resolution if it is trained for that.

To fully use a model similar to this for a predictive coding-motivated approach to learning in a real-time setting, the model would also need to learn and adapt to changing inputs. Such a setting would be more complicated to implement as the current model is trained offline, but it would be sufficient if the adaptation rate of the network matches the pace of the changes in the input statistics.

The work is structured as follows: Chapter 2 contains related work and background knowledge about deep neural networks, predictive coding and touch. Chapter 3 shows an analysis of the data we used for our experiments, as this is a vital point for each machine learning application. Chapter 4 then describes our two main prediction architectures together with another experiment with a deep autoencoder. It also contains various implementation details. Chapter 5 reports the experimental results and compares our main models and Chapter 6 then closes out the work with a discussion of our findings and an analysis of multiple possible avenues for future work.

2. Background and related work

This chapter contains the background information and the related work, which the following chapters are built upon. Section 2.1 first talks about deep neural networks in general and then contains more detailed information about the techniques used in this work, except for some parts of the discussion, which are then explained separately. Section 2.2 explains briefly the key points of the predictive coding theory and Section 2.3 talks about different aspects of touch in general and other related work.

2.1. Deep neural networks

Many parts of this chapter make use of information from the book *Deep Learning* by Goodfellow et al. [18].

In this work, we combine different variants of Deep Neural Networks (DNNs) to make predictions about the continuation of sensory-motor trajectories. These methods are widely used in machine learning and we employ them as they are considered the current state-of-the-art techniques to give a computer the ability to learn complex models about the statistics of a given amount of data.

The most basic neural network is the *Multilayer Perceptron* (MLP), which is the conceptual basis for all types of DNNs used in this work. A MLP takes a vector x as input and outputs a vector y. The data therefore consists of inputs x and corresponding labels y and it is assumed that there exists a function f^* that maps each x to a specific y:

$$y = f^*(x) \tag{1}$$

A MLP is then trained to approximate f^* with a function f given a set of parameters θ:

$$y' = f(x; \theta) \tag{2}$$

f can be made up of nested functions, making the neural network *deep*. Each function is called a *layer* and the layers that are not in direct contact with either x or y are called *hidden layers*.

At its core, the computation of f is done by a network of simple computational units, that are called *perceptrons*. These units take in a vector and output a scalar. Each layer can consist of multiple perceptrons (or neurons; not to be confused with biological neurons) that, when the layer is a hidden layer, are connected to other neurons in layers up and down stream. The computation inside each neuron is done according to the following equation:

$$o = \phi(w^T i + b) \tag{3}$$

Here, i contains the input vector and o is the scalar output of the neuron. w contains the weights for the input, b is a scalar called the *bias* and ϕ is a non-linear function, called

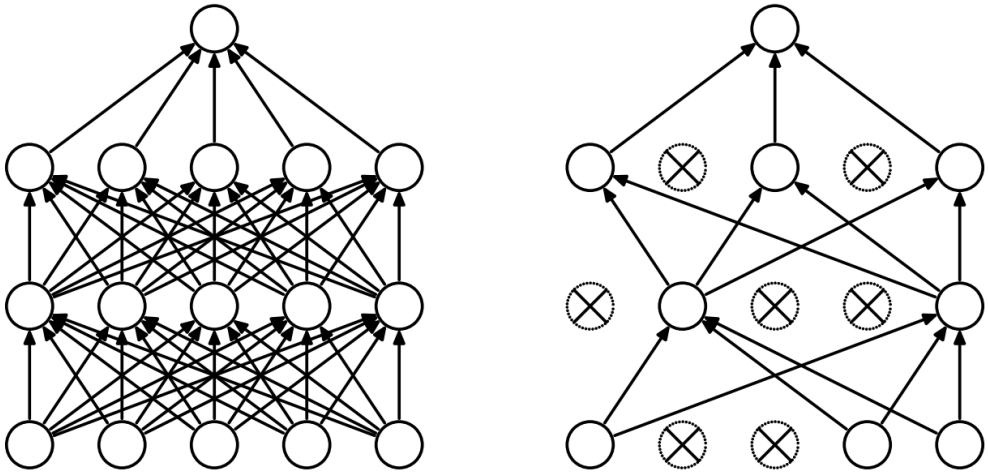
the *activation function*. w can be thought of as a way to emphasize or deemphasize certain parts of the input, while b gives the neuron the ability to shift the linear transformation $w^T i$. ϕ then extends those capabilities with a way to express non-linear mappings from i to o within each unit. Each w and b of each neuron make up θ and is trained to together resemble f^* as close as possible. The training is done with a process called *gradient descent*. Gradient descent changes θ for each given set of training samples, based on the gradient towards a direction so that the output of the network resembles the corresponding labels more closely. The directions in which to change the parameters are obtained by *backpropagation* [51]. An important part of the backpropagation process is the way in which the difference between y and y' is computed. This is done with a *loss function*, which is a hyperparameter that is usually chosen by the model architect and described further in Section 2.1.1. A bigger difference usually results in a stronger signal for the network to change its parameters.

There are many different ways how to apply the gradient to the network weights. This is done by an algorithm called *optimizer*. Historically, the gradients were either calculated for the whole training set at once (called *batch gradient descent*) or each individual training sample (called *stochastic gradient descent*), with the former method showing very slow convergence and the later having big jumps in the loss function, as the network reacts to each training sample individually. Modern gradient descent implementation all perform *mini-batch gradient descent*, which combines the best of both worlds. Here, the gradients are calculated for a batch of samples of predefined size (called the *batch size*). The choice of this hyperparameter usually does not have a big impact on the outcome of the learning, but it can significantly influence convergence times. There are many other optimizers, which all aim to further optimize the convergence time and accuracy in the loss space. They usually come with multiple hyperparameters like the *learning rate*, which determines how much a network weight is changed according to its gradient.

The training is usually done throughout multiple *epochs*. One epoch refers to the learning with the whole training set, which is then repeated in the next iteration. Also, the whole dataset is commonly divided into three parts: the training data, the validation data and the test data. The training data is used for the actual training of the network and usually contains the biggest chunk of the data. The validation data validates the training progress after each epoch (see Section 2.1.2 for situations where this information can be useful). The test dataset is only used at the very end of the model search to ultimately quantify the learning results on untouched data.

2.1.1. Loss functions

The calculation of the training error is an essential part of every deep learning algorithm. The formula to calculate this is called the *loss function*. There are many different ways to go about the loss calculation. One of the most straight forward ways is to simply calculate it as the average squared distance between the labels y and their



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Illustration of dropout. A former fully connected network becomes a lot more sparse, with highly different connection layouts for every training iteration. Taken from [57].

corresponding network outputs y' for a test dataset $\mathcal{X}$:

$$\mathcal{L}_{MSE}(\mathcal{X}) = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} (y - y')^2 \tag{4}$$

This is called the *mean squared error* (MSE). The objective of the function f mentioned in Equation 2 is then to find parameters θ, which minimize $\mathcal{L}$. There are many other ways to set up the loss function, such as using an absolute distance instead of the squared one. This is called the *mean absolute error*.

2.1.2. Regularization

Machine learning algorithms only care about the data they are trained with, while the typical goal is to end up with an inference process that has learned to generalize from the data to new unseen scenarios [17, 18]. This is usually tested during training time by the discrepancy of the results for the training and validation test set (see Section 2.1). It is called *overfitting* if the network does perform significantly worse on the validation set than on the training set.

There are numerous ways to potentially prevent or lessen overfitting. One way is to simply collect more training data, while many others are concerned with the actual learning implementation. One commonly used way is to penalize the network for using big network weights, as they point towards the network heavily overemphasizing specific parts of the calculation, which typically leads to a weak generalization. This is implemented as an additional term in the loss function (for example in addition to the one shown in Equation 4), which then forces the network to juggle both the objective of loss minimization as well as the regularization at the same time.

Another very common way is to use *dropout* [57], where every node of a layer may not be used during training of one batch according to a predefined probability. The method is illustrated in Figure 1. This effectively means, that the layout of the training network may be different for each training step. Applying dropout forces the network to generalize and build up redundancies in the computation, which can be a very effective way to prevent overfitting.

It is also possible to stop the network from further adjusting its weights when the progress on the generalization has stopped. This can be observed by the results for the validation dataset not improving anymore and is usually the moment when strong overfitting starts to occur. This effect can be mitigated by stopping the network early from further adjustments. This technique is aptly called *early stopping* [46].

2.1.3. Long short-term memory (LSTM)

LSTMs [29] belong to the class of recurrent neural networks (RNN), that are most commonly used for sequence modeling. RNNs apply the same data transformation for every input vector of one coherent data stream. Classical single layer RNNs contain a n-dimensional hidden state h_t and are getting a single time step vector x with length m as input. This is processed by replacing the hidden state with the output of a function f that is parametrized by the previous hidden state and the current input vector.

$$h_t = f(h_{t-1} + x_t) \tag{5}$$

f is usually an elementwise tanh or sigmoid function. Additionally, three independent affine transformations $T_{i,j} : \mathbb{R}^i \to \mathbb{R}^j$ are being used to control the different aspects of the data flow inside the RNN.

$$h_t = f(T_{n,n}h_{t-1} + T_{m,n}x_t) \tag{6}$$

The output of length o is derived from the hidden state.

$$y_t = T_{n,o}h_t \tag{7}$$

The affine transformations are done just as in Equation 3 by applying a weight matrix W and a bias vector b to an input i. The concrete values of W and b are learned through a variant of backpropagation called *backpropagation through time* (BPTT) [64]. This allows for the gradient to be computed and applied throughout the computation of multiple successive time steps, where the internal values of the RNN at time step t depend on all previous inputs and the initial weights. This process is called *unrolling* the network.

RNNs can also be stacked in layers on top of each other according to the following update equation:

$$h_t^l = f(T_{n,n}h_t^{l-1} + T_{n,n}h_{t-1}^l) \tag{8}$$

RNNs are touring-complete [55] and there have been efforts to leverage this capability by building a Neural Touring Machine that is able to learn simple algorithms while being trained end-to-end with gradient descent [23].

However, when applying classical RNNs to simple real world problems, they are very difficult to make work. One key problem is the emergence of extreme gradients [3]. Gradients can quickly go to zero or rise to very big values, with both cases posing big problems for the learning. Diminishing gradients are called the *vanishing gradient problem*, where the gradient signal shrinks fast due to the many multiplications in the backpropagation process for unrolled sequences. This poses a big problem to the task of keeping a long term memory inside the hidden states. *Exploding gradients* are often dealt with by a simple technique called *gradient clipping*, where the gradients itself are thresholded [45].

LSTMs offer a solution to these problems through a more complex internal structure. They can be used and plugged into each other just like classic RNNs but they consist of two internal states h and c of size n together with a weight matrix that effectively modulates how the internal states are updated when new input comes in. The following equations show how LSTMs work internally:

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} T_{2n,4n} \begin{pmatrix} h_{t-1}^{l} \\ h_{t}^{l-1} \end{pmatrix} \tag{9}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g \tag{10}$$

$$h_t^l = o \odot \tanh(c_t^l) \tag{11}$$

Figure 2 visualizes the equations. During every update, four vectors i, f, o and g, called *gates*, are calculated from the inputs. These gates control the internal behavior of the LSTM. The cell c can be thought of as a memory vector and updates to it are modulated by three of the four gates. The hidden state vector h is in more direct contact with the actual output, while c can keep information for longer time periods without having it leak out. This is in direct contrast to classical RNNs where the network state is always altered directly. The LSTM gate f modulates how and when the memory vector is rewritten and i and g control the concrete updates to it. Gate o is responsible for controlling what information of the memory vector is leaking out into the hidden state.

LSTMs deal much better with the vanishing gradient problem of classical RNNs due to the additive interaction in Equation 10, which allows the gradient to be backpropagated through the unrolled network without many losses. However, this is only fully true when the forget gate has a value of 1. LSTMs show a much better resilience against vanishing gradients in practice and can thus learn dependencies of much bigger range than classical RNNs.

While there are many different LSTM variants out there, recent findings by Greff

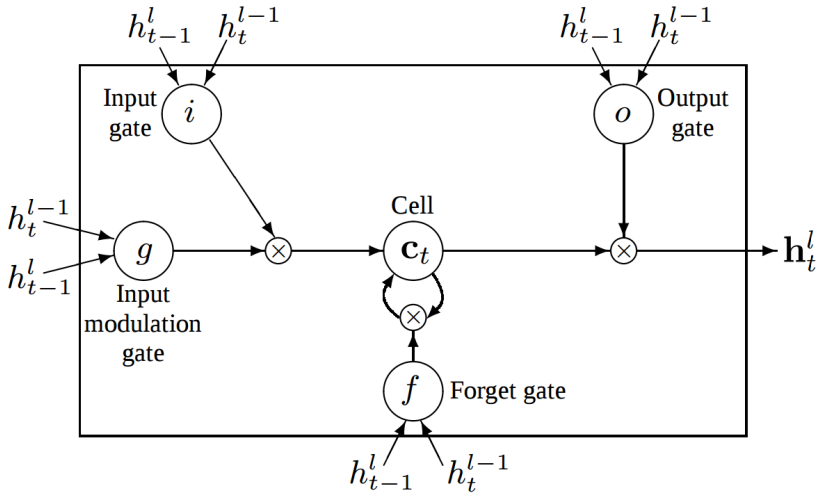


Figure 2: Visualization of the inner workings around an LSTM cell. Taken from [66].

et al. [24] lead to the conclusion that the concrete LSTM implementation only has a minor impact on the performance. LSTMs have been successfully used for handwriting generation [21], machine translation [60] and image captioning [65], amongst many other tasks.

2.1.4. Sequence to sequence learning

While DNNs can be very successful for a mapping of sequences of fixed sizes, they are not able to deal with varying input and output lengths. Architectures involving RNNs appear to be the most suitable for this task, due to their nonlinear dynamics evolving around multidimensional hidden states [20, 60]. As shown in Section 2.1.3, LSTMs are also able to store important information for a long time when processing an input sequence. This makes them especially suitable for sequences that contain long term relations that are relevant for the learning.

The predictions of variable length are done by appending the network output to the previous input in order to predict the next one. This autoregressive procedure makes each time step prediction of a predicted time series highly dependent on the previous ones. This is in stark contrast to the inference process of classical DNNs, where each prediction is completely independent. The feeding back of previously generated tokens allows for predictions of arbitrary length, but it can also lead to an amplification of errors made early in the generation of the sequence [2].

2.1.5. Mixture density networks (MDN)

Neural networks can directly output a prediction but they can also be used to probabilisticly encode an answer. This is done by combining a regular neural network with a mixture density output. The resulting network is called a *Mixture Density Network*

(MDN) [5]. Instead of giving a direct answer, the network outputs the parameters of a Gaussian Mixture Model (GMM) [40], which is then used to sample from. This has the advantage that the neural network is able to express the output statistics for an input x up to an arbitrary level of detail if it has the computational power to do so.

In case of a prediction of real valued 2D touch points it makes sense to use a GMM made of bivariate distributions. The MDN then has to output 6 parameters per mixture component i for N components in total: the means μ^i, the standard deviations σ^i, the correlations ρ^i and the weights of the mixtures π^i.

$$x_t \in \mathbb{R} \times \mathbb{R} \tag{12}$$

$$y_t = \left(\left\{ \mu_t^i, \sigma_t^i, \rho_t^i, \pi_t^i \right\}_{i=1}^{N} \right) \tag{13}$$

The output of the neural network has to be processed further in order to ensure that a GMM can be constructed from them. For example, the π_t values have to sum up to one, so a softmax function is applied to them in our work. All relevant postprocessing steps are shown in the Equations 14 - 16. A hat on a variable denotes the equivalent raw neural network output.

$$\pi_t^i = \frac{\exp(\hat{\pi}_t^i)}{\sum_{j=1}^{N} \exp(\hat{\pi}_t^j)} \implies \pi_t^i \in (0,1), \sum_i \pi_t^i = 1 \tag{14}$$

$$\sigma_t^i = \exp(\hat{\sigma}_t^i) \implies \sigma_t^i > 0 \tag{15}$$

$$\rho_t^i = \tanh(\hat{\rho}_t^i) \implies \rho_t^i \in (-1,1) \tag{16}$$

The probability density function for every possible next input x_{t+1} is then calculated with the refined values as follows:

$$\Pr(x_{t+1}|y_t) = \sum_{i=1}^{N} \pi_t^i \mathcal{N}(x_{t+1}|\mu_t^i, \sigma_t^i, \rho_t^i) \tag{17}$$

The loss of the whole network is then the summed up negative log-likelihood for the T steps of the input sequence x.

$$\mathcal{L}(x) = \sum_{t=1}^{T} -\log \left(\sum_i \pi_t^i \mathcal{N}(x_{t+1}|\mu_t^i, \sigma_t^i, \rho_t^i) \right) \tag{18}$$

MDNs were used successfully in conjunction with LSTMs by Graves [21] to generate convincing-looking handwritings based on the IAM-OnDB dataset [37]. The dataset consists of 12179 handwritten lines and each line is made up of 700 pen touch points on average. The values are represented by two real values and were converted to offset values from one point to the next. The network then predicted the parameters of a bivariate GMM comprised of 20 components to sample a predicted point from. Graves also included information in every point delta of whether the pen was lifted after

the movement was recorded and incorporated this information into every step of the prediction process.

The network can also be altered to synthesize convincing looking handwriting for a given character sequence by giving the network the ability to learn a soft attention window over the input at every given step of the synthesis. In this case, the sequence prediction changes from points-to-points to characters-to-points.

2.1.6. Autoencoder

An autoencoder is a neural network that has the simple mission to output exactly what it gets as input [28, 18]. The common set up is to funnel the input through a bottleneck, the hidden state h, so that the network has to figure out what information is most important in order to minimize the reconstruction loss. More formally, the network consists of an encoder $h = f(x)$ and a decoder part $r = g(h)$. An autoencoder is called *undercomplete* if the encoding dimension is smaller than the input dimension. The whole network is trained end-to-end like a regular MLP and the encoder and decoder networks can consist of all kinds of variants of the MLP archetype. It is even possible to replace the hidden state with an LSTM when focusing on convincing reconstructions of sequences as done by Lotter et al. [38].

Autoencoders learn to approximate PCA if the decoder network is linear, the loss is the mean squared error and no regularization constraints are applied to the network. This is not the desired behavior in most cases. Much more useful representations can be learned if f and g are nonlinear functions. To achieve this, a delicate balance between the size of the hidden space and the power of the encoder and decoder is necessary. For example, if f and g are very powerful nonlinear functions they could learn to incorporate the whole content of all training samples in their weights and map them to a single integer in h without learning anything about abstract representations [18]. A common way to circumvent such behavior in practice is to impose regularization constraints on the network (see Section 2.1.2) .

2.2. Predictive coding

Predictive coding is a theory that suggests that everything we as humans (and other organisms with brain-like structures) do originates from a drive to become better predictors of our next sensory states [8, 15]. While the idea gained a lot of popularity over the past decade, it can be traced back to von Helmholtz in the 1860s [27]. For an embodied agent, there are in principle two ways to reduce prediction error: either by changing what to expect or by changing what to perceive. Predictive coding thus leads to a combined theory of how to deal with the outside world internally (by altering internal world models) and externally (by engaging with the world directly). It offers a unified account of action, perception, cognition and emotion [8] and with that an explanation about how we are able to carry out complex tasks in an ever-changing world.

A more in-depth view of how predictions and prediction errors might come together to ultimately produce such behavior is displayed in Figure 3. Here, the internal world model is built up hierarchically and there are two streams of information flowing between the raw input sensors on the bottom to the most abstract internal models on the top. The bottom-up flow is made up of prediction errors which meet the top-down flow of increasingly more concrete predictions on each level. Both streams are concerned about the same occurrences in space and time and a matching is attempted. Any remaining differences that could not be explained with a model of this abstraction are passed on to the next higher layer and the process continues. The remaining prediction error forces the models to adapt throughout the hierarchy, so that later predictions are less likely to produce such errors. The ultimate goal is to be be able to predict the incoming signal on the very first layer, so no error has to be forwarded and no change has to happen. This theory is in line with the Bayesian brain hypothesis [32], which suggests that the brain employs a generative and probabilistic model of the world, which it tries to keep in sync with incoming sensory inputs.

Computational models that showcase aspects of predictive coding are usually concerned with visual input [49, 56, 31] as compared to other sensory channels. Furthermore, all deep neural networks (see Section 2.1) can be seen as an implementation of the predictive coding frameworks in that they are hierarchical models where each layer changes due to an error signal that is passed on through the network.

2.3. Perspectives on touch

In this section we present different aspects of touch as well as related work that dealt with touch data.

2.3.1. 2/3 power law of planar movement

This work is about the touch input of geometrical forms (see Section 3 for an overview of the shapes) on a flat touchscreen. The movement that created those forms are similar to planar movements made with a pen. Lacquaniti et al. [34] have shown that the instantaneous velocity of drawings depends directly and only on the current curvature. This means that the *angular velocity* stays constant. This holds true for free drawings as well as for movements that follow a predetermined shape. The mathematical relationship takes the approximate form of a power law:

$$a(t) = kc(t)^{\frac{2}{3}} \tag{19}$$

Here, $a(t)$ is the angular velocity at a given point in time and $c(t)$ the respective curvature. This relationship is also referred to as the *2/3 power law*.

Later research by Schaal et al. [53] has shown that the 2/3 power law looses in predictive quality with growing pattern sizes for 3D movements. Here, it seems like, the 2/3 power law seems to be apt only for the generation of smooth trajectories and not as a general rule.

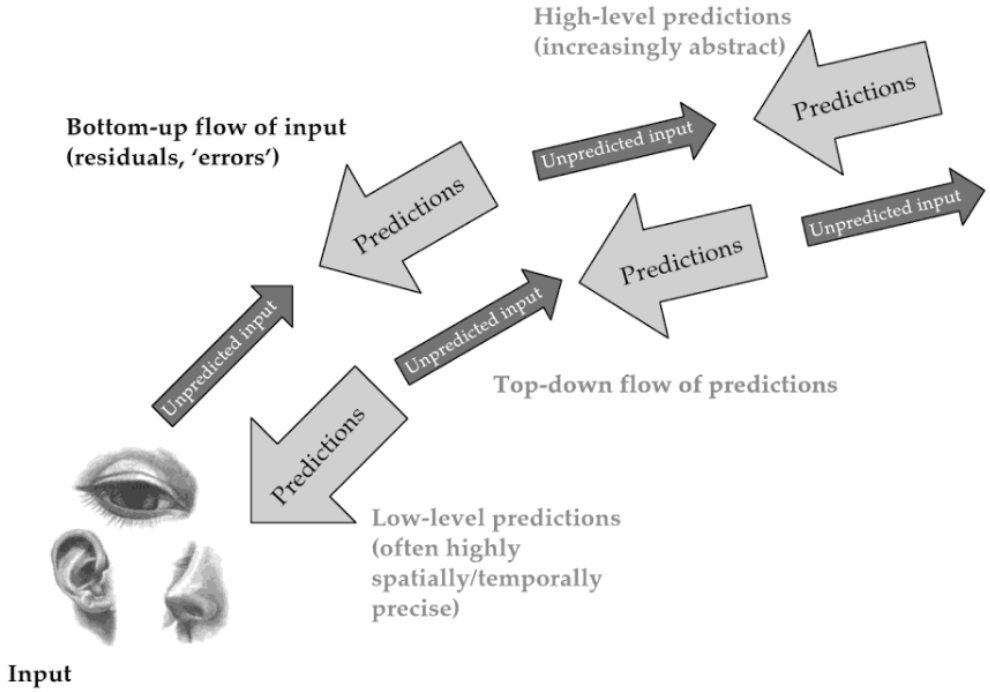


Figure 3: Top-down and Bottom-up flow of predictions and prediction errors according to the predictive coding theory. Levels further away from the input are of higher abstraction. At each level, both streams meet and residual prediction errors are passed on to higher levels to drive changes in more abstract models. Taken from [8].

2.3.2. Sequential learning from touch input and other senses

This work is concerned with the learning of statistical properties of sequential touch input via computational means. Conway et al. [9] investigated this topic for the sequential touch learning in humans and compared it to the sequential learning of vision and audition. It was the first of such work concerned with touch input. The researchers found, that the learning of statistical properties of touch input is possible, but that the accuracy is lower than for the learning of auditory or visual input. Among the three senses, the sequential learning of auditory input seemed to be developed best. This discrepancy may point towards the processing of different sensor inputs in separate cortical areas in the brain when it comes to basic pattern recognition.

2.3.3. Working with touch input

Related work that uses machine learning on touch data can be divided by the way the respective data was acquired. One group worked with data collected from touchscreens or similar devices, while the other received input from material that was or can be used in artificial skin, which in most cases was attached to a robot or robotic parts. The concrete machine learning techniques should be agnostic to how the data was acquired unless the learning has real-time constraints.

This work falls into the first category and the most closely related work is the one by Graves [21], which is described in detail in Section 2.1.5.

Another example for such work is Weir et al. [63], which uses Gaussian process regression to improve the accuracy of input on a touchscreen.

Shirafuji et al. [54] is a good example for the second category. Here, a simple MLP is used to detect if an object is slipping out of a robotic hand during grasping. The hand had been endowed with piezoelectric PVDF film and strain gauges for sensing.

A lot of other work in this field is concerned with touch pattern classification. This is often used to recognize social touch gestures on a robot equipped with artificial skin. Hughes et al. [30] worked with 2D touch data from multiple different sources. They extracted features from data frames with a deep autoencoder and other means. Those features were then processed further using Hidden Markov Models and finally jointly classified with logistic regression in order to detect and classify predefined touch patterns.

Flagg et al. [13] evaluated four machine learning methods to classify affective touch gestures on an artificial furry lap-pet, that can sense touch on the artificial fur as well as on the surface.

Gastaldo et al. [16] classified patterns on a sensor surface made of PVDF piezoelectric film, while focusing on the direct processing of the touch data in tensor form.

3. Data

To consider the concrete data used for machine learning applications is vital for their performance and decisions made for the data collection process and the subsequent pre-processing can make or break the chosen model architecture [52]. For this work, we decided to work with rather short trajectories that were taken from a long motion made by a human on a smartphone screen. Each of those long trajectories were made by repeating a characteristic geometric shape and consists of 45.000 - 55.000 points in total before preprocessing. We collected seven different types and focused on a diversity in shapes and complexity. Example segments of the long trajectories are shown in Figure 4 and additional information about the data can be found in Table 1. The data can be conceptually divided in *continuous motions* and *discontinuous motions*, depending on whether the input finger had to be lifted and moved a considerable distance in order to draw the shapes. Also, some data is made up solely of curved motions while others follow a purely linear concept. One motion combines both by alternating a triangular and circular shape. As the amount of data matters [59], we payed attention to collect rather long motion sequences. This is especially necessary, as the data is quite noisy as it originates from human movement. For the learning, the long trajectories were then cut into 30 point long pieces by using a sliding window approach with a step size of one. The set of subtrajectories is thus much bigger in size on disk and carries many redundancies.

The phone used to collect the data was a Samsung Galaxy S7, which has a display diameter of 12.5 cm, a resolution of 1440 x 2560 pixels (resulting in 577 ppi and an aspect ratio of about 1.78) and seemed to record touch input with about 100 Hz. The data was collected with the Android app Sensors2OSC [1] and consists of 2D single point trajectories in floating point format within a numeric range of $(0, 1)$ in both dimensions. This normalization leads to a different representation of space in the two dimensions as the display is not quadratic in shape. This transformation is reversible and was not found to be problematic for the learning process. However, the nonquadratic shape of the input display can lead to more complex input data with more periods of slowdown in horizontal movements than with vertical ones. Section 2.3.1 explains those effects in more detail.

We removed all points that had at least one coordinate value of -1.0 during pre-processing, which seemed to occur when the input finger was lifted. We also boiled down the original long trajectory to a list of points where each successive point changes in both dimensions. Changes in only one dimension were very common in the data. This could constitute a loss of valuable information as partially duplicate points can be an indication for specific input patterns and speed, but we found this to work better in practice. The 2D point sequence was then converted to relative coordinates, which reduced the effective input and output sequence length by one. This makes the learning invariant under translation and was found to have a big positive impact on the learning results. Possible methods to make the learning invariant under scaling as

[1] `https://sensors2.org/osc/`

Table 1: Number of subtrajectories and the typical shape length for each dataset after preprocessing. The number of subtrajectories varies a lot despite all datasets having a similar number of raw data points due to the preprocessing steps described in Section 3. The typical shape lengths are estimates drawn from a few random samples. The actual shape sizes can differ as the input originate from a human input motion. A longer length can indicate a more difficult learning problem.

Dataset	Number of subtrajectories	Typical shape length
Circles	12689	65
Continuous circles	12429	200
Eights	25073	170
Horizontal lines	19391	140
Rectangles	22357	190
Spaced rectangles	11237	145
Triangles/circles	13547	220

well as rotation are discussed in Section 6.1.3, but are not used in this work. The data was then standardized to have a mean of zero and a standard deviation of one. The subtraction of the mean did not change the data by much, as the relative coordinates were already almost centered around zero, but the effect of having a unit-variance was very profound. It seemed to allow for the training of much bigger networks without having to deal with an exploding loss.

All datasets as well as the source code for the preprocessing are publicly available (see Section 4.4).

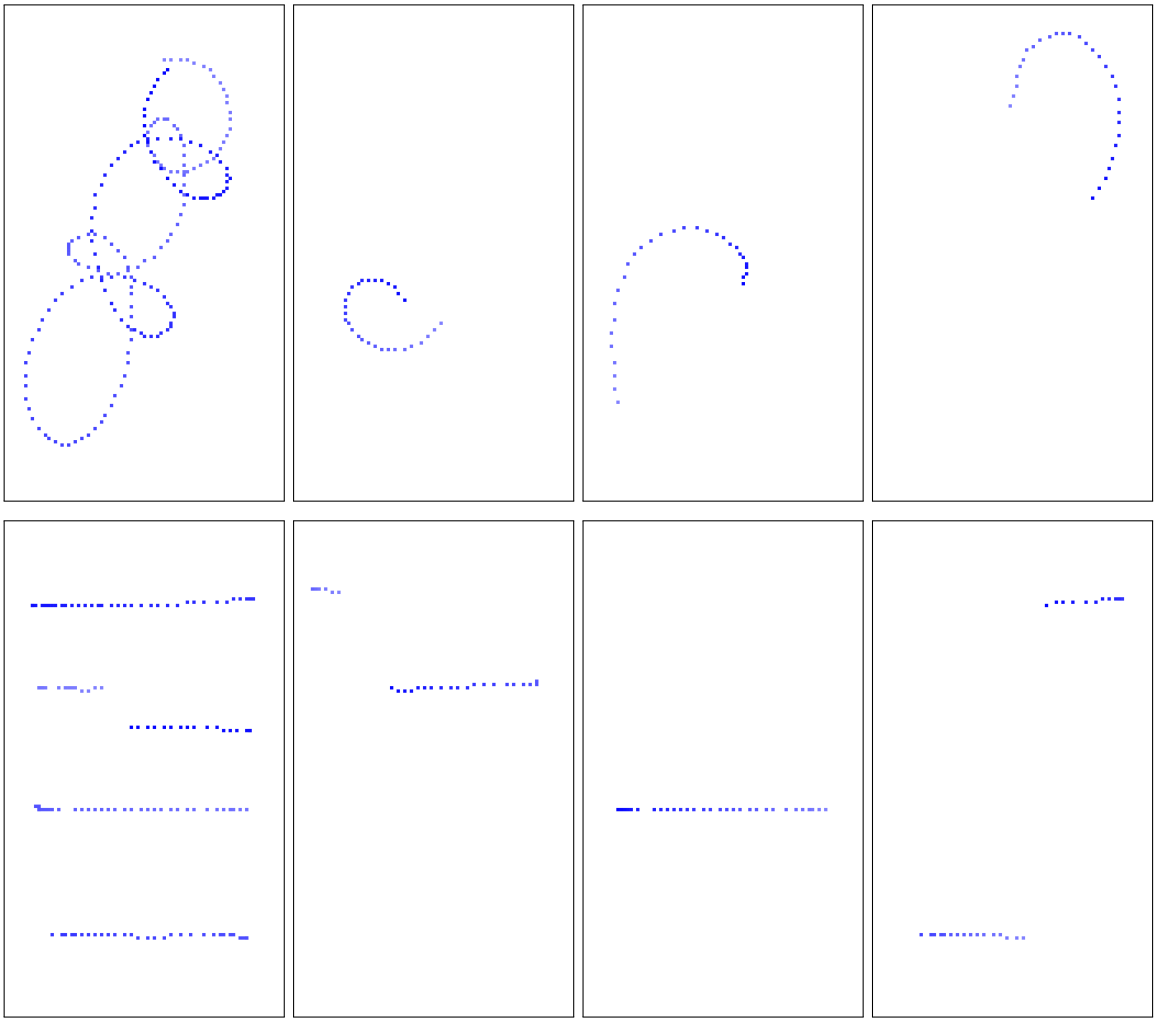


Figure 4: Example visualizations of two of the seven trajectories. Visualizations of the other five can be found in Appendix A. The leftmost image of each row displays the characteristic shape of a long motion. The other pictures of each row depict selected 30 point long subtrajectories, that are used as the actual input for the prediction models. The top row shows the *continuous circles dataset*, where the data points move back and forth diagonally. On the bottom are examples of the *horizontal lines dataset*, which features big jumps from left to right between each line and a big jump from the bottom to the top every four strokes.

4. Model architectures

The goal of this work is to find ways to make visually convincing - and not just quantitatively optimal - predictions for the continuation of human made touch trajectories on a touchscreen. We present two main models for this task. The first model is referred to as *Basic LSTM* and has a very simple architecture. It is described in Section 4.1. The second one is more complex and we call it the *LSTM-MDN*. It is described in detail in Section 4.2. We will then show results of experiments with an autoencoder architecture in the Section 4.3 and close out the chapter with the explanation of a few key implementation details in Section 4.4.

4.1. Basic LSTM

This network constitutes the most direct approach for the touch point prediction problem we could come up with if we were to use recurrent neural network elements, which have been proven to perform well for sequence prediction tasks. It can be seen as the baseline for more complex models in order to justify architectural additions.

4.1.1. Model architecture

The model consists of a single layer LSTM with 128 units that outputs the 2D coordinates of one point given a 30 point input sequence. It was implemented with version 2.0.8 of the deep learning framework Keras.

All relevant hyperparameters can be found in Table 2. The input to the LSTM is regularized by a dropout layer and the network contains in total 67.330 - 68.362 trainable parameters depending on the output size. The length of the input sequence appeared to be approximately optimal regarding the model performance and training time and was determined mainly by manual search. We did not change the default LSTM equations of this particular Keras version as Greff et al. [24] found that changes to that tend to not have a big impact. We lowered the default learning rate of the RMSProp optimizer, but left the other default values unchanged. The weights initializations were left unchanged as well. The hyperparameter optimization was not done as extensively as for the LSTM-MDN model, but we feel confident that we came rather close to an overall optimal configuration by using a similar set of parameters. This seems evident as the MSE results are better for the Basic LSTM network than for the LSTM-MDN network (see Section 5.3 for a full comparison). As we are working with multiple different datasets, we tried to find a hyperparameter configuration that worked reasonably well for all of them, as our goal is to find steps towards a general real-time prediction method.

All further result of the Basic LSTM network are always for the five point prediction variant.

4.1.2. Training phase

The set of subtrajectories of each dataset was split in 80%, 10% and 10% parts for the training, validation and test data sets, respectively. The method of obtaining the

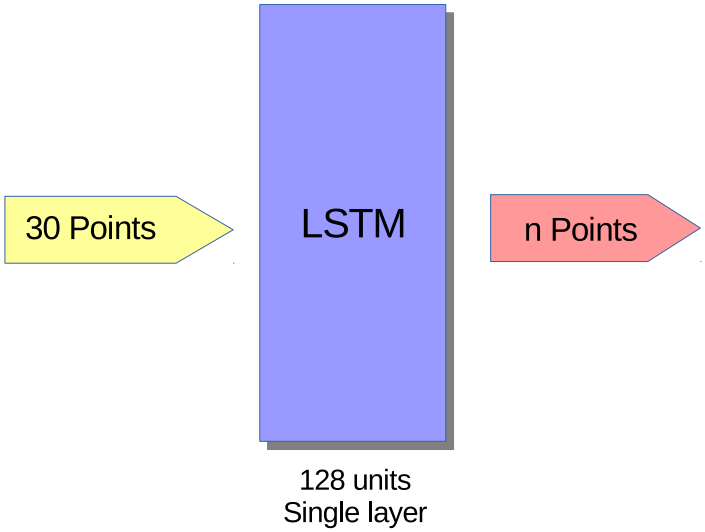


Figure 5: The model architecture of the Basic LSTM network during training. The network can be set up to predict an arbitrary number of consecutive points in the future as long as the data provides the possibility to do so.

Table 2: Hyperparameters for the training of the Basic LSTM network.

Parameter	Value
Loss	Mean absolute error
Optimizer	RMSProp [62]
Learning rate	0.0002
Gradient clip	10.0
LSTM input dropout	0.2
LSTM units	128
Input point length	30
Output point length	1 or 5
Epochs (max)	200
Early stopping	6 Epochs
Batch size	64

Table 3: Hyperparameters for the training of the LSTM-MDN network. Parameters are only mentioned if they are different from those in Table 2.

Parameter	Value
Loss	MDN loss (see Section 2.1.5)
LSTM output dropout	0.3
Number of components	5
Output point length	1
MDN layer init stddev	0.1
Minimum component chance	0.02

data is laid out in Section 3 and the learning followed the standard machine learning procedures described in Section 2.1. We used 30 consecutive touch points as input and the network was trained to predict either only the next one or the next five points at each training step. As the model only gets a limited number of points as input, it can not learn dependencies longer than that.

We used early stopping after 6 epochs if no improvements on the validation set were detected and saved the model with the best performance of all epochs. This model was then used for inference. Training for a dataset with a one point prediction converged on average after about 31 epochs, which took 13.24 minutes on average to be computed in total.

4.1.3. Inference

The goal of the inference process is to predict a 40 point continuation of a 30 point input sequence. The output sequence is predicted in an autoregressive manner by appending each predicted series of output points (either of length one or five) to its input sequence, while leaving out the starting sequence of corresponding length. The new sequence is then used to predict the next series of points and the process is repeated until a 40 point sequence is obtained. As the output sequence is longer that the input, the last 10 predicted points did only get previously predicted points as input. The normalization described in Section 3 is removed after the prediction is finished. This inference process is similar to the one used by Graves et al. [21] and took about two minutes for an averagely sized dataset.

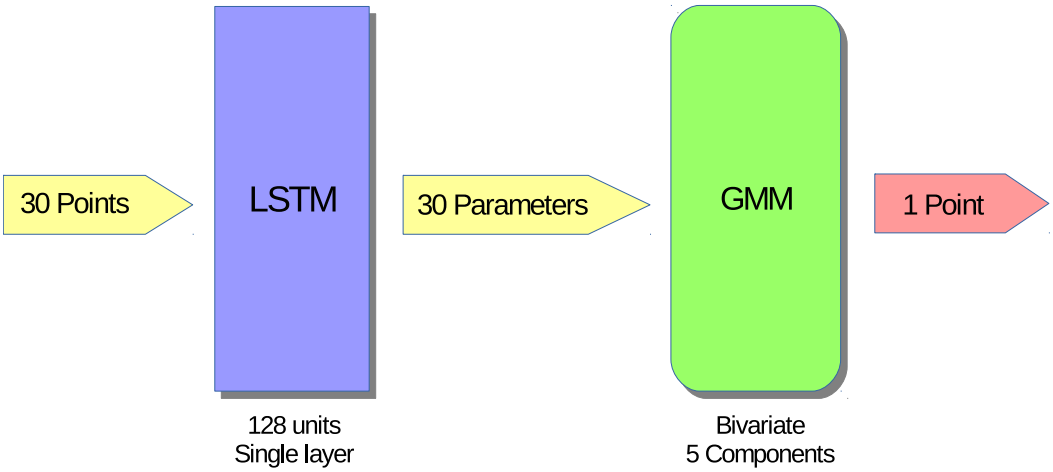


Figure 6: Model architecture for the LSTM-MDN network during training. The LSTM outputs the parameters for a bivariate Gaussian Mixture Model with five components, which is then used to sample the predicted point from.

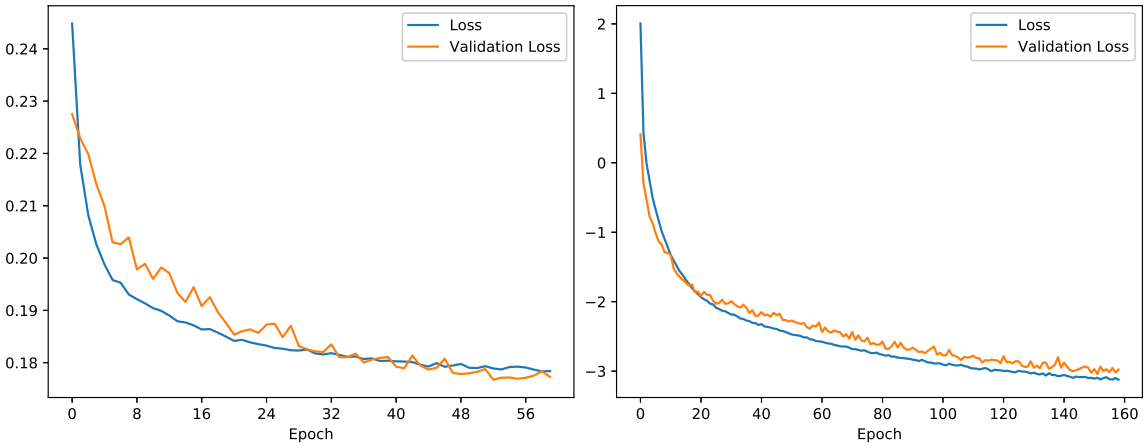


Figure 7: Loss history for the spaced rectangle dataset for the Basic LSTM network on the left and the LSTM-MDN architecture on the right. This dataset took the largest amount of epochs to converge for both networks. Both histories seem to show a convergence towards an asymptote with the network overfitting the data on the left for the first half of the epochs. In general, there seems to be a slight tendency to overfit the data, which is also prevalent for other datasets. This could be mitigated with further tuning of the regularization; perhaps by applying weight size penalties. The current application of dropout seems to be relatively close to optimal for the current hyperparameter setting and at least the rectangles dataset as Figure 11 suggests.

4.2. Long short-term memory with mixture density output (LSTM-MDN)

This model, which we will call LSTM-MDN for short, extends the Basic LSTM described in Section 4.1. It has a more sophisticated output in order to deal with the restrictions that come from only having the possibility to output an exact point sequence. We only describe elements of the architecture that are different from the Basic LSTM.

4.2.1. Model architecture

As shown in Figure 6, the LSTM layer of this model is trained to output the parameters of a bivariate Gaussian Mixture Model (GMM), which is then used to sample a predicted point from. This more indirect encoding of the prediction allows for the learning of complex contingencies. For example, when learning to predict discontinuous data, such as the horizontal lines dataset (see Figure 4 for a visualization), this model has the flexibility to output a *jump probability* for each prediction step.

This model architecture is similar to the model used for the handwriting prediction in Graves [21], except that it misses the explicit encoding of jumps and has fewer trainable parameters with 70.942 in total. Graves also used a method for the calculation of partially approximate gradients. This meant the model could receive input from up to 10.000 points in total, but the errors were only backpropagated to the start of each 100 point long parts, which the input sequence was made of.

The hyperparameters for this model are shown in Table 3. All hyperparameters were determined either by trial and error or by using a more complex hyperoptimisation (see Section 4.4.3 for more details).

4.2.2. Training phase

The training phase was very similar to the one described in Section 4.1.2. For this model, the additional application of dropout to the output of the LSTM layer seemed to allow for the training of bigger networks. The effectiveness of dropout outside of the recurrent transition is in line with the findings of Zaremba et al. [66]. The lowering of the default learning rate and choice of the optimizer also seemed to help with that. A slow down of the learning rate on a learning plateau (which in our case meant no significant validation loss improvements for three epochs) was found to have a slightly harmful effect on the learning results. Training for a dataset converged on average after 78 epochs, which took 18.1 minutes on average to be computed.

4.2.3. Inference

The inference process is conceptually the same as for the Basic LSTM (see Section 4.1.3), except for how exactly a point prediction is obtained. This follows the process described in Section 2.1.5. Additionally, we implemented a requirement for a minimum likelihood of a component to be selected to sample from. If this requirement is not met, we repeat the component choice until it is fulfilled. This helped mitigating effects, where

a very unlikely selection of a component would offset all successive point predictions due to the autoregressive inference process. To the authors best knowledge, this method was not used before and more details about the implementation of the MDN can be found in Section 4.4.1.

4.3. Autoencoder experiments

We experimented with incorporating the reducer part of a deep autoencoder into our Basic LSTM architecture. The idea was to leverage the low dimensional representations of the latent space to help the LSTM by explicitly preprocessing the long term input past in a meaningful way. We first experimented by training just an autoencoder to reproduce a 40 point long input sequences to see if meaningful representations would emerge in the latent space. The results for the seven datasets can be seen in Figure 8 and 9. We used a two layer encoder and a two layer decoder with a total of 14.348 trainable parameters and a two dimensional latent space. The activations were all softsign functions [42] and we did not utilize regularization as we did not find it to be helpful. The network was optimized to produce the most compelling representations, which meant a good separation of the data in the latent space.

We then built the architecture shown in Figure 10 to make use of those learned representations. The model did not perform as well as the Basic LSTM on the prediction task, which could be the result of two causes. Firstly, the representations in the latent space of the integrated reducer network were not as compelling as the ones of the pure autoencoder. We made a conscious decision to integrate it like that as we hoped that the network would then learn to reduce the long term input in a way optimized for the prediction task and not only for the reconstruction of the input. Maybe giving the deep reducer part a head start by taking the initial weights from a trained autoencoder or using convolutional layers could have helped to produce better representations. And secondly, there is a conceptual problem with this approach. The Basic LSTM network should be able to produce comparable representation in its hidden states if it has a 100 point input and it learns that those representations actually help the learning. In some sense using just an LSTM is a more general and thus presumably more powerful learning approach and we might have fallen in the trap of overengineering with this architecture.

4.4. Implementation

Every model was implemented in Python with Keras 2.0.8 on top of a TensorFlow backend. TensorFlow only supports a floating point precision of 32 bits, but we did not knowingly encounter problems with this precision constraint.

Our source code, the data and the trained main models are all publicly available on the website GitHub [2]. The README file in the repository contains information about how to test a pre-trained model or to start a new training.

[2] `https://github.com/soi/master_thesis`

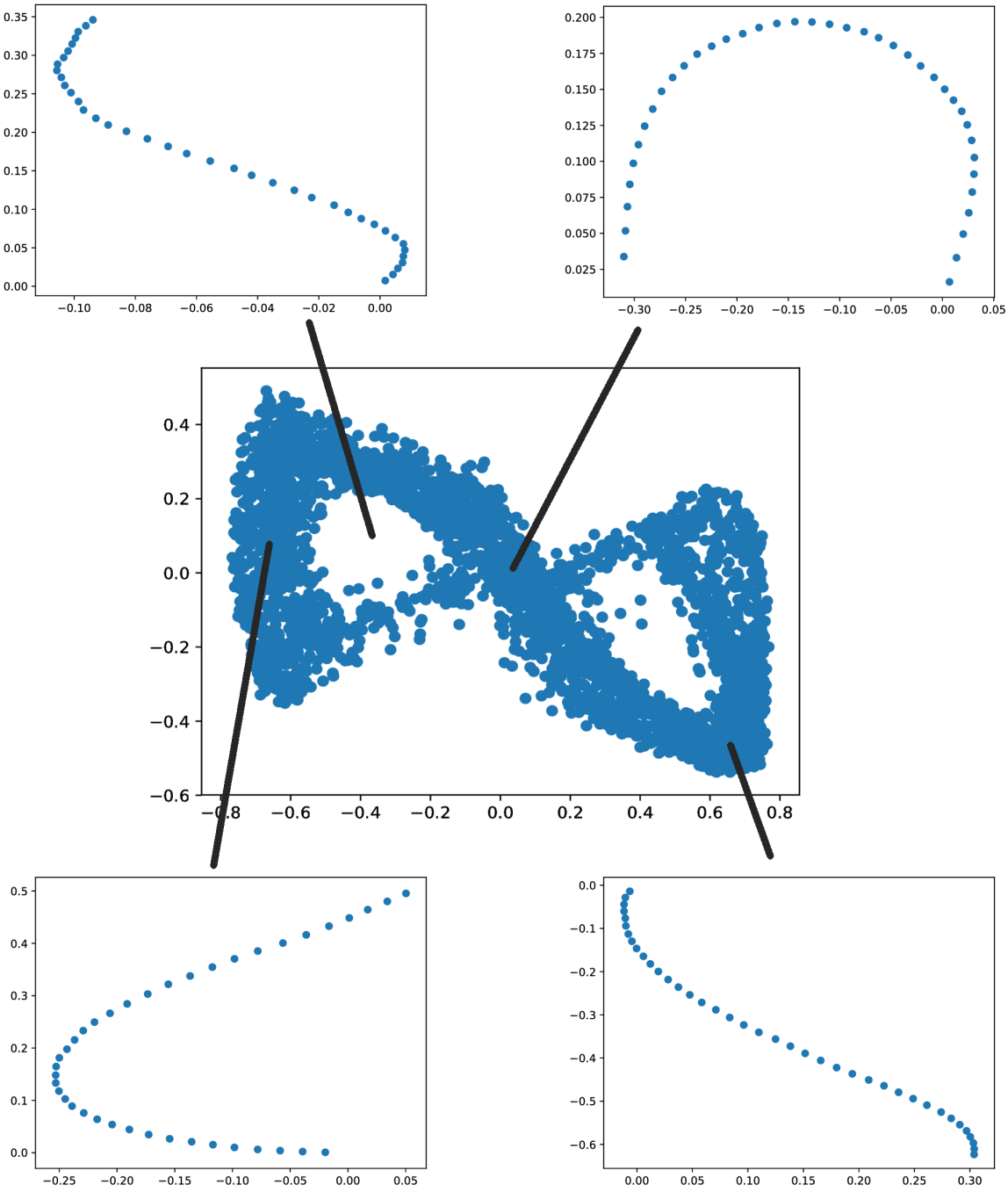


Figure 8: Latent space visualization for the dataset made up of repeated shapes of eights. In the center is the two dimensional latent space of all test set subtrajectories with example decodings on the top and on the bottom. We set aside 15% of the data to be used as the test set. For this dataset, that consists of trajectory parts of eights figures, the latent space emerges into an eight-like figure as well. This result was consistent for multiple trials except for small deviations in shape and orientation. The different positions in the latent space encode different trajectory types with similar trajectories being close to each other. The latent spaces for the other six datasets are shown in Figure 9. One of the videos mentioned in Section 4.4.2 shows a more detailed visualization of the latent space for the rectangles dataset.

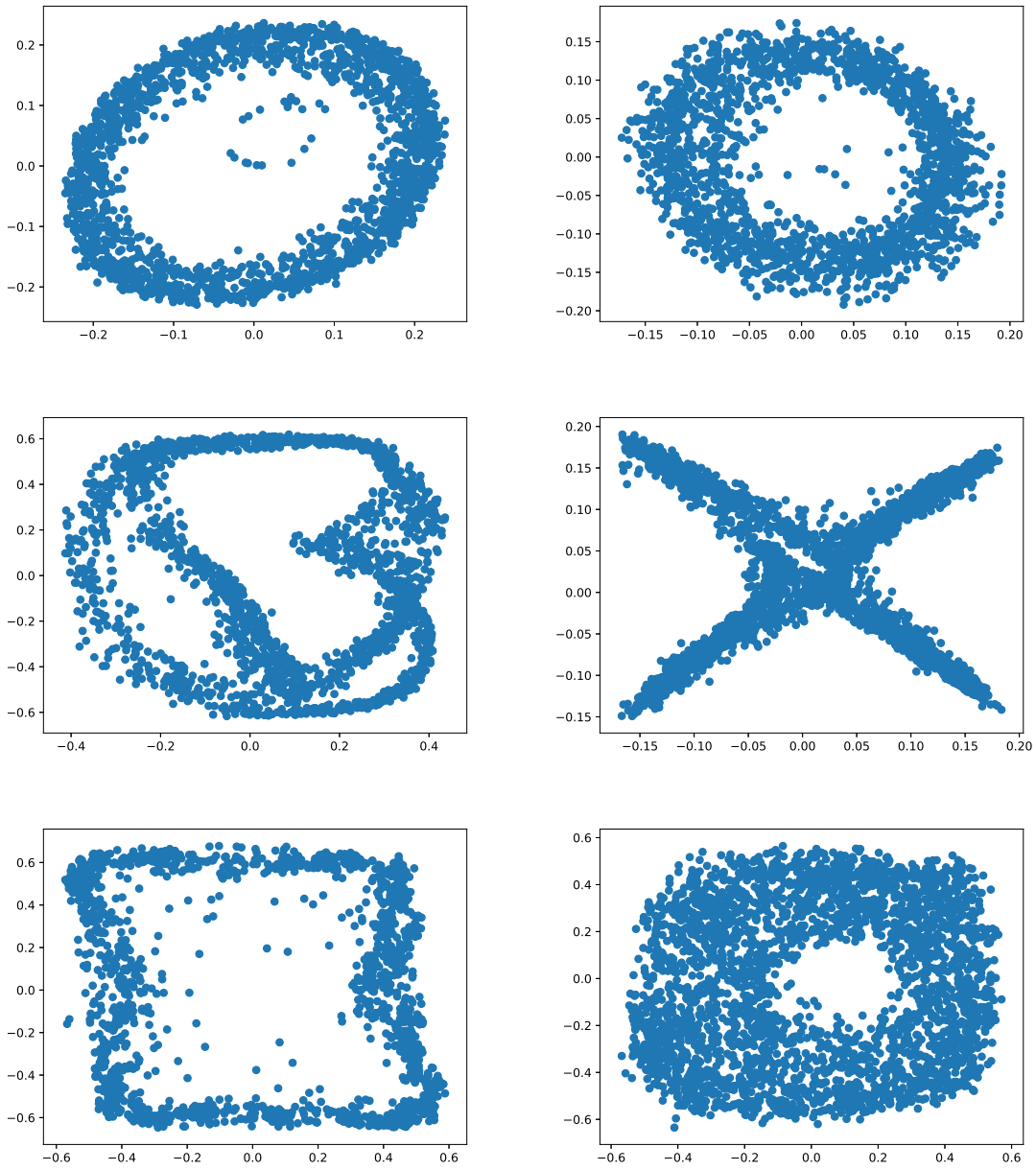


Figure 9: Latent space visualization for the test sets of all datasets except the eights dataset, which is shown in Figure 8. Top row: circles and continuous circles datasets. Both contain only curved shapes, which appear to lead to the emergence of a circular latent space separation. The right latent space shows a less clear separation as the data is more complex. Middle row: triangles/circles and rectangles. Both datasets lead to much different formations compared to the top row. The triangles/circles dataset, which presumably contains the most variety of shapes, also forms the most sophisticated latent space. Bottom row: spaced rectangles and horizontal lines. Both datasets are discontinuous and both seem to have trouble forming a clear separation of the input data with the left one showing many outliers and the right one being almost evenly distributed. The jumps in those datasets seem to lead to a smaller number of subtrajectories that are very similar.

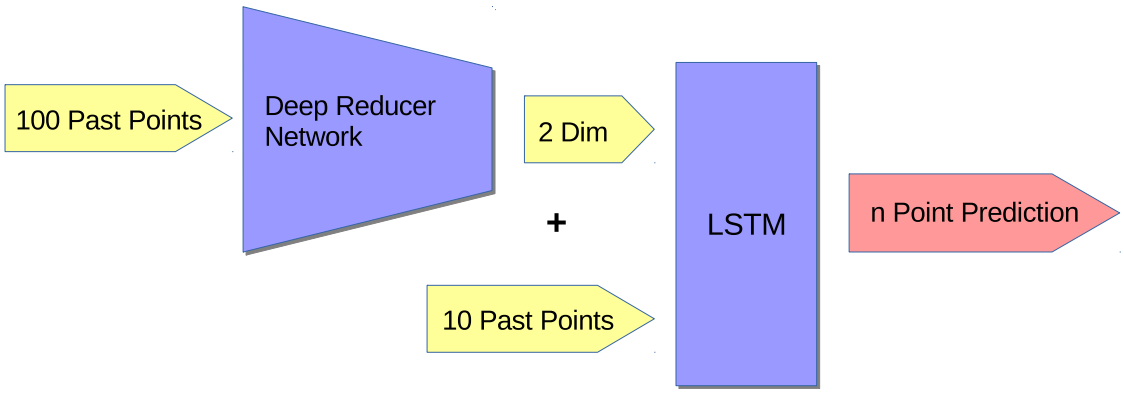


Figure 10: Basic LSTM architecture with deep reducer augmentation. The output of the reducer network was combined with each input point of the short term input sequence that goes directly into the LSTM. The final output is a series of points of arbitrary length.

4.4.1. MDN implementation

The implementation of the LSTM-MDN architecture, described in Section 4.2, builds upon the mixture density output implementation of the handwriting predictions of Graves [21] in TensorFlow by a person named hardmaru [25]. The loss implementation for our work resides in the file `mdn2d.py` inside the source code repository. We modified the code by removing the explicit jump encodings that were used in Graves' model. The whole mixture density implementation effectively consists of a loss implementation and a sampling mechanism, with the sampling being done in Python. The GMM output of the LSTM is implemented as a fully connected layer with 30 nodes.

During inference, the multivariate sampling included in Numpy did occasionally report that a covariance is not positive semidefinite despite all eigenvalues being positive. We did not find this to be a problem in practice.

4.4.2. Visualization

The prediction results and the training data were visualized with a custom GUI program implemented with TkInter[3]. The program is capable of displaying graphs alongside the prediction that contain additional information about the prediction process and saving the displayed information to disk. Example visualization of the two main models and the autoencoder experiments can be seen in three videos, that are attached to the thesis and have been uploaded to YouTube[4].

[3] `https://wiki.python.org/moin/TkInter`
[4] `https://www.youtube.com/watch?v=BfA3GDX1uDE&list=PLtlUploF7NprxZ9tT9hpYPoxb7tZWFbOS`

4.4.3. Hyperparameter search

The process of finding good hyperparameters for a given model concept can be crucial for the performance of the network. However, for this work we did not optimize for every single dataset as that would not be necessary for our goal laid out in Section 1. The search for close-to-optimal parameters was done by a combination of manual and automated search for a few select datasets. We usually started by defining very broad boundaries for the parameters we want to optimise together and then decided if it would be better to probe them on a linear or exponential scale. We then randomly sampled from this combined parameter space, evaluated the results and narrowed down the search space to where we felt a minimum could be. Using a combination of manual and automated search was found to be much more time-efficient than pure manual search.

Random search has been shown to be empirically superior to grid search [4]. While our parameters were aligned on a grid, we did sample from this space randomly and only computed the whole grid for the MSE landscape shown in Figure 11. This figure shows a rather smooth MSE surface with clear directions on where to find an approximate minimum. Only computing some random samples from such a grid would already give us a good hint on where to find good parameters. There is also some randomness involved in the learning process and the sampling of the predicted points when the LSTM-MDN model was to be optimized.

Some parameters were also determined purely by manual search.

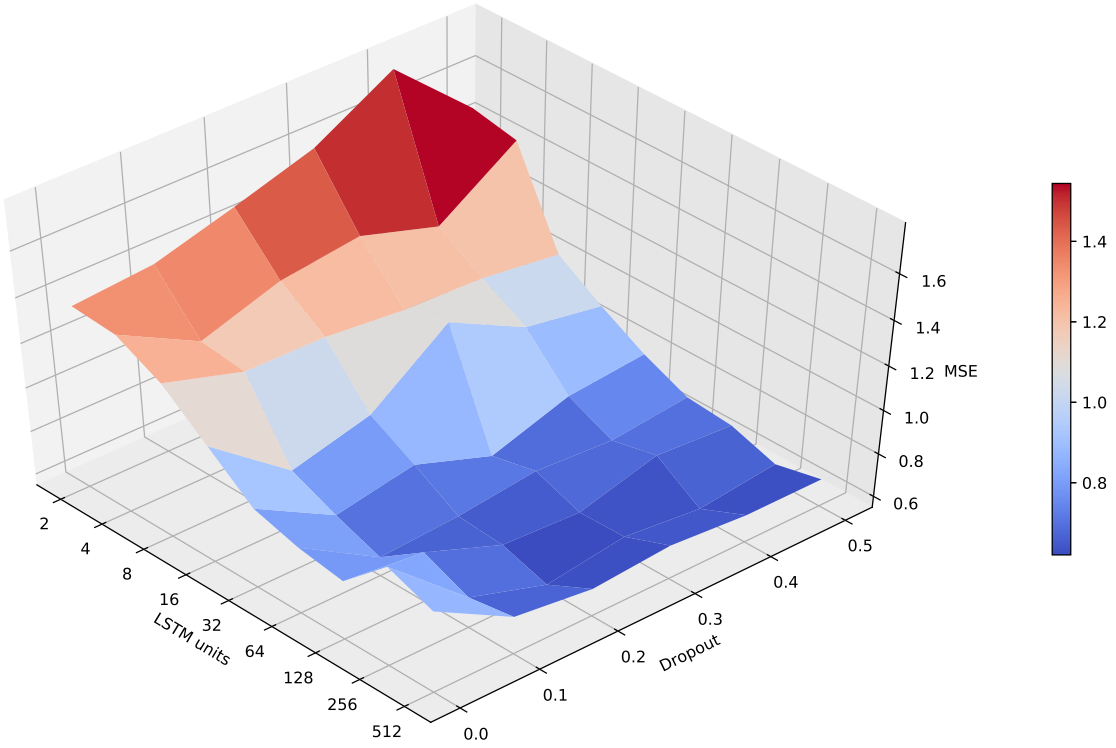


Figure 11: MSE landscape in the space of two hyperparameters of the LSTM-MDN model: the number of LSTM units and dropout applied to input and output of the LSTM with the same value. The LSTM units are on an exponential scale while the dropout values are aligned linearly. The training for this figure was stopped after four epochs without improvement instead of six when doing the training of the models evaluated in Section 5 to speed up the search process.

5. Experimental results

In this section we will first describe the performance of each of the main models, which were introduced in Section 4, and then proceed to compare them directly. We will give a comprehensive account of the MSE results for each model and dataset in Section 5.3. Further analysis will focus on key points underlined by a few selected examples from varying datasets due to the relatively big number of datasets and models to cover. We hope that through this process a comprehensive picture of the overall pros and cons of the models can emerge and potentially show promising steps towards a suitable solution for our goal of a general real-time touch predictor. The main focus of the more in depth analysis will be on the LSTM-MDN network as it has the more flexible architecture and seems to display much more complex behavior at least when analysing the output layer.

All quantitative errors are calculated as MSE per data point, which equates to the average squared Euclidean distance. This appears to be the same method used as in Graves [21].

5.1. Basic LSTM

This relatively simple model already performs very well on all datasets except for one major downfall: it is unable to predict big jumps in the touch trajectories. For all continuous datasets, the predictions are more convincing than those of the LSTM-MDN model both quantitatively (see Table 4) and visually (see Figures 17 and 18). However, this changes when the data is much more discontinuous. The direct output of one concrete point makes it so the network is punished twice for a prediction that does not occur at exactly the predicted point in the sequence. However, the error is guaranteed to spike only once if the network never predicts a jump as shown in Figure 18. The exact moment the human test person lifted his finger from the touch display can vary a lot, so the network can not predict the exact moment with a big enough certainty to justify to make those predictions in order to minimize the training loss. Section 5.3 provides further inside into this topic. Fundamental changes to the model architecture, especially in how the loss is calculated, are needed to change this behavior.

We evaluated two model variants that had a one and five point output training length, respectively. While the five point output was found to produce results with a slightly lower MSE for some datasets (see Table 4), we did not see big differences between the two variants. This is surprising as it would be expected that a prediction of multiple points at once would help mitigate escalating effects of prediction errors along the autoregressive prediction process. This might be an indication that the application of dropout on the input already accomplished a lot in terms of error tolerance. We also did not notice those effects when evaluating example prediction results for this network architecture.

5.2. LSTM-MDN

The LSTM-MDN model is set up to encode everything probabilistically, including jumps in the data. While it offers more flexibility for the prediction process, this also means that every prediction now depends on chance to some degree. This leads to an inferior MSE performance across all datasets (see Table 4). We tried to reduce the effects of this randomness by tweaking the learning and inference process as described in Section 4.2.1, but those actions only worked to some degree. The chances the network takes at each step of the prediction process inevitably lead to a few unlikely samples, which can then offset all consecutive predictions due to the autoregressive prediction process. There is also a lot of variance in how the LSTM learns to encode the characteristics of a dataset. While the behavior for one dataset is rarely completely different, it can vary a lot in detail due to the randomness of the initial weights and the shuffling of the training samples.

In the remainder of this section we will continue with an analysis of multiple different aspects of the behavior of the MDN. A possible future measure that could lead to an improvements is discussed in Section 6.1.2.

5.2.1. Changes in π value during prediction

The LSTM-MDN model displays great flexibility in the encoding of a point trajectory within its LSTM weights. This can be demonstrated by comparing two architecturally identical models that were each trained with different datasets. In Figure 12 we plot example predictions alongside the respective changes of the π values throughout the 40 point prediction process. One dataset is circular and continuous while the other one consists of linear and discontinuous shapes.

The top right of the figure shows that the GMM components all cyclically change in importance for the first dataset with three different components being the most likely ones to be selected at some point. It appears that some distributions became specialized to predict only certain parts of the trajectories and all of them are then weighted according to their individual usefulness for the prediction of a specific touch input step. However, not all components seem to have emerged to fulfill a specialized role as for example the blue and red components seem to be useful at similar trajectory parts. This is to be expected as we trained with a fixed number of components for each dataset and some datasets appear to offer the possibility for the emergence of more specialized distributions than others.

The GMMs show different dynamics when predicting for the horizontal lines dataset as displayed in the bottom row of Figure 12. Here, only one component is the most prevalent one and it is encoding the short distance steps from right to left that make up most of the data. But despite this dominance, at least one more specialized component emerged with the one displayed in red. This one has much higher standard deviations and means as it encodes the jumps in the data. A spatial visualization of the different densities that emerge for this dataset is shown in Figure 13. It can be seen in Figure 12 that the red component rises in importance together with the chance for a jump to

come, which is towards the start and end of the prediction. It takes a while until it gets selected despite the favorable change in selection chance. Once it is selected, the jump from the middle left to the lower middle right is predicted and its importance falls down rather quickly as two jumps right after another are very unlikely. Its chance to be selected rises again later in the predicted sequence when the time for the next jump comes closer. The blue component shows a similar pattern, but it does not encode the actual jumps in the data. We speculate that this component encodes the higher likelihood of a slowed down trajectory at the beginning and end of a stroke.

The π values are not the only parts of the predicted GMMs that change dynamically, but they usually appear to show the most drastic changes. This could be due to the fact that they are only single values of a GMM that can be adjusted to completely switch the way the network predicts. All of this behavior emerged just with BPTT through a vanilla single layer LSTM and a probabilistic learning goal in order to maximise the likelihood for the next point. Graves had similar results for the handwriting predictions [21].

5.2.2. Spatial Gaussian mixture model visualizations

In this section we will analyse combined visualizations of all GMMs used to predict a sequence. The examples are shown in Figure 13.

The dataset on the top left has very predictable point by point continuations and thus leads to the emergence of only very narrow densities. The pathway made up of the likelihood under the GMMs is much more smooth and therefore more similar to the data than the actual predicted point sequence. This is due to the sampling process, where not always the most likely point is predicted. This is one reason why the Basic LSTM outperforms the LSTM-MDN networks so drastically for the simpler datasets (see Section 5.3 for a direct comparison of the two).

On the top right is an example from the rectangles dataset. While it is continuous and very predictable in how it progresses linearly, this dataset has a higher variance when it comes to the switch in direction. This is expressed by the GMMs in their horizontal extends, which seem to be present for every point of the upwards part of the prediction. The GMMs will show more and more downward extend as the prediction reaches the typical horizontal end point.

The figure on the bottom left shows a prediction for the horizontal lines dataset. This dataset has similar properties to the previously described one, except that it features big jumps horizontally after every stroke. Every jump also comes with a vertical change, that can vary a lot as one out of four vertical changes crosses almost the whole screen. As described in Section 5.2.1 and shown in Figure 12, the LSTM learns to output specialized densities for regular point continuations as well as the jumps and weighs them accordingly. The extend of the jump encoding density is shown by the big shape on the right. This density has very high variances, which explain the frequent errors in the position of the jump predictions (see Section 5.2.3 for an analysis of false predictions). We speculate that this is due to the high variance in the vertical jump extend and the inability of the model to know where exactly it is on the screen

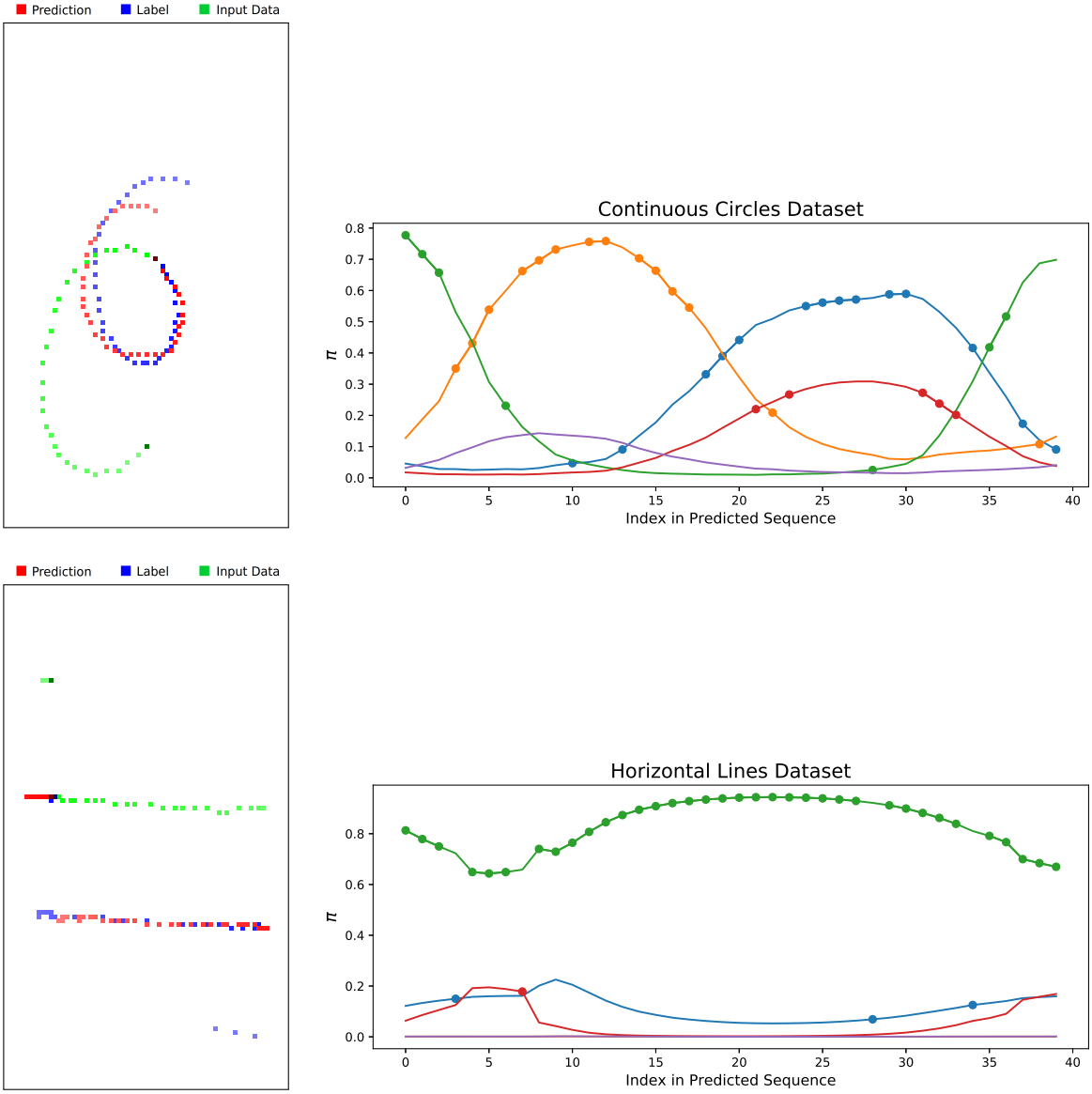


Figure 12: Predictions for two subtrajectories on the left and their corresponding GMM component selection chances on the right. On the left is the touch input trajectory in green, the actual continuation in blue and the predicted continuation in red. The opacity reduces with consecutive points. On the right are plots of the π values of the five mixture components for each step of the 40 point long prediction. A dot on a plotted line means that this component was selected to be the sampling distribution for the next predicted point. On the top is an example from the continuous circles dataset and on the bottom is one from the horizontal lines dataset. Both predictions have an above-average accuracy.

with only a 30 point input length and relative coordinates. A more sophisticated GMM output could emerge if the input length would exceed the typical shape length (see Table 1).

The visualization on the bottom right is from the spaced rectangles dataset, which appears to feature the most complex statistics. The trajectory has sudden horizontal and vertical changes in direction as well as jumps in both directions. The data appears to be very noisy in general as well. Those more complex statistics are evident by the high variances in the visualized GMMs with densities for the horizontal and vertical jumps being clearly visible. This dataset seems to be the most prone to very unreasonable predictions, as will be shown in the next section.

5.2.3. Predictions going wrong

So far we have analysed what usually happens inside the GMMs when trajectories are predicted, but what happens when predictions go completely wrong? Figure 14 shows some of the worst examples we could find for the respective datasets and Figure 15 provides some inside to what is happening during the prediction process for the bottom row examples. For the continuous datasets it was very difficult to find predicted trajectories that went completely off course. This is also to be expected as the GMMs for those datasets are much more narrow (see Figure 13). However, for the discontinuous datasets, predictions go wrong regularly. Some mistakes are so severe that we would not expect a human predictor to make them under normal circumstances. The bottom row of Figure 14 shows two examples of those.

On the left of Figure 14 is a prediction for the horizontal lines dataset. The predicted sequence starts on the top left and proceeds to jump after a few points (see Figure 15), which is reasonable. However, many more jumps are triggered closely after that, which scatter points all over the area. This is unlikely to happen so often in a row but it is not uncommon for a jump to occur soon after a previous one as the jump probability does not immediately decline to zero after a jump, which we feel like it should happen according to the data. The result is a scattered and offset trajectory, that only remotely resembles the correct one. Another common error is to never predict a jump as the chance to select the jump density rarely rises over 20%. The predicted sequences are then very similar to the predictions of the Basic LSTM network.

The right bottom shows a totally unreasonable prediction from the spaced rectangles dataset. At first it does a good job at matching the vertical line but then a presumably unlikely jump prediction sets everything off and the network appears to show chaotic behavior as it is confronted with a situation it has apparently not been trained for. This appears to be evident by the sudden changes in the π values of the output GMMs depicted in the bottom row of Figure 15. However, the network appears to reach a known state again around the prediction of point 30 when it starts to output a straight vertical line at the wrong place and time. This is common when predictions go wrong for this dataset. The overall result is the prediction of a shape that is not even remotely in the training data.

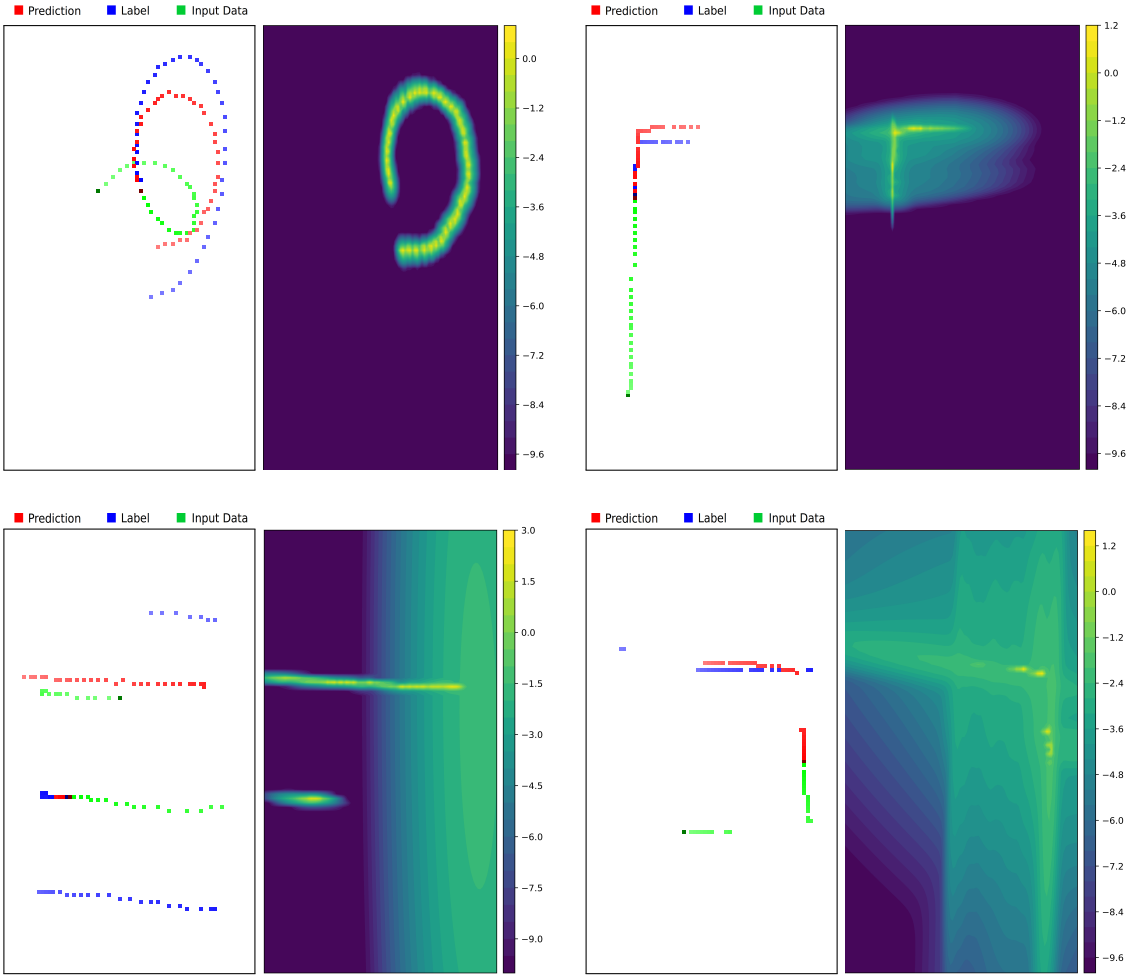


Figure 13: Visualizations of the cumulative density of all GMMs for one predicted sequence for the extent of the touch input area. On the left of the four subplots are example predictions of four different datasets. On the top are the continuous datasets made up of continuous circles and rectangles and on the bottom are two examples of the two discontinuous datasets with horizontal lines to the left and spaced rectangles to the right. On the right side of each subplot are the cumulated densities on a logarithmic scale. The yellow parts usually follow the predicted red points on the figure to the left. The color bars denote the log_{10} value of the combined density function for each point on a 100x100 raster. We transformed the values logarithmically as the individual distributions can vary a lot in their extent and we found this to be a convenient way to display all of them in one picture.

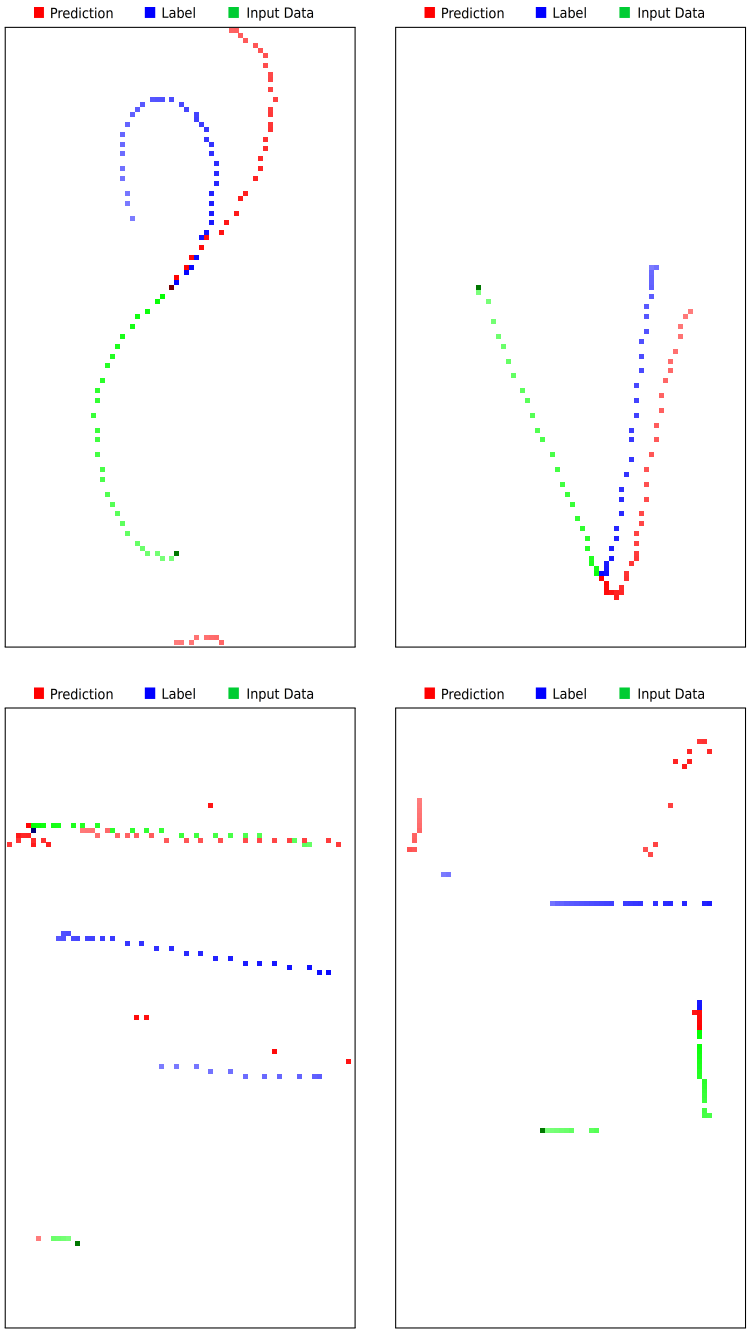


Figure 14: Four plots of predictions going wrong. We tried to find the most drastic examples for each of the datasets. On the top are two examples from the continuous eights and triangles/circles datasets. On the bottom are examples of the two discontinuous datasets. Predictions that surpass the border are wrapped around. The change of the π values for the bottom predictions is displayed in Figure 15.

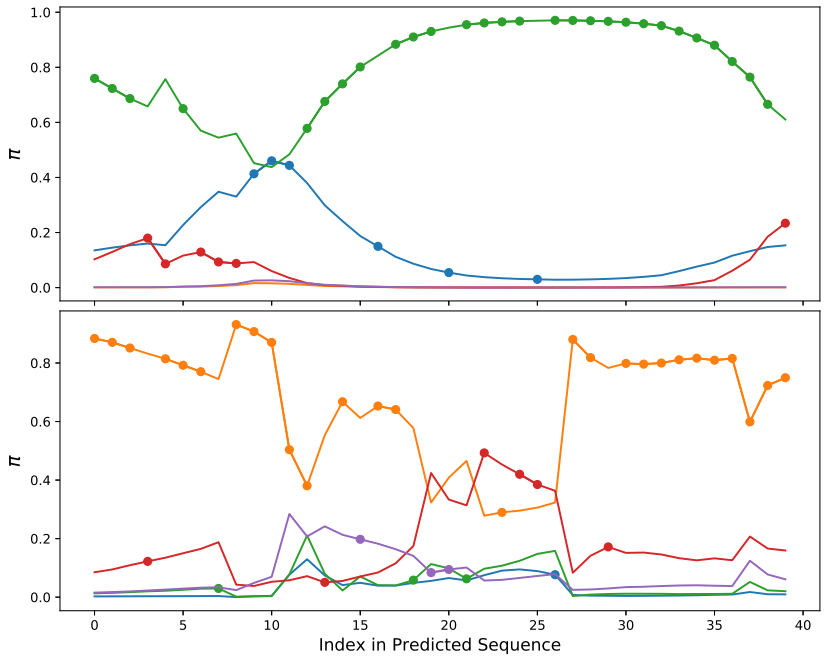


Figure 15: Change in π value for the GMMs that made up the predictions in the bottom row of Figure 14. On the top is the plot for the example of the horizontal lines dataset and on the bottom the one with the spaced rectangle parts. A dot on a plotted line means that this component was selected to be the sampling distribution for the next predicted point.

5.3. Comparison of main models

Table 4 shows a comparison of the MSE for all main model archetypes. As stated in the previous sections of this chapter, both models have their strengths and weaknesses. The Basic LSTM model does a better job at predicting for the continuous datasets, while the LSTM-MDN architecture allows for an encoding of finely granulated contingencies that lead to jump predictions for subtrajectories of the discontinuous datasets. It does so by using a very different loss approach. In this section we compare both models visually and quantitatively.

The Figures 17 and 18 show example predictions of both models for the same data. For the circles dataset, the Basic LSTM network is more convincing when looking at shapes as well as at the numbers. The plots of Figure 17 show that both models struggle to predict the rather big jump on the middle right of the blue trajectory, as the input speed increased due to the 2/3 power law. However, the LSTM-MDN also has a lot of relatively incorrect predictions towards the end. The error in MSE appears to match the visually perceived error quite well.

The situation is different for the horizontal lines dataset. Here, the MSE is lower for the predictions of the Basic LSTM network, but the actual shapes can be much more convincing visually for the LSTM-MDN architecture. This is due to the high likelihood of a double spike in MSE when going for a jump prediction. This also shows, why the Basic LSTM never learned to predict the jumps: as it was trained to minimize a similar measure with the mean absolute error it is more optimal to never predict a jump and live with only one high error spike. The LSTM-MDN model on the other hand tries to optimise the likelihood of the prediction under the data through a GMM and not for the MSE or a similar measure. Looking at it this way, it is not surprising that the Basic LSTM network outperforms the LSTM-MDN when judged by a very similar metric than the one it was trained for.

While the difference in MSE for both model archetypes can be big, the models share similarities in how the MSE changes throughout the predicted sequences. Figure 16 shows that the MSE steadily rises in similar fashion with subsequently predicted point for the eights dataset. While the absolute errors are higher for the LSTM-MDN model, it shows that both architectures have approximately linearly growing difficulties to predict further into the future when this metric is used for this dataset. The overall pointwise MSEs for the other continuous and circular datasets show similar properties. However, for the discontinuous dataset displayed on the right of the figure, the MSE does not change by much over time except for a significant low for the first few points. While the MSE of the LSTM-MDN shows more variance, the overall error stays approximately even for both cases. This is due to the domination of the jump prediction errors, which are far more significant for the overall average error than growing uncertainties over time. This is also evident by the MSE plots in Figure 18. The results for the other discontinuous dataset are very similar.

The similarities of both models in their development of the MSE over time might be due to statistical properties of the data such as a growing variance in point position for further away points in time.

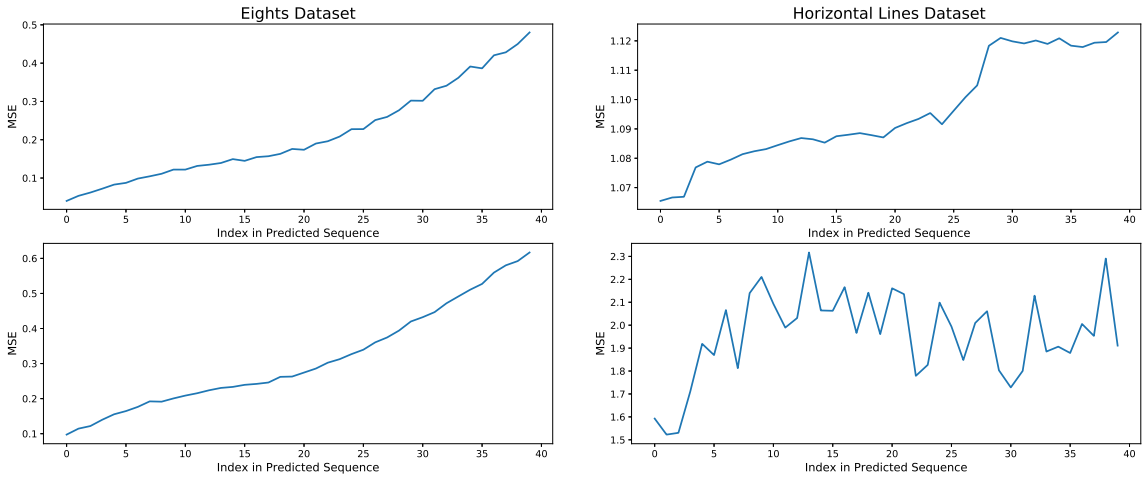


Figure 16: MSE for the complete test sets of the eights and the horizontal lines datasets. On the top of each column is the error plot of the Basic LSTM model. On the bottom are the graphs for the LSTM-MDN models.

Table 4: The MSE for the 40 point long predictions for both architectures and each dataset. Results for the Basic LSTM network are reported for training setups to predict either one or five points simultaneously. If this metric is used, the Basic LSTM architecture performs better with all versions for all datasets than the LSTM-MDN models. The five points variant outperforms the single point one for two datasets and breaks about even for the others.

Dataset	Basic LSTM		LSTM-MDN
	One point	Five points	
Circles	0.1027	0.0975	0.1625
Continuous circles	0.2552	0.2474	0.3086
Eights	0.2095	0.2128	0.3181
Horizontal lines	1.0948	1.0957	1.9697
Rectangles	0.4686	0.3769	0.6457
Spaced rectangles	0.9617	0.9553	1.5686
Triangles/circles	0.1640	0.1088	0.1989

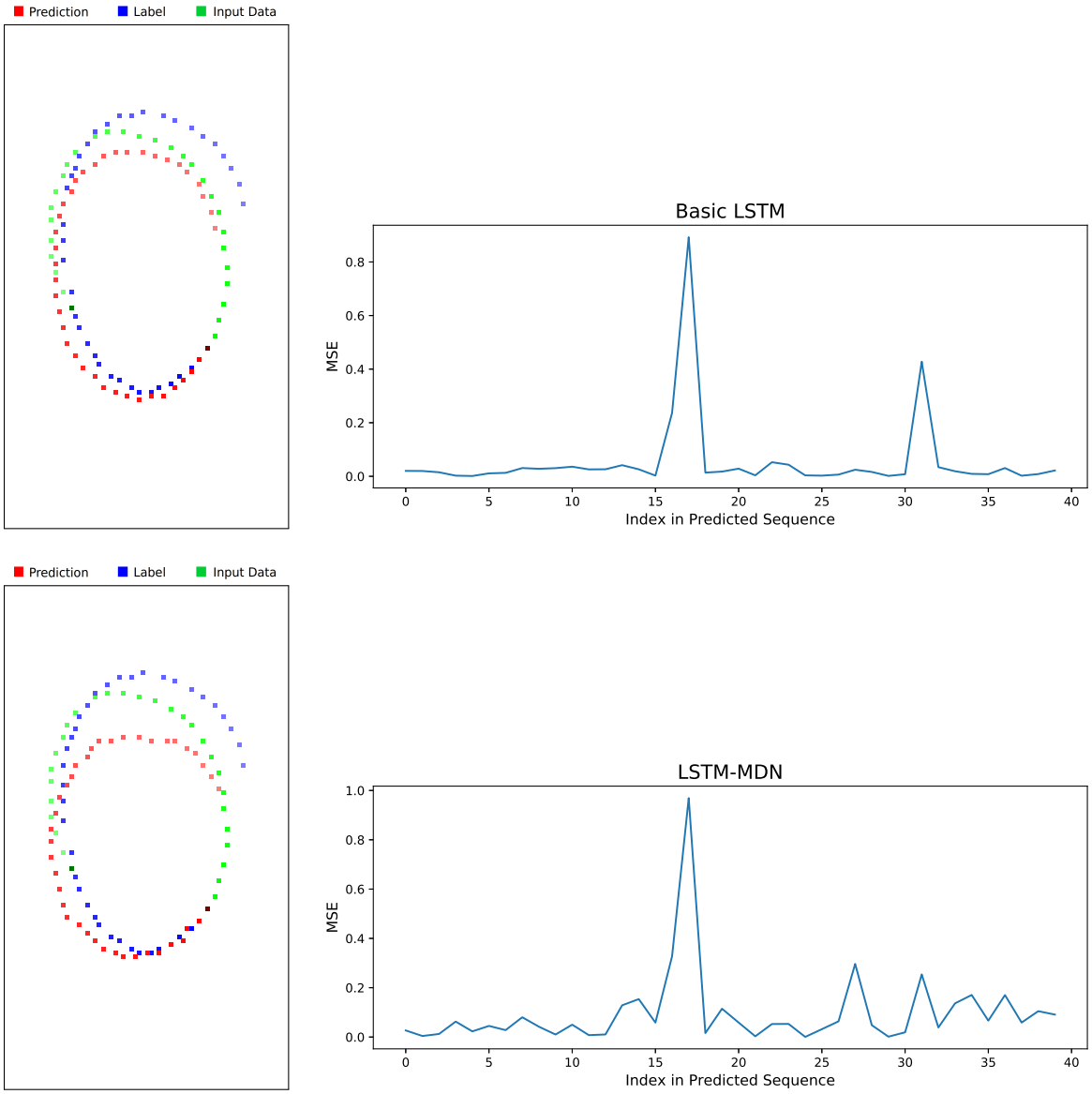


Figure 17: Comparison of two predictions for the same input sequence of the circles dataset by the Basic LSTM model on the top and the LSTM-MDN model on the bottom. On the left are the visualized trajectories. On the right are the corresponding plots of the data pointwise MSE. The MSEs are shown with different vertical axes and the total MSE is 0.056 for the Basic LSTM output and 0.097 for the output of the LSTM-MDN. The visual and quantitative discrepancy appears to be slightly higher than average, but not untypical for this dataset. For a comprehensive overall MSE comparison see Table 4. One of the videos mentioned in Section 4.4.2 shows multiple predictions of the Basic LSTM model for the continuous circles dataset together with the respective error.

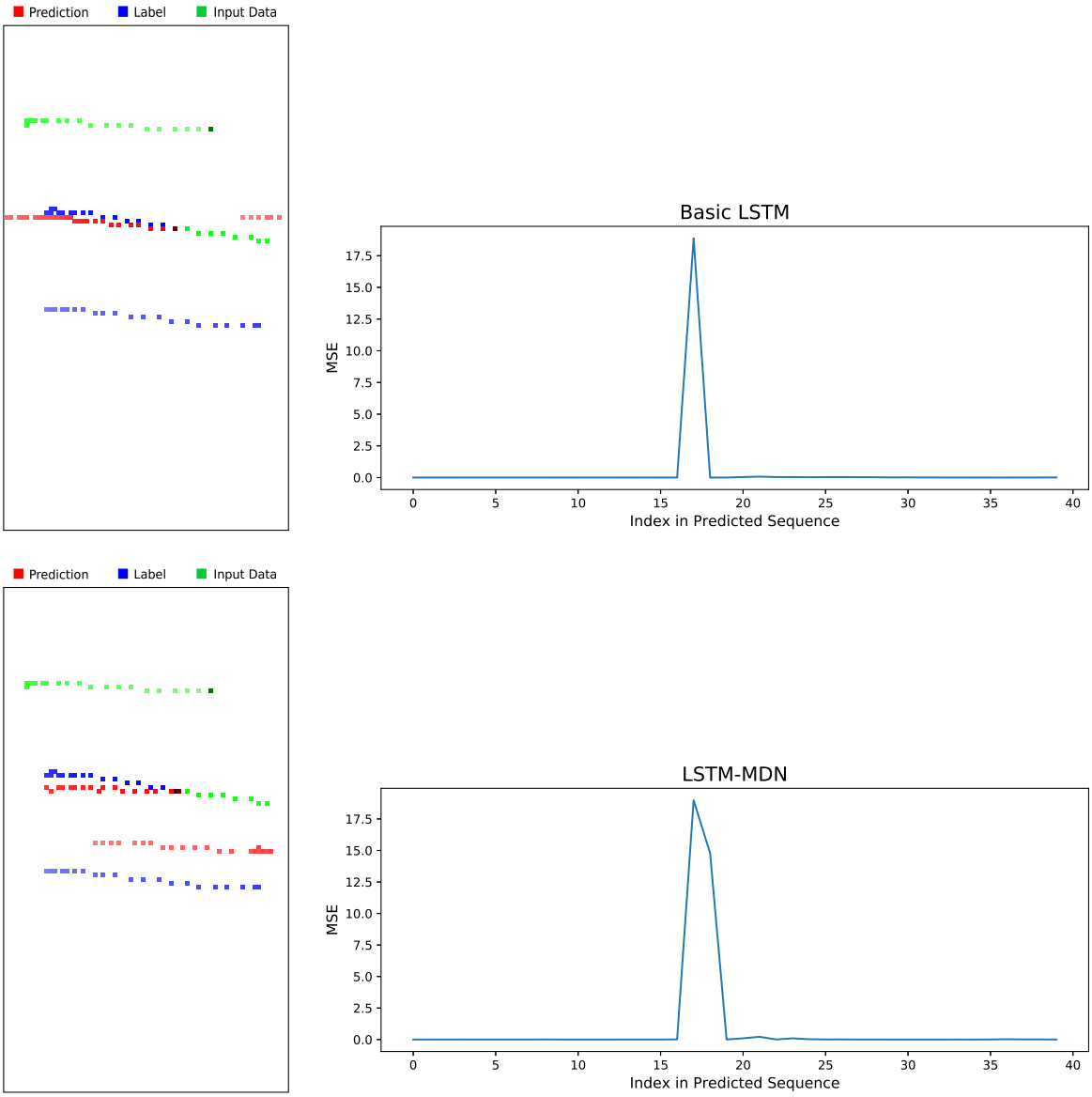


Figure 18: Similar comparison to Figure 17, but for a subtrajectory of the horizontal lines dataset. The Basic LSTM output has a MSE of 0.483 and the one of the LSTM-MDN output is 0.859. The MSE plots on the right are dominated by the errors of a jump prediction on a wrong moment or a missed jump prediction. As it is very unlikely to hit the exact jump moment of the data, the LSTM-MDN usually has two spikes in MSE for such predictions and the Basic LSTM has just one as it never predicts a jump. This leads to a situation where the MSE error discrepancy does not match with the visual error impression as the predictions on the bottom seem to be more convincing. One of the videos mentioned in Section 4.4.2 shows multiple predictions of the LSTM-MDN model for this dataset.

6. Discussion

The goal of this work was to find ways towards real-time touch predictions on a robot equipped with artificial skin with modern machine learning techniques. We evaluated multiple architectures for different toy datasets and two architectures emerged as the most promising. Of those two, the Basic LSTM has a more straight forward architecture and has shown the most accurate predictions (both visually and quantitatively) for continuous and in particular also circular datasets. The LSTM-MDN, on the other hand, showed more flexibility, as it was able to incorporate big jumps in its predictions, something that the Basic LSTM was not capable of. However, this came at the price of frequent sampling errors, small and big, leading in to some extent tolerable cases to not-so-straight curvatures and in the worst cases to completely wrong predictions. Those false predictions could then also offset subsequent inference processes, which sometimes resulted in totally unreasonable future trajectories.

Overall, we found that the most critical point of each architecture was how the loss was calculated. This is understandable, as this is the very origin of the signal that makes the networks adapt and it is the main difference in both architectures. This observation surfaces the main common problem: their respective losses do not directly push the network into the exact direction we want them to move. We want them to produce the most visually convincing predictions and nothing else. But a mean absolute error only cares about single point distances and not about the predicted trajectory as a whole and the MDN loss leads only to an optimization of the likelihood of a correct single point prediction. The key, it seems like, would be to find a loss that was obtained through an evaluation that is much closer to what we humans do when we would judge a trajectory prediction. We will discuss possible ways to do so in Section 6.1.1.

We also evaluated the usage of an autoencoder for dimensional reduction of a time series of touch input data and tried to incorporate this capability into a prediction architecture. While the latent space representations seemed promising, we were not able to make a combined architecture outperform the Basic LSTM. This was probably in part due to the much more complex architecture with many more hyperparameters to tune and also due to the general kind of approach to the problem that we used here. Instead of letting the network decide how to reduce the input, we hand-engineered a lot of that in with our addition of the deep reducer part. The Basic LSTM would be, in theory and given an input of the same size, capable of doing similar reductions on its own. But it would only learn to perform them if they are useful for the learning task. This end-to-end learning approach has been a common trend in many machine learning research fields ranging from language translation [60] to the learning of control policies in robots [36] and we have come to the conclusion that it would probably be the right approach to take for our problem as well.

Coming back to the long-term goal laid out in Section 1, we think that using either of the main networks would be a good starting point to set up touch inference in real-time, depending on the complexity of the input data. Both networks have shown the ability to encode complex statistics over time, which can be used to predict a mid-sized trajectory with reasonable average accuracy. While the Basic LSTM would

be preferred for simpler movement, the LSTM-MDN could be used for more complex touch input if the likelihood of very wrong predictions can be reduced (see Section 6.1.2 for a possible approach). It is also unlikely that the effect of those sampling errors would be as pronounced as in the setup of this work, as the predictions in a real-time setting would hardly need to be 40 points long in order to predict between two touch input interrupts. Also, the input could be much longer, which could lead to the encoding of more complicated statistics, although it would also increase the need for training data and possibly require a different set of network parameters. We will revisit the topic of real-time deployment in Section 6.1.6.

We also feel confident that the networks would be able to encode useful statistics of more natural movements as they already have shown great flexibility when dealing with our - arguably rather constrained - datasets (see Figure 13 for example). Training with more arbitrary input would likely require much more training data to capture the more complicated statistics as well as additional hyperparameter tuning.

This work was motivated by the predictive coding theory and we feel that our architecture shows many aspects of it. The short term goal was to predict spatiotemporal touch information and we did this with architectures that clearly show the ability to model statistics on different time scales (see for example Figure 12). However, it is not clear how this understanding emerged and if a hidden hierarchical dependency structure is present inside the LSTM weights. Further investigation of the dynamics of the hidden LSTM states would be required to illuminate this. As such a structure is not known, it is not possible to perform explicit top-down simulations. All changes of our models during learning are also driven solely by the prediction error, which is very typical for deep learning applications.

6.1. Possible future work

Future work could go into two general directions: either improving the prediction models in the offline setting that we worked with here, or by incorporating them into a robot and working with a real-time prediction setup. We will focus mainly on the former option throughout this section as this work was concerned with an offline setting and a concrete prediction setup on a robot in real-time would be highly dependent on computational and morphological contingencies.

Many of the possible directions in this section are non-exclusive.

6.1.1. A different loss function

One of the main take-aways from this work is that, if the predictions of either of the main networks were to be improved, a different loss function appears to be the most promising avenue for further development. As stated before in the introduction of this chapter, we conclude that a loss is needed that is able to give a more fitting feedback on the primary objective of producing visually convincing continuations.

A possible avenue for this is using an *adversarial loss*, a method popularised by Goodfellow et al. [19]. Networks that use such a loss are called *Generative Adversarial*

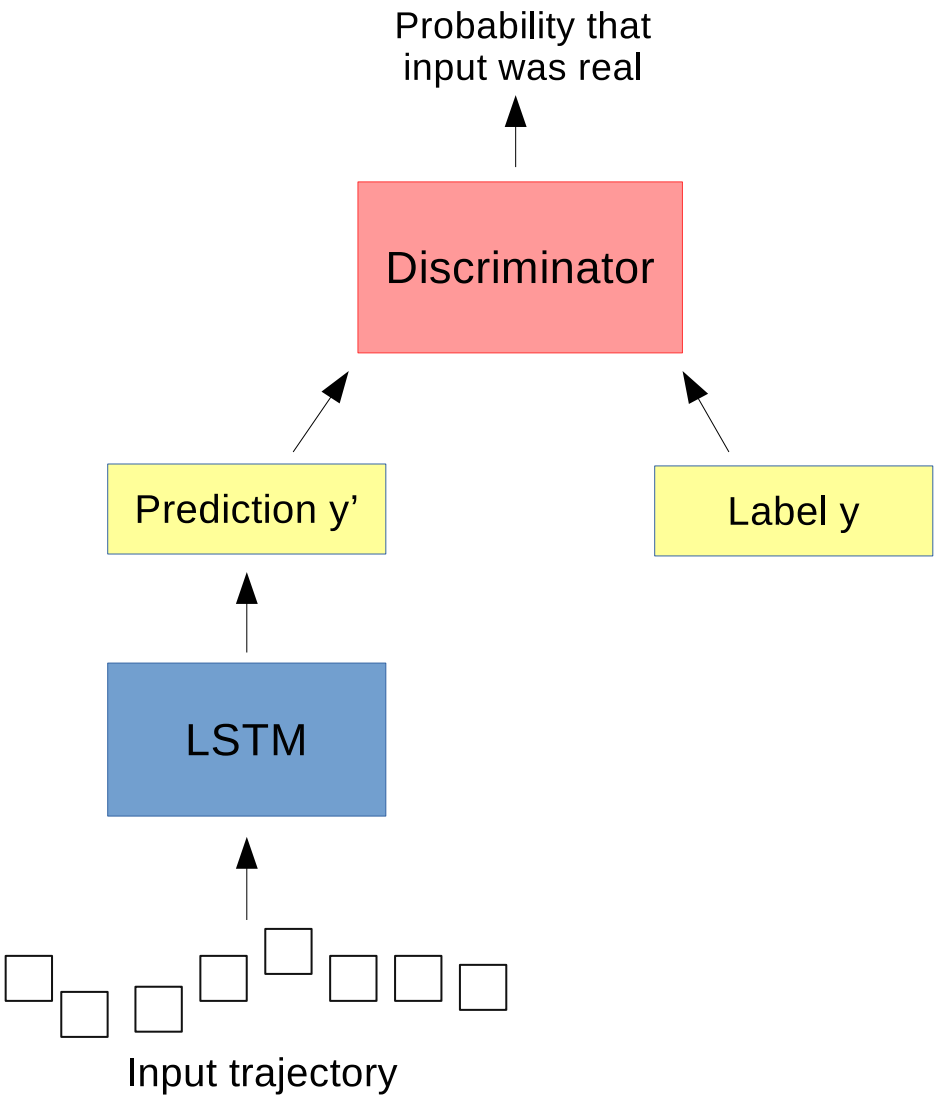


Figure 19: Sketch of a possible GAN architecture for time series predictions. The discriminator represents an evolving loss function and gets presented either a prediction or a label at random. This kind of architecture would be an interesting avenue to explore in our opinion, but it is highly speculative if it would work out.

Networks (GANs) and they evolve the learning process from a simple loss optimization into a two player game. One player is a neural network that outputs predictions (or, as it is common with GANs, generates new data such as pictures) and is called the *generator* or G. The other one is the *discriminator* or D and the task of this second network is to learn to distinguish generated data from real data. For that it is exposed to both at random during the learning phase. The generator G now has to try to fool D while both get better at their respective tasks. This has the advantage, that G is not just trying to optimize a rather simple, hand-chosen loss function but rather a complex and automatically optimized function represented by D. D can be discarded once the training is finished. GANs can be difficult to train but they have shown great success in generating new pictures from greatly reduced representations [47]. The adversarial loss can also be used in conjunction with a more traditional loss like the MSE in order to crisp up visual predictions [38].

Figure 19 shows a rough idea for a GAN architecture. While this seems to make sense in theory, it is not unlikely that it will not yield the desired results due to its complexity and the fact that GANs have so far mainly been proven useful for image generation from random data. Both the Basic LSTM and the LSTM-MDN could be used with this setup, as it is possible to backpropagate through the sampling process as shown by Graves [22]. While it would be possible to backpropagate through the autoregressive prediction process, it would probably be difficult to maintain a meaningful gradient signal through it, especially with the LSTM-MDN. Also, the input length for the prediction network would ideally be as long as for the labels or at least long enough to contain all necessary statistics to make a comparable prediction (which might for our datasets correspond to the typical shape length shown in Table 1).

6.1.2. Scheduled sampling

One key problem of the LSTM-MDN architecture is the negative effect of sampling errors on subsequent predictions. This is due to the fact that the LSTM has not been trained to deal with its own sampling errors but rather was always fed correct data during training. While the application of dropout on the input and the implementation of a minimal selection chance for the choice of a component of a GMM helped with this problem, it still appears to be responsible for many predictions going wrong (see Section 5.2.3).

Bengio et al. [2] proposed a possible solution for this problem, which they call *scheduled sampling*. It works by gradually replacing more and more true tokens with self-sampled ones during the training phase to make the network more robust against its own errors when being in inference mode later on. This method has since been found to be useful in some circumstances [7, 12, 48], while not yielding any improvements in others [6]. This technique can be readily added to the current training setup and could be used in conjunction with the previously mentioned mitigation methods.

6.1.3. Data augmentation

Data augmentation is a way of enhancing the training dataset by adding transformations of the original training samples to it. When working on image classification for example, it could make sense to flip every image horizontally while preserving the label, so the model learns to incorporate the statistics of such pictures as well. If applied in the right way for a specific dataset, it can lead to better generalization of the learning and can thus be seen as a form of regularization. It is done extensively when working with images [26, 33] but it can also be applied to time series data, for example when working on speech recognition [1].

Data augmentation would make a lot of sense for the training of our models if the data were more complex and would for example come from unconstrained human-robot interactions. In this case, it would be possible to increase the invariance of the learning under rotations by creating many rotations of the long input trajectories. Also, the scale invariance could be increased by scaling the shorter subtrajectories by many different factors. Applying those transformations would increase the training data a lot and could be a step towards making the learning more robust. The learning is already invariant under translations of the trajectories (assuming no limits of the input medium) as we train it only with relative coordinates.

6.1.4. More complex touch data

Our model could be extended to use more complex touch input data. A possible way to deal with such data would be to convert the input and output to a picture and use *convolutional* layers [35] in the prediction model. Networks that use such layers are called *Convolutional Neural Networks* (CNN). These layers consist - in short - of multiple filters that are applied spatially over a picture to extract basic features like lines and curves. Adding additional convolutional layers on top of each other can lead to the detection of increasingly more complex structures if the detection of those are helpful for the learning task. The effect of generating a reduced representation of an image can also be reversed with *deconvolutional* (deCNN) layers. Combining a LSTM with CNN and deCNN layers for sequential image predictions has been done before by Lotter et al. [38].

Using CNN layers would also be useful when extending the model to use a *touch area* instead of just a point. Many touchscreen devices at least provide the parameters of an ellipse that surrounds the touched area.

Touch devices also often provide pressure information, which can be recorded at the exact same time as the spatial touch information. Using pressure data together with regular touch information would be multimodal learning (see Section 6.1.5).

It would also be possible to include *multi-touch* information in the data and the predictions. The processing could be done with CNN layers or in a multimodal setting that is similar to the possible addition of pressure information. Using a CNN-deCNN architecture would scale up to any desired number of simultaneous touches while preserving the input and output sizes. However, those marginal sizes would also need

to be much bigger in general.

All extensions mentioned in this section are non-exclusive.

6.1.5. Offline multimodal learning

It is a lot easier to infer if a busy street is free to be crossed with both visual and auditory information available. We, as humans, are not only able to predict the information that comes to us through one sensory channel but through many at the same time. While the prediction of one information stream can be useful, the correlation of multiple ones can lead to a more rapid and deeper understanding of the state of the world and has been shown to influence the understanding of speech [39] and improve the learning for students [14].

Machine learning with previously recorded multimodal data has been shown to improve performance [43] as well as being somewhat resistant to noisy [10] or missing data [58]. A possible future avenue for this work could be to develop a multimodal model that incorporates touch input together with other sensory channels in order to predict future inputs of one or multiple channels at the same time. Additional sensory channels could be implemented as additional values of each LSTM input vector, that represent one time step.

6.1.6. Real-time predictions

The overall goal of this work was to find small steps, with the help of the tools of deep learning, towards the final goal of having a embodied artificial agent that is driven to interact and learn about the world just so it can improve the accuracy of its multimodal sensory predictions. This goal appears to be very far off but there have been some improvements in that direction. Roncone et al. [50] make use of visual and tactile input on an iCub humanoid robot [41], that is equipped with artificial skin, for specific developmental purposes. Hwang et al. [31] have built a hierarchical and multimodal predictive coding-inspired architecture around a simulated iCub robot that tries to predict its incoming visual and proprioceptive information.

As mentioned in the beginning of this chapter, we assume that our model can be adapted to a real-time inference setting (unimodal or multimodal) without big fundamental changes to the model itself. It should also be possible to endow the model with means to deal with missing data as it could simply be trained with some values in input vectors missing from time to time. This should be especially useful when working in a multimodal setting.

LSTMs have shown to be useful for real-time predictions [44] and when dealing with data from a simulated robot [11]. However, it would be more difficult to add the capability for online learning to the system, although the learning would not be strictly constrained to work in real-time as well. It would also be necessary to investigate how the prediction network would be able to adapt to constantly changing input statistics.

References

[1] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[2] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179, 2015.

[3] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[4] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[5] Christopher M Bishop. Mixture density networks. 1994.

[6] Samuel R Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D Manning, and Christopher Potts. A fast unified model for parsing and sentence understanding. *arXiv preprint arXiv:1603.06021*, 2016.

[7] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 4960–4964. IEEE, 2016.

[8] Andy Clark. *Surfing uncertainty: Prediction, action, and the embodied mind*. Oxford University Press, 2015.

[9] Christopher M Conway and Morten H Christiansen. Sequential learning by touch, vision, and audition. In *Proceedings of the Cognitive Science Society*, volume 24, 2002.

[10] Andreas Eitel, Jost Tobias Springenberg, Luciano Spinello, Martin Riedmiller, and Wolfram Burgard. Multimodal deep learning for robust rgb-d object recognition. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 681–687. IEEE, 2015.

[11] Zackory Erickson, Alexander Clegg, Wenhao Yu, Greg Turk, C Karen Liu, and Charles C Kemp. What does the person feel? learning to infer applied forces during robot-assisted dressing. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 6058–6065. IEEE, 2017.

[12] Chelsea Finn, Ian Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. In *Advances in Neural Information Processing Systems*, pages 64–72, 2016.

[13] Anna Flagg and Karon MacLean. Affective touch gesture recognition for a furry zoomorphic machine. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction*, pages 25–32. ACM, 2013.

[14] JD Fletcher and Sigmund Tobias. The multimedia principle. *The Cambridge handbook of multimedia learning*, 117:133, 2005.

[15] Karl Friston. The free-energy principle: a unified brain theory? *Nature Reviews Neuroscience*, 11(2):127–138, 2010.

[16] Paolo Gastaldo, Luigi Pinna, Lucia Seminara, Maurizio Valle, and Rodolfo Zunino. A tensor-based approach to touch modality classification by using machine learning. *Robotics and Autonomous Systems*, 63:268–278, 2015.

[17] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural Networks*, 4(1), 2008.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[19] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[20] Alex Graves. Sequence transduction with recurrent neural networks. *arXiv preprint arXiv:1211.3711*, 2012.

[21] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[22] Alex Graves. Stochastic backpropagation through mixture density distributions. *arXiv preprint arXiv:1607.05690*, 2016.

[23] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[24] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 2016.

[25] hardmaru. Handwriting generation demo in tensorflow. `http://blog.otoro.net/2015/12/12/handwriting-generation-demo-in-tensorflow/`. Accessed: 10/03/2017.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[27] H von Helmholtz. Concerning the perceptions in general. *Treatise on physiological optics,*, 1866.

[28] Geoffrey E Hinton and Richard S Zemel. Autoencoders, minimum description length and helmholtz free energy. In *Advances in neural information processing systems*, pages 3–10, 1994.

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

[30] Dana Hughes, Nicholas Farrow, Halley Profita, and Nikolaus Correll. Detecting and identifying tactile gestures using deep autoencoders, geometric moments and gesture level features. In *Proceedings of the 2015 ACM on International Conference on Multimodal Interaction*, pages 415–422. ACM, 2015.

[31] Jungsik Hwang, Jinhyung Kim, Ahmadreza Ahmadi, Minkyu Choi, and Jun Tani. Predictive coding-based deep dynamic neural network for visuomotor learning. *arXiv preprint arXiv:1706.02444*, 2017.

[32] David C Knill and Alexandre Pouget. The bayesian brain: the role of uncertainty in neural coding and computation. *TRENDS in Neurosciences*, 27(12):712–719, 2004.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[34] Francesco Lacquaniti, Carlo Terzuolo, and Paolo Viviani. The law relating the kinematic and figural aspects of drawing movements. *Acta psychologica*, 54(1):115–130, 1983.

[35] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[36] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[37] Marcus Liwicki and Horst Bunke. Iam-ondb-an on-line english sentence database acquired from handwritten text on a whiteboard. In *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*, pages 956–961. IEEE.

[38] William Lotter, Gabriel Kreiman, and David Cox. Unsupervised learning of visual structure using predictive generative networks. *arXiv preprint arXiv:1511.06380*, 2015.

[39] Harry McGurk and John MacDonald. Hearing lips and seeing voices. *Nature*, 264(5588):746–748, 1976.

[40] Geoffrey McLachlan and David Peel. *Finite mixture models*. John Wiley & Sons, 2004.

[41] Giorgio Metta, Giulio Sandini, David Vernon, Lorenzo Natale, and Francesco Nori. The icub humanoid robot: an open platform for research in embodied cognition. In *Proceedings of the 8th workshop on performance metrics for intelligent systems*, pages 50–56. ACM, 2008.

[42] Rohit Mundra and Richard Socher. Cs 224d: Deep learning for nlp. lecture notes: Part iii. spring 2015. `https://cs224d.stanford.edu/lecture_notes/LectureNotes3.pdf`. Accessed: 10/01/2017.

[43] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 689–696, 2011.

[44] Anh Nguyen, Thanh-Toan Do, Darwin G Caldwell, and Nikos G Tsagarakis. Real-time pose estimation for event cameras with stacked spatial lstm networks. *arXiv preprint arXiv:1708.09011*, 2017.

[45] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[46] Lutz Prechelt. Early stopping-but when? *Neural Networks: Tricks of the trade*, pages 553–553, 1998.

[47] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[48] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732*, 2015.

[49] Rajesh PN Rao and Dana H Ballard. Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature neuroscience*, 2(1), 1999.

[50] Alessandro Roncone, Matej Hoffmann, Ugo Pattacini, Luciano Fadiga, and Giorgio Metta. Peripersonal space and margin of safety around the body: Learning visuo-tactile associations in a humanoid robot with artificial skin. *PloS one*, 11(10):e0163713, 2016.

[51] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

[52] Asma Saleem, Khadim Hussain Asif, Ahmad Ali, Shahid Mahmood Awan, and Mohammed A Alghamdi. Pre-processing methods of data mining. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 451–456. IEEE, 2014.

[53] Stefan Schaal and Dagmar Sternad. Origins and violations of the 2/3 power law in rhythmic three-dimensional arm movements. *Experimental Brain Research*, 136(1):60–72, Jan 2001.

[54] Shouhei Shirafuji and Koh Hosoda. Detection and prevention of slip using sensors with different properties embedded in elastic artificial skin on the basis of previous experience. *Robotics and Autonomous Systems*, 62(1):46–52, 2014.

[55] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150, 1995.

[56] Michael W Spratling. Unsupervised learning of generative and discriminative weights encoding elementary image components in a predictive coding model of cortical function. *Neural Computation*, 24(1):60–103, 2012.

[57] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[58] Nitish Srivastava and Ruslan R Salakhutdinov. Multimodal learning with deep boltzmann machines. In *Advances in neural information processing systems*, pages 2222–2230, 2012.

[59] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. Revisiting unreasonable effectiveness of data in deep learning era. *arXiv preprint arXiv:1707.02968*, 2017.

[60] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[61] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[62] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[63] Daryl Weir, Simon Rogers, Roderick Murray-Smith, and Markus Löchtefeld. A user-specific machine learning approach for improving touch accuracy on mobile devices. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 465–476. ACM, 2012.

[64] Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, 1(4):339–356, 1988.

[65] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

[66] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[67] Heiga Ze, Andrew Senior, and Mike Schuster. Statistical parametric speech synthesis using deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 7962–7966. IEEE, 2013.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den October 31, 2017 .....

A. Additional trajectory visualizations

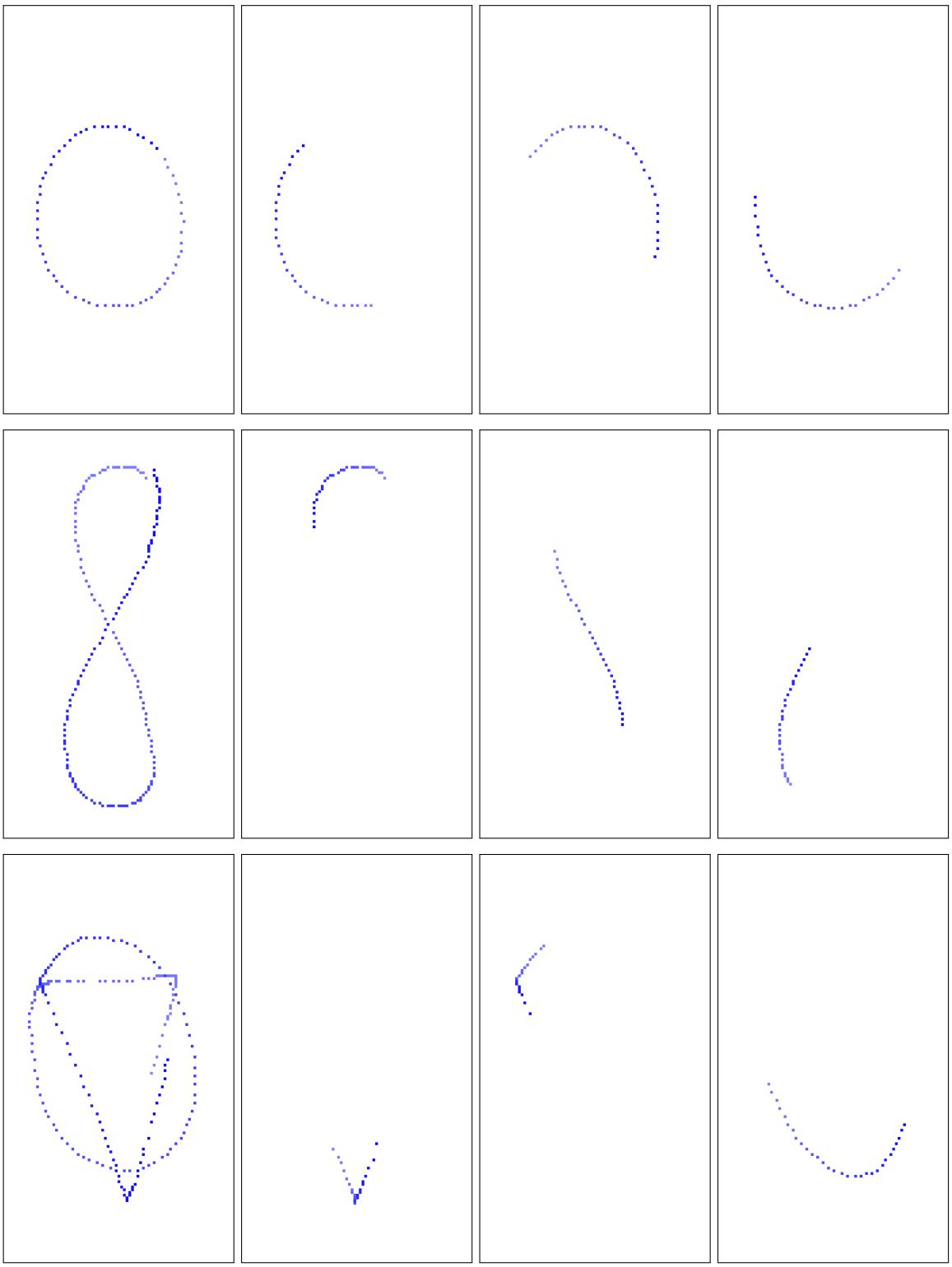


Figure 20: Trajectory visualizations of three of the seven dataset. The other visualizations can be found in Figure 4 and Figure 21. The top row shows the *circle dataset*, which is made up of curved and continuous movements and has the shortest typical shape length (see Table 1 for a comparison). It also seems to display the least amount of variance across its shapes. The *eights dataset* in the middle row has similar properties, except for a longer shape length. On the bottom are examples of the *triangles/circles dataset*, which is continuous and consists of a combination of curved and linear shapes. It has the longest typical shape length.

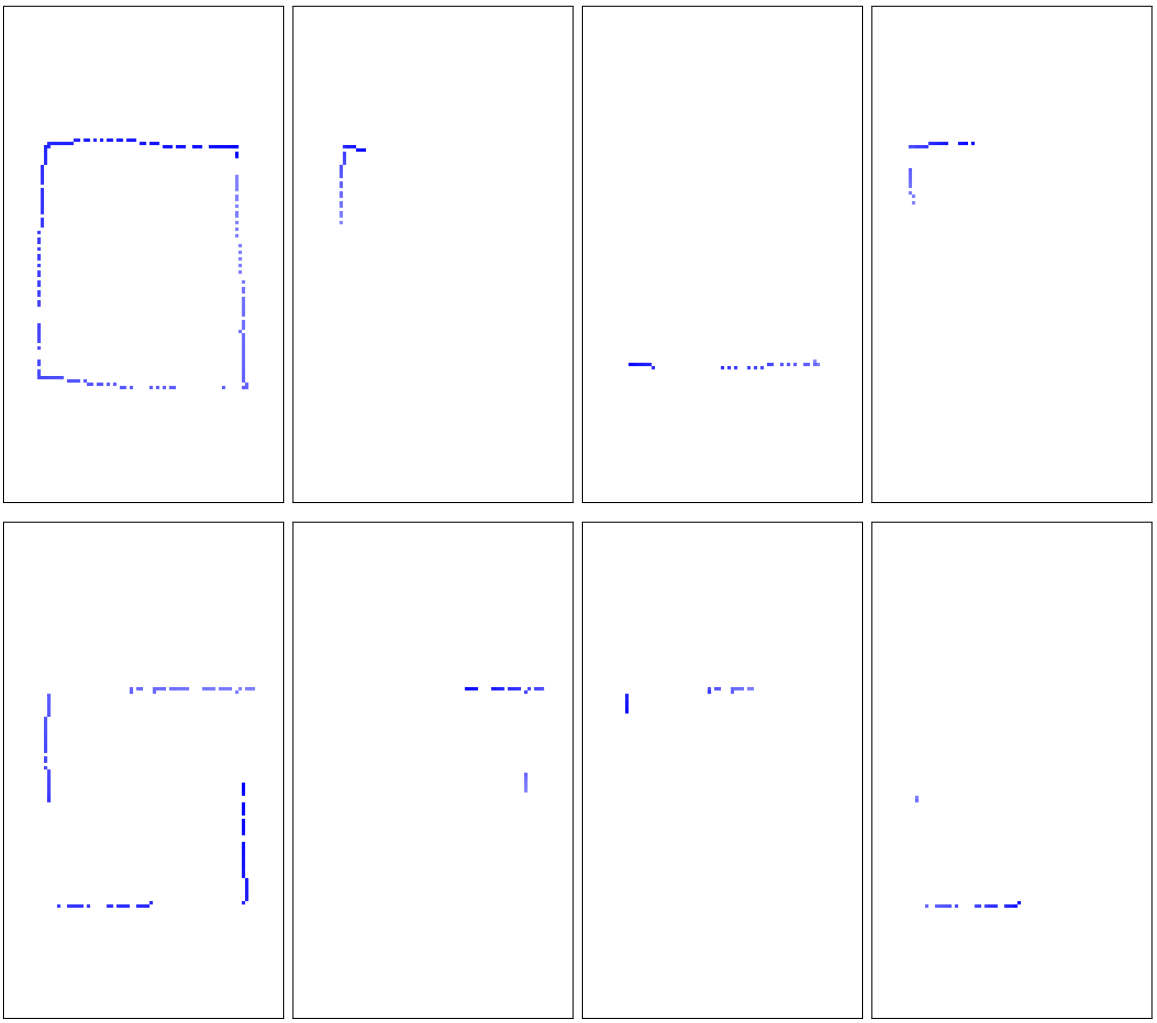


Figure 21: Trajectory visualizations of two of the seven dataset. The other visualizations can be found in Figure 4 and Figure 20. The top row shows the *rectangles dataset*, which features shapes from a continuous and approximately linear input movement. Despite the movement being continuous, the input finger has to come to a full stop at the sharp corners, resulting in an arguably more complex input pattern than what e.g. the circles dataset is made of. The circles dataset is displayed in Figure 20. The bottom row shows the *spaced rectangle dataset*, which is similar in shape to the rectangles dataset, except that each line is only drawn approximately half the way. This results in big jumps in space and direction and also produces a lot of noise as it is very difficult for a human to always draw the very short lines in the same lengths on a display with a diameter of 12.5 cm. Both datasets seem to have more missing data points than other datasets, which makes their data even more complex.