

# **Applying LSTM neural networks to biological cell movement**

Project Report

by Johannes Rieke

(johannes.rieke@gmail.com)

31 March 2016

Lehrstuhl für Biophysik

Friedrich-Alexander-Universität Erlangen-Nürnberg



Supervisor: PD Dr. Claus Metzner

## **Abstract**

Neural networks with Long Short-Term Memory (LSTM) were used on scientific time series data. Each time series contains the positions of a biological cell while it moves through one of three environments (a collagen network, a plastic surface, or a plastic surface coated with fibronectin). The networks were used for two tasks: 1) Classifying the movement trajectories based on the cell environment. Several networks of increasing complexity were trained on parts of the trajectories, using softmax classification. The best networks achieved an accuracy of ~95 % (on test data) and generalized well to longer trajectories. 2) Generating new movement trajectories by predicting one step of a time series after another. For this purpose, LSTM was combined with the idea of a mixture density network (MDN): It does not predict the values of the next time step directly, but outputs the parameters of a mixture distribution, from which they can be sampled. The generated trajectories replicated the shape as well as the rough statistics of the original dataset.

## Contents

1. Introduction .....	1
2. Theory.....	1
2.1. Neural Networks .....	1
2.2. Recurrent Neural Networks.....	3
2.3. Long Short-Term Memory.....	4
3. Data .....	5
4. Classification.....	6
4.1. Network Architecture .....	6
4.2. Training .....	7
4.3. Results .....	7
5. Generation .....	9
5.1. Network Architecture .....	9
5.2. Training .....	10
5.3. Results .....	10
6. Conclusion .....	13
References .....	13
Acknowledgements.....	15
Statement of Authorship.....	15

# 1. Introduction

Neural networks with Long Short-Term Memory (LSTM) are powerful tools to analyze sequential data. Today, they are heavily employed in natural language processing, speech recognition, and many other areas. The popularity of LSTM stems from its ability to “memorize” inputs over fairly large times. This enables it to make sense of quite complex connections in the data.

Due to its sequential nature, LSTM is particularly suited to analyze time series data, a common format in science. In this project, LSTM networks were applied to movement trajectories from biological cells. Each time series contains the subsequent positions of a cell while it migrates through one of three environments. The interesting aspect about this data is that cell movement is a highly stochastic process, often described by random walks. Hence, it is not easy to predict or make sense of this data, especially at a small scale.

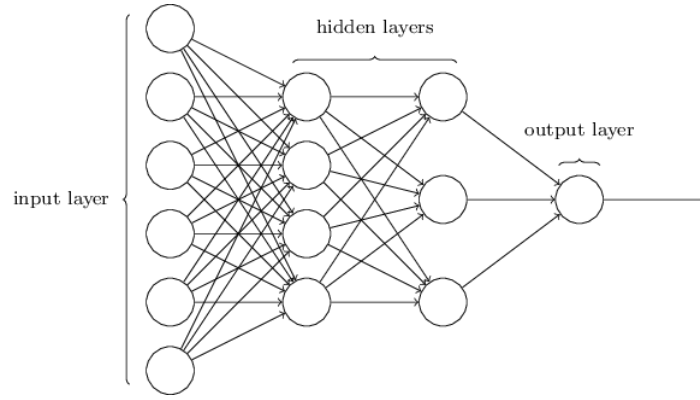
The LSTM networks were used for two tasks: 1) Classifying the movement trajectories based on the three cell environments, and 2) generating new movement trajectories based on the features that the neural network has learned. For both tasks, LSTM networks were successfully employed in the past: In terms of sequence classification (task 1), LSTM networks were used to classify phonemes (Graves 2013), recognize handwriting (Graves 2008), and classify actions in soccer videos (Baccouche 2010). In terms of data generation (task 2), LSTM networks were used to make up new text (for example in the style of Shakespeare or Wikipedia; Karpathy 2015, Graves 2013), handwriting sequences (Graves 2013), images (Gregor 2015), music (Eck 2002), and even new levels for Super Mario Brothers (Summerville 2015).

All neural networks for this project were implemented in Python 2.7.8 (van Rossum 1995), using the keras framework (version 0.3.2; Chollet 2016) with the Theano backend (version 0.8.0.dev0; Bergstra 2010, Bastien 2012). The code is available at <https://github.com/jrieke/lstm-biology>.

## 2. Theory

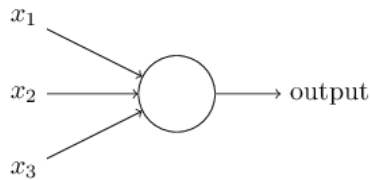
### 2.1. Neural Networks

Artificial neural networks process information, using an architecture inspired by nerve cells in the brain. In contrast to traditional algorithms, they tend to be very good at tasks which require a certain level of “intelligence”, such as speech recognition or image classification. Importantly, neural networks can perform these tasks by learning on raw data, without requiring any kind of feature-engineering. The following section gives a short overview of the basic concepts, for a thorough introduction see Nielsen (2015) and LeCun (2015).



**Figure 1:** Neural network with two hidden layers. Circles depict neurons, arrows depict connections.  
Reprinted from Nielsen (2015).

Neural networks consist of neurons (which process signals) and weighted connections (which transfer signals from one neuron to the other). The neurons are typically outlined in layers (fig. 1): The first layer (input layer) represents the raw input data (e. g. the pixel values of an image). The input values are transferred to the next layer (via the connections) and integrated and altered by its neurons. In this fashion, the signals are processed throughout the layers of the network, yielding more and more abstract representations at each step. Eventually, the last layer (output layer) generates an output (e. g. classification probabilities for the image content). All layers between the input and output layer are called hidden layers. Networks with two or more hidden layers are considered “deep” neural networks (Nielsen 2015), which recently coined the popular term “deep learning”.



**Figure 2:** A neuron with three incoming connections and one outgoing connection.  
Reprinted from Nielsen (2015).

How do neural networks process information? In short, each neuron integrates the signals from its incoming connections (usually from the previous layer) and emits a new signal to its outgoing connections (usually to the next layer). For a neuron with multiple inputs  $x_i$  (fig. 2), this output can be described by:

$$y = \sigma \left( \sum_i w_i x_i + b \right)$$

Here, each input  $x_i$  is multiplied by the weight  $w_i$  of its connection. Then, all (weighted) inputs are summed and the bias  $b$  of the neuron is added (a kind of threshold for the incoming signals). Finally, an activation function  $\sigma$  (usually nonlinear) is applied. Popular choices for the activation function are the step function, sigmoid or tanh. For computational efficiency, this equation is often rewritten with vectors and matrices to describe the computations in a complete layer.

The power of a neural network arises from tuning its parameters (namely the weights and biases) until it produces the desired output. This process is called training or learning. The remarkable point here is that a neural network learns from raw data itself - it does not require a human being to extract features from the data in order to make sense of it. There are two major ways of learning: Supervised and unsupervised. In supervised learning, the “correct” output for each data sample is known. The network can then compare its own output to this desired solution and updates its parameters accordingly. In unsupervised learning, the desired output is not known, and the neural network has to make sense of the input data itself (not discussed in detail here).

In order to update its parameters during supervised learning, the neural network first has to evaluate how well it already works. This is described by the loss or cost function, which measures the difference between the actual and the desired output. Popular examples of loss functions are mean squared error and cross entropy. The aim of training is to minimize the loss function. This is usually done via gradient descent: After evaluating the loss on some data, we calculate its gradient with respect to the weights and biases. The gradient is a vector which points into the direction of increasing loss. Therefore, we simply update the parameters of the network in the direction of the negative gradient to decrease the loss. For a single weight  $w$ , this update step can be described by:

$$w' = w - \eta \frac{\partial L}{\partial w}$$

Here,  $\partial L / \partial w$  is the partial derivative of the loss function with respect to the weight. This partial derivative represents one component of the gradient vector. The parameter  $\eta$  (learning rate) controls the magnitude of the update. Typical neural networks contain thousands or millions of weights and biases. Therefore, calculating all partial derivatives of the gradient is not straightforward. The popular backpropagation algorithm offers a computationally efficient solution for this problem.

To control the learning process, a number of different algorithms can be employed (also called optimizers). They differ in how the loss function is evaluated and how the parameters are updated. Some of the most widely used optimizers are stochastic gradient descent, RMSprop and Adam.

## 2.2. Recurrent Neural Networks

As outlined above, most connections point in the direction of the output layer. Networks with such connections are called feedforward: They process signals from layer to layer, starting at the input layer and ending at the output layer. In contrast, recurrent neural networks (RNN) contain recurrent connections: These point “backwards”, i. e. from the output to the input layer. Therefore, recurrent nets need a concept of time: For example, consider a neuron with a recurrent connection to itself. Then, the output of this neuron is fed into the same neuron at the next time step.

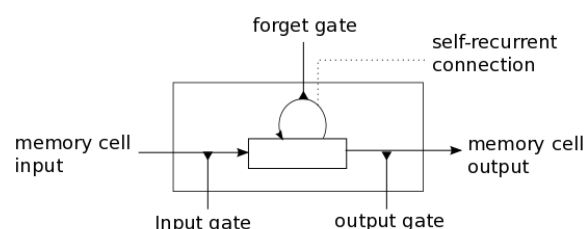
The advantage of recurrent neural networks is that they can “memorize” data: By using the time property of the recurrent connections, an input to the network can influence its output at

a future point in time. This makes recurrent nets highly suitable for sequential data, such as text, music or time series. Also, by feeding data into the network sequentially, the inputs and outputs of a recurrent net can have arbitrary length (in contrast to feedforward nets, which deal with fixed-size vectors). Thus, recurrent neural networks can map sequences to single outputs (e. g. text sentiment analysis), single inputs to sequences (e. g. image captioning), or sequences to sequences (e. g. translation).

## 2.3. Long Short-Term Memory

In principle, simple recurrent networks are able to memorize data over arbitrary time spans, given that they have the right parameters. However, in practice they struggle to learn long-term dependencies (Hochreiter 1991, Bengio 1994). Therefore, Hochreiter (1997) introduced the concept of Long Short-Term Memory (LSTM). This recurrent architecture is specifically designed to memorize data over long time spans. The concept was later expanded by Gers (2000).

Today, there are many variants of the original LSTM model. The explanation here follows the implementation in keras, which is based on LISA lab (2015) (including their modification to neglect the cell state for the output gate). The same algorithm is also explained in Olah (2015). For further details on LSTMs and possible applications see Graves (2008).



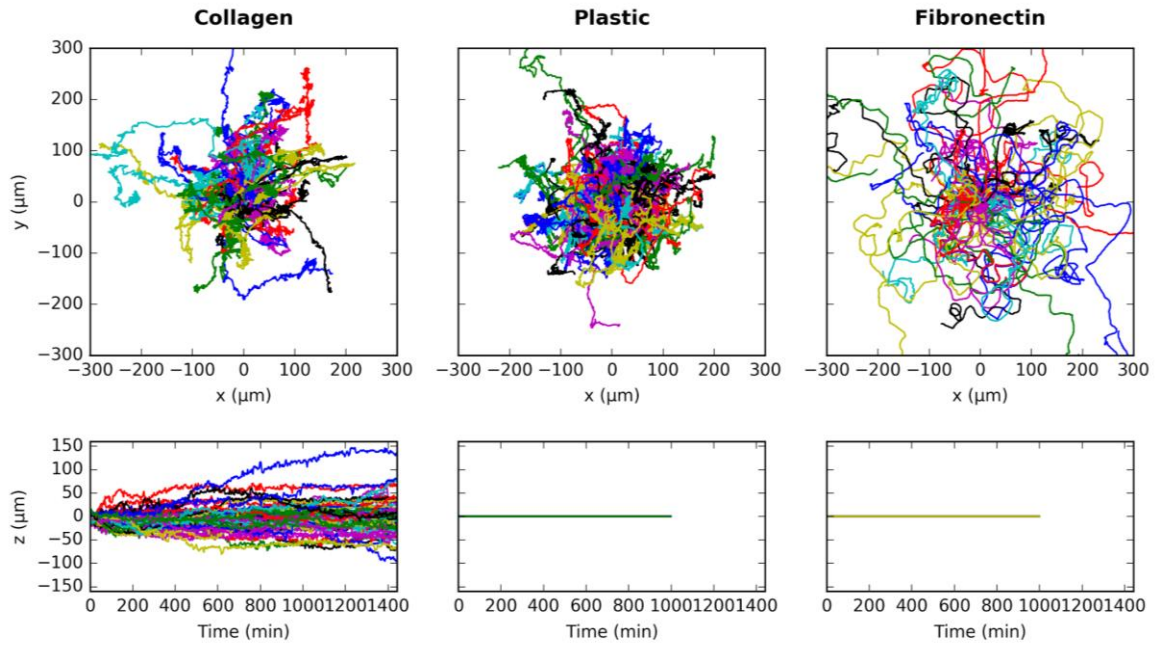
**Figure 3:** LSTM memory cell. Reprinted from LISA lab (2010).

The core principle of LSTM is the memory cell (fig. 3): It consists of a neuron with a single self-recurrent connection. This connection has weight 1, which enables the neuron to keep its state (“the memory”) for an unlimited amount of time. In contrast, the recurrent weights in a simple RNN usually have values other than 1. This makes the neurons lose their state after some time and creates exploding or vanishing gradients during training. Both effects prevent the network from developing a long-term memory.

As the weight in the LSTM cell is 1, there has to be another way to change the neuron state. This is realized via three so-called gates: The forget gate decides which fraction of the cell state to erase at each time step. The input gate controls which fraction of the current input to add to the cell state. The output gate regulates which fraction of the cell state to output. The actions of all gates are based on the input of the current time step and the output of the previous time step. Each gate has separate weights and biases, so that the memory cell can learn to control each gate operation based on the desired behavior. Especially, it can learn to block the input and forget gate, which preserves the cell state and therefore the memory.

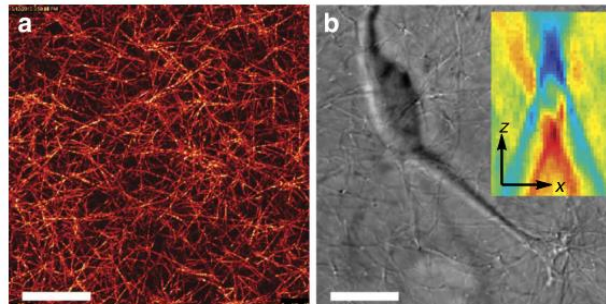
### 3. Data

The dataset consists of 311 movement trajectories from MDA-MB-231 breast carcinoma cells. Each trajectory contains the 3-dimensional positions ( $x$ ,  $y$ ,  $z$ ) of a migrating cell, sampled at discrete time points. The trajectories belong to three different classes, corresponding to the environment of the cell: Collagen (a 3D collagen network, fig. 5A), Plastic (a 2D plastic substrate), and Fibronectin (a 2D plastic substrate coated with the adhesion ligand fibronectin). Table 1 displays additional information for each class, fig. 4 shows plots of all trajectories in the dataset. Note that movement in the  $z$ -direction can only occur in the 3D collagen network.



**Figure 4:** All trajectories in the dataset.  
Top:  $x$ - $y$ -plane. Bottom: Movement in the  $z$ -direction over time.

All trajectories were generated from microscopy images of the cells during migration (fig. 5B). Cell positions were tracked by the characteristic refraction pattern around the nucleus (inset in fig. 5B; accuracy of  $2 \mu\text{m}$  r. m. s.). The data was recorded by Julian Steinwachs at the University of Erlangen-Nuremberg. It is also described in the statistical analysis by Metzner (2015).



**Figure 5:** Data recording via microscopy imaging. **A:** 3D collagen network. **B:** Cell that has migrated into the collagen network. Inset: Refraction pattern around the cell nucleus. Scale bars  $20 \mu\text{m}$ .

Reprinted from Metzner (2015).



The data was preprocessed in three ways: Firstly, the trajectories were resampled. Originally, the trajectories of the three categories had been recorded at different time intervals (table 1). This was brought to a common sampling interval of 5 min by using every second (Collagen) or fifth (Plastic, Fibronectin) time step. Secondly, the positions  $\mathbf{r}_t = (x_t, y_t, z_t)^\top$  in the trajectories were converted to velocities via

$$\mathbf{v}_t = \frac{\mathbf{r}_{t+1} - \mathbf{r}_t}{\Delta t}$$

with the (unified) sampling interval  $\Delta t$  of 5 min. This made the data easier to normalize and handle (especially for trajectories of different lengths) and improved the results for both classification and generation. The velocities contain the complete information of the original trajectories and can be easily converted back to positions. Thirdly, the velocities were normalized to the range  $[-1, 1]$  (due to the tanh activation in the LSTM cells).

**Table 1:** Summary of the dataset.

<i>Category</i>	<i>Number of trajectories</i>	<i>Time steps per trajectory</i>	<i>Sampling interval</i>
Collagen	65	577	2.5 min
Plastic	177	998	1 min
Fibronectin	69	998	1 min

## 4. Classification

As a first task, the movement trajectories were classified based on the cell environment (Collagen, Plastic, Fibronectin). For this purpose, an LSTM network was developed which takes a trajectory (or part of it) as input. The outputs of the network represent the probabilities to which class the trajectory belongs.

### 4.1. Network Architecture

The network consists of one or more stacked LSTM layers. The bottom layer has three inputs for the velocity at one time step (in x, y, z). The trajectory can then be fed into the network one time step after another. On top of the LSTM layers is a fully connected softmax layer with three outputs (yielding the probabilities for the three classes). The softmax layer scales these probabilities so that they sum up to 1.

The network was implemented using keras' built-in layers LSTM, dense and softmax.

## 4.2. Training

80 % of the trajectories were used for training, the other 20 % were held out as test data. No separate validation data was used due to the lack of an extensive hyperparameter optimization. The training trajectories were split into smaller parts containing 30 time steps each (from here on called mini trajectories). During training, each mini trajectory was fed into the network one time step after another. After the last time step, the resulting classification probabilities were observed. Due to the memory of the LSTM layers, this output depends implicitly on all previous time steps. The class probabilities were then compared to the correct class of the mini trajectory (supervised learning), using a categorical cross entropy loss function.

In each update step, the loss is averaged over several trajectories. Classes with many trajectories contribute to this average disproportionately. As the dataset is quite unbalanced (i. e. the number of trajectories  $N_i$  per class varies; table 1), the network is likely to overfit on classes with many trajectories. To prevent this effect, the contributions to the average loss were scaled by  $1/N_i$  for each class (using the parameter `class_weights` in keras' fit function).

To control the training process, RMSprop with a learning rate of 0.001 was employed. This optimizer works similar to stochastic gradient descent, but divides the gradient by a running average of its recent magnitudes.

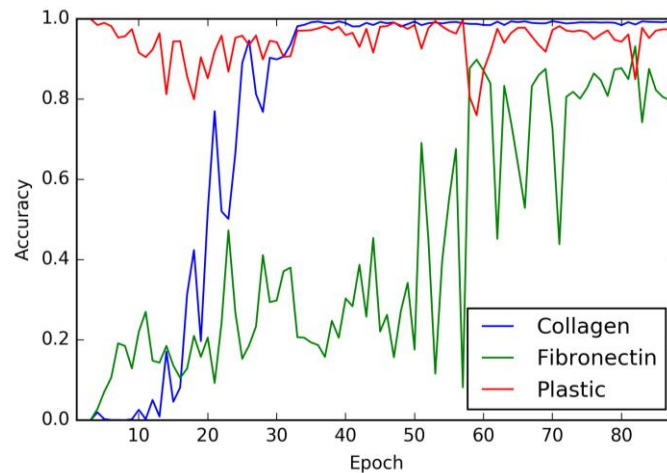
## 4.3. Results

Four experiments with networks of increasing complexity were performed: A) 1 LSTM layer with 5 neurons, B) 1 LSTM layers with 10 neurons, C) 2 LSTM layers with 10 neurons each, D) 3 LSTM layers with 30 neurons each. In all cases, no serious over- or underfitting was observed during training. Table 2 shows the final classification accuracies on the test dataset. The best overall accuracy was ~95 %. Surprisingly, this result required a relatively simple network (B) and was not significantly improved by using more complex networks (C and D). However, network B did not generalize well to longer trajectories (see below), which makes network C the preferred solution for classification. Apparently, network A was too simple to cover the complexity of the data. Comparing the different classes, all networks showed slightly lower accuracies for Fibronectin than for Collagen and Plastic. The classification results of the LSTM networks clearly exceed other methods that were used on the same dataset, namely Restricted Boltzmann Machines (Feulner 2014) and statistical comparison (Metzner 2015).

**Table 2:** Final test accuracies for different networks.

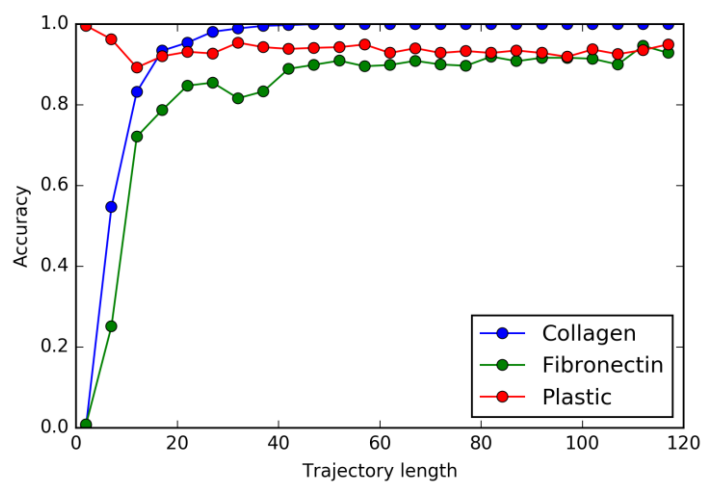
<i>Network</i>	<i>Overall</i>	<i>Collagen</i>	<i>Plastic</i>	<i>Fibronectin</i>
A (1 LSTM layer, 5 neurons)	0.85	0.95	0.96	0.42
B (1 LSTM layer, 10 neurons)	0.94	1.00	0.94	0.85
C (2 LSTM layers, 10 neurons each)	0.95	0.98	0.94	0.87
D (3 LSTM layers, 30 neurons each)	0.95	0.98	0.97	0.80

During training, the networks usually achieved a high classification accuracy for Collagen at first. Later on, they also learned to differentiate the classes Plastic and Fibronectin, albeit much slower. Fig. 6 shows an example of this trend (network B; the high classification accuracy for Plastic at the beginning is an effect of the random initialization of the network).



**Figure 6:** Development of training accuracies during training (network B).

Finally, the networks were tested on mini trajectories of different lengths (sampled from the test dataset). The accuracies for network C are displayed in fig. 7. As expected, the network showed bad results for small mini trajectories (apparently it always predicted Plastic). The classification started to become reasonable at 10 to 15 time steps per trajectory. Apparently, the network needed that many time steps to make sense of the data. Overall, networks C and D generalized well from the training trajectories (30 time steps) to longer trajectories. In contrast, network B (which had a similar overall accuracy), showed bad results for longer trajectories. Also, training the networks with shorter mini trajectories (than 30 time steps) resulted in bad accuracies for longer trajectories.



**Figure 7:** Test accuracies for mini trajectories of different lengths (network C).

## 5. Generation

For the second task, a neural network was trained to generate completely new trajectories, based on the features that it had learnt from the original trajectories. The idea behind generating new trajectories is relatively simple: Feed the velocity for one time step into the network, predict the velocity for the next time step, then feed this value back into the network, predict the velocity for the next time step, and so on.

The most intuitive approach for this is to let the network predict the next velocity directly, i. e. to have three real-valued outputs (for x, y, z). However, such a network quickly settles into an equilibrium and yields the same values over and over again (a kind of average of the learned features). In order to bring the network away from the equilibrium, one can add noise to the output before feeding it back into the net. In preliminary experiments, this made the generated trajectories look more or less realistic. However, the noise covers up some of the fine-grained features of the data (as investigated in preliminary work with one-dimensional data). The final solution in this project follows the ideas in Graves (2013) and Bishop (1994): The network does not predict the next velocity directly, but the parameters of a probability distribution, from which the next velocity can be sampled.

### 5.1. Network Architecture

The network architecture is inspired by the mixture density network (MDN) introduced by Bishop (1994). The network consists of one LSTM layer with three inputs for the velocity (in x, y, z) and 10 neurons. On top is a fully connected layer, which implements a Gaussian mixture model (GMM). Its outputs are the parameters of a mixture distribution, i. e. a linear combination of  $M$  Gaussian distributions.

For each Gaussian (or mixture component), the network outputs the mean  $\mu_i$ , the standard deviation  $\sigma_i$  and the mixture weight  $\alpha_i$ . With these parameters, the probability to get a velocity  $\mathbf{v}_{t+1}$  at the next time step is given by the mixture distribution

$$P(\mathbf{v}_{t+1}|\mathbf{v}_t) = \sum_{i=1}^M \alpha_i(\mathbf{v}_t) \phi_i(\mathbf{v}_{t+1}|\mathbf{v}_t)$$

with the standard Gaussian distribution

$$\phi_i(\mathbf{v}_{t+1}|\mathbf{v}_t) = \frac{1}{\sqrt{2\pi}\sigma_i(\mathbf{v}_t)} \exp\left[-\frac{(\mathbf{v}_{t+1} - \mu_i(\mathbf{v}_t))^2}{2\sigma_i(\mathbf{v}_t)^2}\right].$$

Due to the memory of the LSTM layer, these probabilities depend implicitly on all previous time steps that were fed into the network. Note that the means  $\mu_i$  are three-dimensional vectors (i. e. one value for each dimension x, y, z), but the standard deviations  $\sigma_i$  are scalar. Strictly speaking, this assumes that the velocities in x, y and z are statistically independent (which they are obviously not). However, a Gaussian mixture model with enough components is in principle able to model any probability distribution (Bishop 1994).

Therefore, this is not a serious restriction if the number of mixture components  $M$  is high enough. In all experiments below,  $M$  was set to 10.

The task of the GMM layer at the top of the network is to scale the parameters of the mixture distribution to a reasonable range. Specifically, it limits the standard distributions to positive values (via an exponential function), and tunes the sum of the mixture weights to be 1 (via a softmax function). With the final parameter values, the velocity  $\mathbf{v}_{t+1}$  for the next time step can be evaluated by sampling from the mixture distribution described above.

As for the classification task, the LSTM layer was implemented using keras' built-in LSTM class. However, the LSTM layer was rendered stateful here. This was necessary because each training or prediction step processed only one time point of a trajectory (see 5.2). Statefulness ensured that the LSTM cells kept their internal states (and thus their memory) in between these steps. The GMM layer was implemented as a custom layer on top of keras, largely following the implementation in <https://github.com/fchollet/keras/issues/1061>.

## 5.2. Training

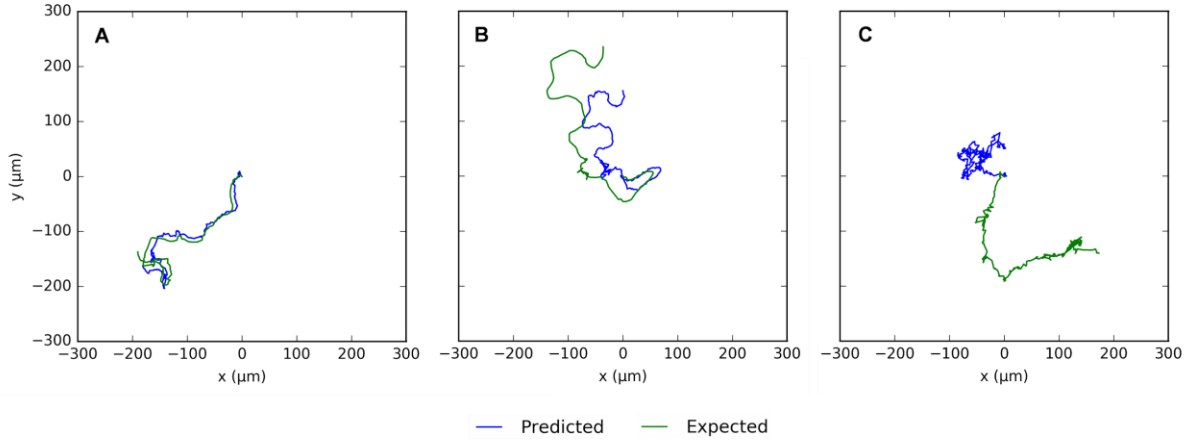
For each category, a network was trained on the complete trajectories. Every training batch contained a single time step. The resulting mixture distribution was then compared to the velocity  $\tilde{\mathbf{v}}_{t+1}$  at the next time step of the trajectory (supervised learning). The loss was modelled by a cross entropy function applied to the mixture distribution (Bishop 1994):

$$L(\mathbf{v}_t) = -\log \left[ \sum_{i=1}^M \alpha_i(\mathbf{v}_t) \phi_i(\tilde{\mathbf{v}}_{t+1} | \mathbf{v}_t) \right]$$

Of course, the batches were not shuffled during training to preserve the order of the trajectories. No test data was held out because the final aim was to generate new trajectories, and not to test the accuracy of the network on existing trajectories. Like for the classification, RMSprop with a learning rate of 0.001 was used as an optimizer.

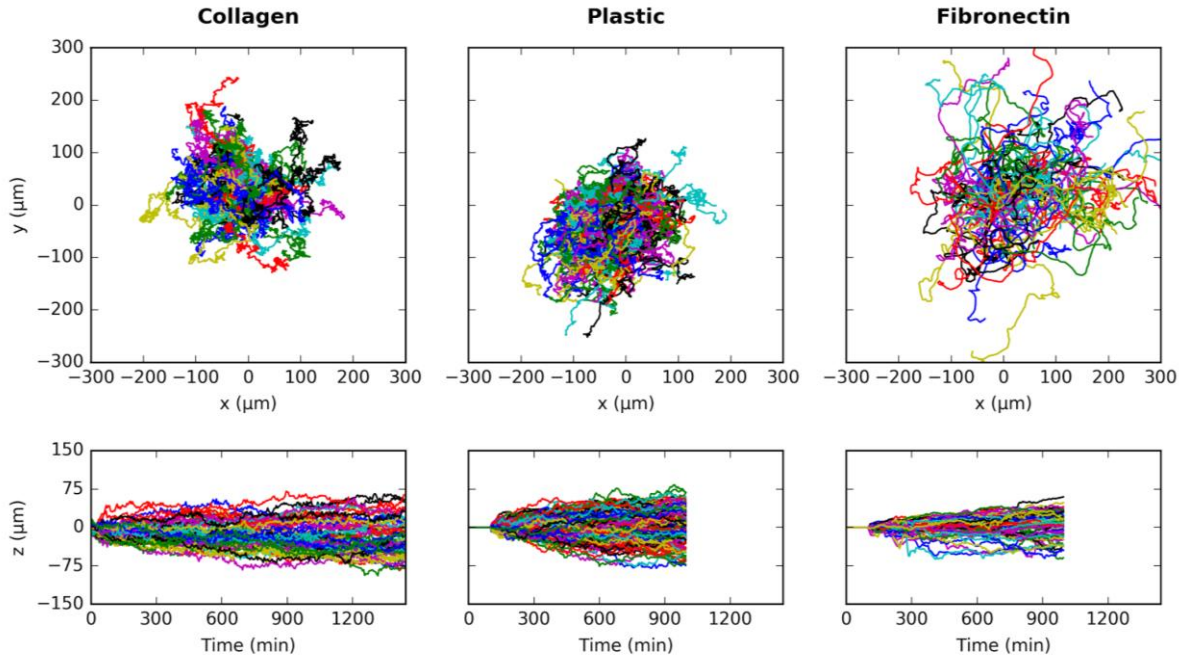
## 5.3. Results

Before generating new data, the network was tested on the training trajectories: For each time step, the velocity of the trajectory was fed into the network, yielding a prediction for the velocity at the next time step. After processing all time steps, the predicted velocities were converted back to positions. Fig. 8 shows some of the original and predicted trajectories in comparison. In some cases, the predictions of the network resembled the original trajectories closely (fig. 8A-B), while in other cases they differed (fig. 8C). Some of the small-scale structures of the original trajectories were not preserved. This is partly attributable to resampling the data before training (and therefore using only a fraction of the time steps, see 3). Due to training on velocities, the predicted trajectories were sometimes scaled or slightly shifted versions of the original ones (fig. 8B). Training on the raw positions mitigated this effect but yielded worse results overall.



**Figure 8:** Exemplary trajectories predicted by the network, using training trajectories as input. A and B from class Fibronectin, C from class Collagen.

After this test, the network was used to generate completely new trajectories: To initialize the memory of the LSTM cells, a seed was fed into the network, consisting of the first 20 time steps of a training trajectory (the network showed similar results for seeds with random or zero values). Then, the velocities of the new trajectory were predicted by repeatedly using the output of the network as an input for the next time step. Due to the probabilistic nature of the MDN, the network produced different trajectories in each run. Fig. 9 shows some generated trajectories for each class.

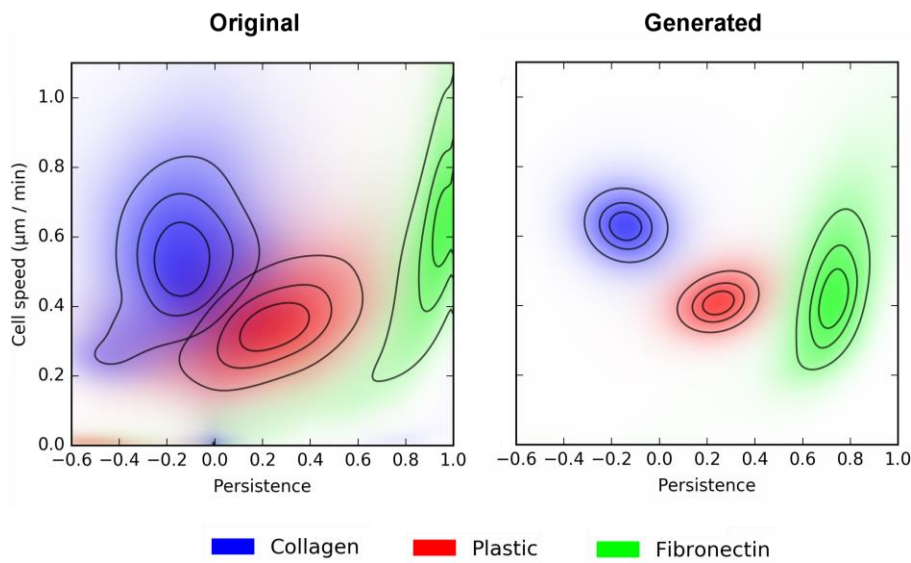


**Figure 9:** Trajectories generated by the LSTM network. Top: x-y-plane. Bottom: Movement in the z-direction over time.

Based on optical comparison, the generated trajectories seemed to resemble the features of the original trajectories (fig. 4) for all three classes (extent in the x-y-plane, angularity and smoothness, overall shape). However, the neural networks did not replicate the missing movement in the z-direction for Plastic and Fibronectin. Also, the networks did not create as

many “outliers” (for example very long trajectories) as in the original data. Finally, in some training runs the generated trajectories showed a preferred direction. This effect was decreased by further training, but could not be completely prevented, not even by normalizing the data differently or using more LSTM layers.

Besides optical comparison, the trajectories were analyzed with the superstatistical framework described in Metzner (2015), using the bayesloop Python package (version 0.5; Mark 2016). In short, the movement of each cell was modelled by a random walk with the two parameters persistence and cell speed (for details see equation 1 in Metzner 2015; note that the cell speed  $c_t$  is a scaled version of the activity  $a_t$  with  $c_t = a_t / \sqrt{1 - q_t^2}$ ). The values of persistence and cell speed are allowed to change over time (the random walk is therefore heterogenous). To analyze the original and generated trajectories, both parameters were evaluated at each time point of each trajectory. Then, the observed parameter combinations were averaged over all trajectories and time points, yielding the parameter densities shown in fig. 10.



**Figure 10:** Parameter densities of the original and generated trajectories.

Black lines represent the 10, 25 and 50 % credible regions. The left plot shows the same data as in figure 5b of Metzner (2015), but differs slightly from there because a scaled version of the parameter model was used.

Comparing the original and generated trajectories, the peaks of the parameter densities are located in the same areas of the parameter space for all three classes. However, the parameter densities of the generated trajectories are much narrower. Apparently, the network was able to successfully replicate the rough statistics of the original trajectories, but not their variation. This matches the observation that the neural network produced less outliers than in the original data. Perhaps a more complex network could learn more of this variation.

## 6. Conclusion

In this project, neural networks with Long Short-Term Memory (LSTM) were applied to real-valued time series, which describe the movement of biological cells. The results demonstrate that the networks were able to learn the features of this data, even though cell movement is a highly stochastic and complex process. Firstly, the movement trajectories were classified based on the cell environment. The high classification accuracy (~95 %) proves that an LSTM network is able to discriminate even slight differences in the data (compare the similar classes Collagen and Plastic). Secondly, a neural network was trained to generate new movement trajectories one time step after another. The resulting trajectories resembled the original data both in terms of shape and statistics. Surprisingly, both tasks required relatively simple networks to yield good results, comprising only one or two LSTM layers with few neurons.

In principle, the procedures and networks developed in this project are not specific to cell movement. They could as well be applied to any other kind of sequential, three-dimensional data. Furthermore, it would be relatively simple to generalize the network implementations to work with time series of any dimension. This opens up interesting opportunities, for example to analyze other scientific time series, predict stock prices, or generate music.

Besides using other data, there are two clear directions for future work: Firstly, tuning the hyperparameters of the networks and trying out slightly different architectures. This will certainly improve the results of both classification and generation. Some results for networks of different complexity were already presented in 4.3, but an extensive hyperparameter optimization was not performed in this proof-of-concept study. Secondly, the neural networks developed in this project could be trained and tested on data from more cells, probably using other cell types and environments.

## References

- Baccouche, M., Mamalet, F., Wolf, C., Garcia, C., & Baskurt, A. (2010). Action classification in soccer videos with long short-term memory recurrent neural networks. In *Proceedings of the 20th International Conference on Artificial Neural Networks: Part II* (pp. 154–159). [http://doi.org/10.1007/978-3-642-15822-3\\_20](http://doi.org/10.1007/978-3-642-15822-3_20)
- Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., ... Bengio, Y. (2012). Theano: new features and speed improvements. *Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop*.
- Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*, 5(2), 157–166. <http://doi.org/10.1109/72.279181>
- Bergstra, J., Breuleux, O., Bastien, F. F., Lamblin, P., Pascanu, R., Desjardins, G., ... Bengio, Y. (2010). Theano: a CPU and GPU math compiler in Python. In *Proceedings of the Python for Scientific Computing Conference (SciPy) 2010*.
- Bishop, C. M. (1994). Mixture density networks. *Neural Computing Research Group Report*.



- Chollet, F. (2016). keras. Retrieved from <https://github.com/fchollet/keras>
- Eck, D., & Schmidhuber, J. (2002). A First Look at Music Composition using LSTM Recurrent Neural Networks. Technical Report No. IDSIA-07-02.
- Feulner, B. (2014). Analysis of cell trajectories with Restricted Boltzmann Machines. University of Erlangen-Nuremberg.
- Gers, F. A., Schmidhuber, J., & Cummins, F. (2000). Learning to forget: continual prediction with LSTM. *Neural Computation*, 12(10), 2451–2471. <http://doi.org/10.1162/089976600300015015>
- Graves, A. (2008). Supervised Sequence Labelling with Recurrent Neural Networks. *Studies in Computational Intelligence*, Vol. 385. Springer. <http://doi.org/10.1007/978-3-642-24797-2>
- Graves, A. (2013). Generating Sequences With Recurrent Neural Networks. arXiv Preprint.
- Gregor, K., Danihelka, I., Graves, A., Jimenez Rezende, D., & Wierstra, D. (2015). DRAW: A Recurrent Neural Network For Image Generation. In *Proceedings of The 32nd International Conference on Machine Learning* (pp. 1462–1471).
- Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Technical University Munich.
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780. <http://doi.org/10.1162/neco.1997.9.8.1735>
- Karpathy, A. (2015). The Unreasonable Effectiveness of Recurrent Neural Networks. Retrieved from <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444. <http://doi.org/10.1038/nature14539>
- LISA lab. (2010). LSTM Networks for Sentiment Analysis. Retrieved from <http://deeplearning.net/tutorial/lstm.html>
- Mark, C. (2016). bayesloop. Retrieved from <https://github.com/christophmark/bayesloop>
- Metzner, C., Mark, C., Steinwachs, J., Lautscham, L., Stadler, F., & Fabry, B. (2015). Superstatistical analysis and modelling of heterogeneous random walks. *Nature Communications*, 6(7516). <http://doi.org/10.1038/ncomms8516>
- Nielsen, M. (2015). *Neural Networks and Deep Learning*. Determination Press.
- Olah, C. (2015). Understanding LSTM Networks. Retrieved from <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- Summerville, A., & Mateas, M. (2015). Super Mario as a String: Platformer Level Generation Via LSTMs. arXiv Preprint.
- van Rossum, G. (1995). Python tutorial, Technical Report CS-R9526. Amsterdam.

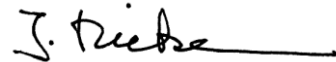
## **Acknowledgements**

Thanks to Richard Gerum, Christoph Mark and Claus Metzner for supervising this project and providing many helpful ideas and comments.

## **Statement of Authorship**

I confirm that this report was composed by me and is based on my own work, unless stated otherwise. No other person's work was used without acknowledgement. All references as well as verbatim extracts were quoted and all sources of information were clearly identified.

Erlangen, 31.03.2016

A handwritten signature in black ink, appearing to read 'J. Niebe', followed by a horizontal line.