Freie Universität Berlin
Department of Mathematics and Computer Science
Institute of Computer Science
Biorobotics Lab

Predicting bee trajectories using Recurrent Neural Networks

Master Thesis

Mehmed Halilovic
Student ID: 4663111

First Examiner: Prof. Dr. Tim Landgraf

July 30, 2019

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben. Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Datum Unterschrift

Abstract

In this work a Prediction model, using LSTMs and mixture density networks was trained to predict trajectories. The model is tested on simple models and then applied to bee trajectories. A simulation was implemented that can run different movement models. The simulation offers a visualization of the movement models. After the Prediction models were trained an analysis that made use of the hidden states of the model was done. Plotting T-SNE and UMAP projections revealed interesting clusters in the hidden states. Furthermore a classification task was solved to see if the hidden states of the Prediction model are able to boost classification performance. The results revealed that if the Prediction model is able to predict realistic trajectories, classification performance can be improved for problems where few labels are available. All the easy movement models were to some extend successfully learned by the Prediction model. The Prediction model was not able to predict realistic bee trajectories.

Contents

Chapter 1

Introduction and Motivation

The honey bee is one of the most famous insects in the world. You can find this insect almost anywhere in the world today. Due to the honey that bees produce they are also used as working animals. Not only the honey is important for farmers but also the fact that bees can be used for pollination of crops. That is why honey bees have a great impact on the environment and have been subject to research many times. Research groups like the BioroboticsLab want to understand the way bees work. This requires a lot of observing of bees, which can be a tedious and time intensive task for a human. The automatic monitoring and tracking of individual bees is already possible (Boenisch et al., 2018)[4](Wild, Sixt & Landgraf, 2018)[3]. By using cameras and computer vision systems thousands of bees can be observed simultaneously. This yields large data sets that need to be analyzed. In order to do so algorithms need to be developed that can recognize different behaviors of bees. Some behaviors can already be automatically detected with a high accuracy, e.g. the automatic waggle dance (Wario et al., 2017)[5].

Advances in Machine Learning provide powerful methods that can be used to develop such systems. In particular the advance of neural networks since 2012 stands out. A lot of problems today are solved by using neural networks. Face recognition (Lawrence et al., 1997)[19], self driving cars (Bojarski et al., 2016)[21] and language translation (Wu et al., 2016)[20] are good examples of fields where neural networks are applied. The ability to use huge data sets to gain insights can also be used in bee research. Bee monitoring and tracking already use these kind of algorithms.

A big problem for behavior classification is that a lot of labeled data is needed to train classifiers. Generating labeled data by hand takes long and can be a boring and expensive task. Reducing the amount of manually labeled data to a minimum is a desired goal.

In this master thesis recurrent neural networks will be used to predict trajectories. Not only bee trajectories, but also trajectories of simulated agents will be used for this. The data used to predict the next trajectory will be past trajectories. Since the tracking of bees is already

automatized no manually labeled data is needed for this task. The model will be tested on the simulated agents and later applied to bee trajectories. After each training of the network, the hidden states will be analyzed to gain information about properties or states of the agents or bees. If the hidden states contain enough high level information to learn a classification task with few labels, this could improve the way research is done on bees.

Chapter 2

Related Work and Background Knowledge

2.1 Beesbook Project

This work is a contribution to the Beesbook Project. The Beesbook project aims to make research on bees easier. In the work of (Boenisch, Rosemann, Wild, Wario, Dormagen & Landgraf, 2018)[4] the team of the BioroboticsLab describes how a whole honey bee colony can be tracked over multiple months. In order to do the research on bees a artificial bee hive was built, where a bee colony can live. This is the main setup of the Beesbook project and works with 4 cameras that are aimed at a bee hive. The setup can be seen in figure 2.1. All bees in the hive are tagged on their back. In order to have all bees tagged they are hatched outside of the bee hive. Then each bee that hatches gets tagged and put into the bee hive. In figure 2.2 tagged bees can be seen. There is a tube leading from the bee hive to the outside. This setup enables tracking of individual bees inside the bee hive. Localization of bees and decoding of tags was automatized by (Wild, Sixt & Landgraf, 2018)[3] using Convolutional Neural Networks (CNN). During the bee season huge amounts of tracking data are gathered. This data will be used in this thesis.

2.2 Predicting Fly trajectories - Eyjolfsdottir

In the paper of (Eyjolfsdottir, Branson, Yue & Perona, 2016)[1] predictions of different types of trajectories were made. In one of the experiments they learned to predict fly trajectories using a generative Recurrent Neural Network (RNN). The network consists of a generative and a discriminative part. This allows the network to predict future motion and to classify current actions. The experimental setup for fruit flies was many flies inside a plane glass cylinder. The

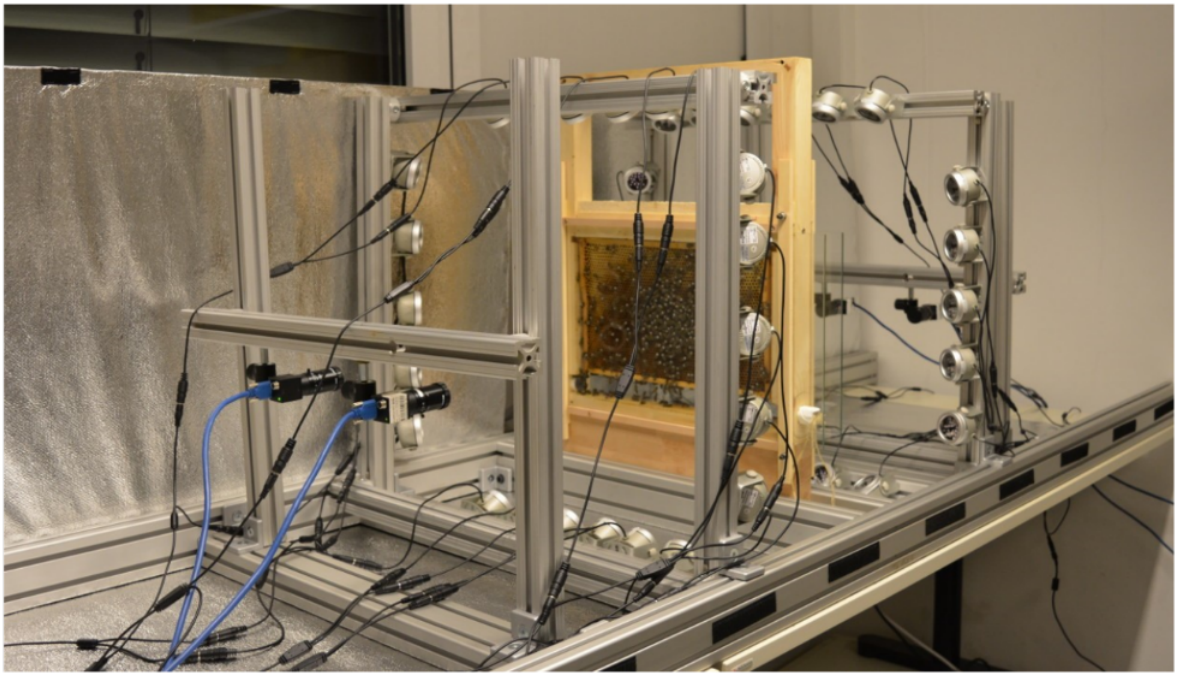

Figure 2.1: Beesbook setup for bee tracking.

The cameras are connected with the blue cables. The bee hive is inside the wooden part in the middle. There is a window on both sides of the hive and the bees can be seen as a brownish chunk in the image. (Image from BiorboticsLab blog[22])

Figure 2.2: Bees with ID tags.

Bees with round ID tags on their back. The tags are round and encode orientation and bee ID. (Image from BiorboticsLab blog[22])

flies were tracked and the trajectories and a representation of the surroundings was saved. The surroundings consisted of other flies that move on the glass plane and the walls of the glass cylinder. The input to the network was a vector where the trajectories and the surroundings are represented as a 2D-Ray-casting array.

The trained network produced trajectories that looked similar to real trajectories. When analyzing the hidden states of the network, they were able to cluster certain properties of flies, e.g. what gender the flies have. The conclusion was that using unlabeled data for predicting trajectories helps to improve the action classification. Also by using T-SNE high level information like fly gender can be separated.

2.3 Predicting Bee trajectories - Burgert

In this work (Burgert, 2018)[6] that also was part of the Beesbook Project, prediction of future bee movement was implemented based on raw images of bees. The goal was to predict the next image of the bee based on the last images of that bee. The model consisted of a 3D Convolutional Neural Network and was able to predict images that looked close the the original ones. While the predictions were not perfect the hidden states had some interesting information about the bees. The hidden states were projected on a 2D plane using Principal component analysis (PCA) and T-SNE. In the projection some behaviors were clustered. While the problem of predicting high resolution images of the future wasn't solved, the work showed that behaviors might appear in clusters in the hidden states and the network learned meaningful information.

2.4 Summary

Labeled data is not always easy to assemble. That is why there is an interest to use unlabeled data or data where labels are easy to acquire to boost the performance of a classification task where a small amount of labeled data is available. This is what both (Eyjolfsdottir et al.)[1] and (Burgert)[6] tried to use to predict movements and behaviors of insects. This idea was also applied to other fields.(Eyjolfsdottir et al.)[1] also applied this technique to handwriting. (Dai & Quoc, 2015)[2] applied it to language processing. In each of these works a network that is generating some sort of sequence was used. Usually an RNN was used for generating the sequences. In order for this to work the sequences generated have to have something in common with the labels in the Classification task. A common problem for predicting a trajectory from past trajectories and classifying the behavior in multiple trajectories is to understand the trajectories itself. This could be the intuitive reason for why this approach was tried so often. In all of the mentioned works some positive result was reached.

Now lets consider the data from the Beesbook project: A lot of data is acquired by automatic tracking of bees. This way unlabeled trajectories that show the path of the bees are gathered. An interesting information about the trajectories would be to know the current behavior of the bee. Since adding these labels manually is an time expensive task we want to minimize the labels needed while still getting a good classification performance. That is why unlabeled trajectories should be used to gain information about the movement of bees. A network that trained to predict future trajectories might have learned to understand different environments a bee can have and what possible movements are. That is why the goal of this work will be to do exactly that, train a recurrent neural network to predict future trajectories and use that network to try to gain more insights about the current state of a bee.

Chapter 3

Implementation

3.1 Objective

The task at hand is as follows: We want to train a model to predict bee trajectories based on past bee trajectories. In order to do that a recurrent neural network will be trained that we call Prediction model. After training the Prediction model will be analyzed and used for classification of labels.

To test the system before applying it on the real bee data a simulation will be implemented. In that simulation easy movement models will be used to generate trajectories for testing. This data is easily understood and can show if the model works with simple agents.

To analyze the Prediction model the simulation will provide the possibility of running the Prediction model. This makes it possible to look at the predicted trajectories. These can then be compared to the trajectories from the real model.

Additionally the hidden states of the Prediction model will be analyzed. For that test sets are used. The Prediction model will run with the data from the test sets and the hidden states generated during that run will be extracted. All the hidden states will then be projected in two dimensions, once using T-SNE and once using UMAP. Each point in that projection will be colored in according to labels from the original test data. The hope is that some labels appear as clusters in the projection.

Finally multiple different models will be trained that can classify certain labels we provide. They mostly differ in the data they use to classify the labels. One of the models will use the hidden states of the Prediction model and we compare the performance among the different models.

	timestamp timestamp w	frame_id numeric (32)	detect smalli	track_id numeric (32)	x_pos smallint	y_pos smallint	orientation real	x_pos_hive real	y_pos_hive real	orientation_hive real	bee_id smallint	bee_id_confidence real	cam_id smallint
1	2016-08-1...	159545571123277221	44	12224357708595641957	3449	2237	-2.15553	138.753	13.1346	2.56242	3044	0.0027569	0
2	2016-08-1...	159545571123277221	75	17694980449671059929	100	2811	2.29414	174.743	227.628	0.717907	1712	0.306738	0
3	2016-08-1...	159545571123277221	65	11875700634960920946	90	2687	1.31696	166.702	228.186	-0.239885	2243	0.953735	0
4	2016-08-1...	159545571123277221	1	17063990963001130659	3187	838	-2.20139	47.93	30.1192	2.51273	2712	0.147329	0
5	2016-08-1...	159545571123277221	79	9361506680109942638	2337	2875	-2.43483	179.545	84.8986	2.29501	2530	0.863972	0
6	2016-08-1...	159545571123277221	69	14754738454714760411	2788	2739	-0.0512378	170.994	55.8668	-1.60911	2908	1	0
7	2016-08-1...	159545571123277221	68	10030180448830221703	3302	2712	1.71953	169.726	22.613	0.145319	1915	0.892657	0
8	2016-08-1...	159545571123277221	42	17550022105603921256	3236	2228	0.0116636	137.972	26.95	-1.54554	1831	0.709226	0
9	2016-08-1...	159545571123277221	37	12314704218373722099	1348	2048	-2.91737	125.361	147.842	1.80033	1457	0.984314	0
10	2016-08-1...	159545571123277221	83	17252495102934704678	2938	2915	0.058604	182.692	46.1757	-1.49577	1650	0.854041	0

Figure 3.1: Bee data points from the Beesbook Database.

3.2 Programming language and deep learning platform.

This project is a contribution to the Beesbook project. Most of the projects that are done at the time of this theses are done using Python and PyTorch[17]. In order to keep the code consistent with other parts of the project Python will be used for all the coding. PyTorch will be used as the main deep learning platform for all the machine learning models in this thesis. Matplotlib [16] is used to plot all the graphs in this thesis. A simple graphics library that was written for the book by John Zelle (2010)[18] was used to implement the visualization of the simulation.

The code for all the methods described in this chapter is available at `https://github.com/AffeMitStein/trajectory-prediction`.

3.3 Data - Beesbook Project

In this section the shape of the data will be explained. The trajectory data was generated by the Beesbook Project. The setup that generates this data was mentioned in chapter 2. For each bee the position, orientation, time, bee id, and track id are saved. The data is saved in a database. In Figure 3.1 ten data points from that database are shown.

3.3.1 Data model

In this section the question of what the input to the Prediction model will look like is answered. Each bee will be looked at individually, which means the input of the network will process time series data of one bee at a time. Trajectory data is needed to predict movements. Trajectories will consist of position, orientation and a representation of the vision. Positions and orientations will be given as relative to the previous step. In the real world bees need to react to other bees, e.g. bees can not run through one another. That is why the vision of the bee is part of

the input. A raycasting array similar to the one in (Eyjolfsdottir et al.)[1] will be used as the representation of the input. The vision combined with the position and orientation of the bee will be the trajectory and therefore the input to the Prediction model.

The same data model will be used for the trajectories that were generated by the simulation.

3.3.2 Preprocessing for bee data

To obtain trajectories as described in the data model some preprocessing is needed. The database only contains data points of individual bees at individual times.

To calculate the vision of a bee the neighbourhood needs to be scanned for other bees and walls. That is why for each timestamp a KD-Tree is built. The bee position is used to build the KD-Tree and a quick nearest neighbour search for every bee is done. This way all bees within a certain range can be obtained. Now the neighbourhood is transformed into 2D-Arrays. The 360 degree vision around the bee is divided into 16 bins. Each bin represents a section of the vision. If a neighbour is inside a section and not further away than the maximum visual range, a value is assigned to that bin. The value is proportional to the distance of the neighbour. The maximum visual range is predefined and represents how far each bee can see. Additionally for every section there is a binary bin, two orientation bins, an activity bin and two bins that represent the relative angles between the two bees. The binary bin is an indicator whether a bee is in a section. Orientations are represented with two values, which is why there is two arrays for each orientation value. The activity is proportional to the average speed the neighbouring bee is moving at. The walls are represented as a real 2D-Raycasting array, which means a ray for every bin is compared to the borders of the bee hive. If a ray hits a wall within the maximum visual range we add a value to that direction, that is proportional to the distance to the wall.

By sorting the data points by time and track-ids we can find successive data points of single bees. This way we can save the path of the bee from track-beginning to track-end. While doing that relative coordinates are calculated by taking the difference between positions and orientations of successive timesteps.

Finally the tracks are separated in windows of a chosen size. The size of the windows determines how many previous steps from the bee are used in training to predict the next step. Some windows need to be filtered and removed from the training set. This is done to filter tracking errors in the data or tracks where a lot of successive timestamps are missing. The most common examples are windows where bees have traveled an unrealistic distance in short time or windows where successive data steps are many seconds apart.

3.4 Simulation

Bee trajectories are very complex and to evaluate if anything meaningful is learned is very hard itself. To verify that the prediction system works simpler movement models will be used for testing. Therefore an agent based simulation is implemented. To generate trajectories the simulation uses the same data model as the data model for the bees.

The simulation is able to simulate agents and make them move according to one of the movement models. Each agent has a position, an orientation, a history of all his movements and knows his neighbourhood up to a maximal range. The neighbourhood consists of other agents and borders. The visualization is kept simple, only displaying position, orientation and a trail of the agent.

The simulation can be used to generate training data based on different movement models. The data is in the same shape as the Beesbook data.

The trained network can also be plugged into the simulation in order to watch how the agents behave when moving according to the trained Prediction model.

The Pseudo code for the basic simulation loop can be seen in the following box:

```
def do_simulation(num_agents, num_timesteps, draw = True):
    timestep = 0
    win = 0
    if draw == True:
        initialize_graphics()
    agents = initialize_agents(num_agents)
    for i in range(num_timesteps):
        update_vision(agents)
        update_agents(agents)
        for ai in agents:)
            move_agent(ai, win, draw)
        if draw == True:
            update_graphics()
        timestep = timestep + 1
    close_graphics()
    return agents
```

As can be seen the basic structure of the code is very simple. In a addition to the 3 initialization parameters *num_agents*, *num_timesteps* and *draw* multiple other simulation parameters can be set. Those parameters include the height and width of the borders of the world, the model used for the simulation run, how many visual bins the agents will use and the maximum visual range

for every agent. The simulation starts with *initialize_graphics()* and *initialize_agents()*. These initialization functions are run to prepare the world with the agents. The main loop starts by updating the vision for every agent in *update_vision()*. This is where the Ray casting is done and the values for all the vision bins are calculated. In *update_agents()* the movement of the agents is calculated based on the state of the agents and the algorithm used. After that every agent is moved according to those calculations. Finally the visualization is updated with the new positions of the agents and the loop repeats. Once the loop repeated *num_timesteps* times the simulation stops by closing the graphics and returning the history of all the agents.

3.5 Simulation models

The simulation can be run with the four models described in this section. During initialization the agents run a few steps with a random walk. This is done because some models work by analyzing recent trajectories.

3.5.1 Simple model

This model will be used to see if the network is able to learn basic movements like walking in straight lines and avoiding wall collisions. The rules of the model are as follows:

- Use the same movement vector as in last step. Keep velocity at 1 pixel per timestep.
- If a wall is seen gain an acceleration away from the wall. Acceleration speed is dependent to the distance to the wall.
- The agent has a 360-vision with a radius of 50 units.

Trails produced by this model can be seen in figure 3.2. As can bee seen agents move in straight lines. Once they come near a wall they avoid it by moving away from it.

3.5.2 SimpleCouzin model

In this model agents have to adjust their movement according to the movement of their neighbours. Depending on how far a neighbour is away different policies are used. The rules for this model were inspired by the work of (Couzin, Krause, James, Ruxton & Franks, 2002)[13]. In that paper a movement model is presented where the space around the moving agent was divided into 3 zones. Agents use different policies to react on other agents, depending on the zone the other agents are in. This will be adopted for our model to test if our network can

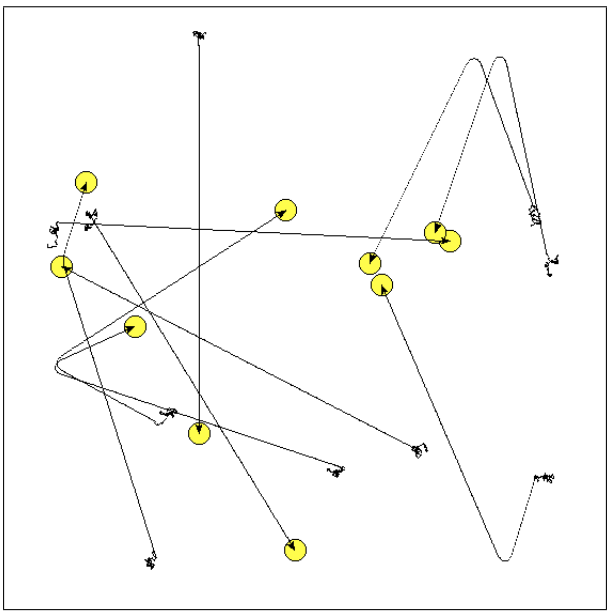

Figure 3.2: Snapshot of the Simulation running with the Simple Model.
The agents are represented as yellow circles. The orientation is represented by an arrow inside the circle. Every agent leaves a black line as a trail, so recent movement can be seen. In the image we can see the path of random walk in the trail of every agent. Once the agents start moving according to the Simple Model the trail gets straightened.

later learn to react to different positions and movements of other agents. In figure 3.3 we can see the partition of the area around an agent. Our model is called SimpleCouzin because it is not the exact movement model described in (Couzin et al.)[13]. The rules for the model are described here:

- Agents walk in a straight line with a base velocity of 1.
- Assign all neighbours to one of three zones based on the distance.
- For every neighbour in the zones the direction of the movement vector will change slightly, but the own velocity has the most influence on the next movement vector. Every neighbour has the same influence and changes the direction according to following rules:
  - If the neighbour is in ZOR (Zone of repulsion): Move away from that neighbour
  - If the neighbour is in ZOA (Zone of attraction): Move towards that neighbour
  - If the neighbour is in ZOO (Zone of orientation): Move in same direction as the neighbour.
- If a wall is closer than 45 units gain an acceleration away from the wall. Acceleration speed is dependent to the distance to the wall. This can temporarily change the velocity of the agent, but when the wall gets out of sight the velocity will return to the base velocity of 1.

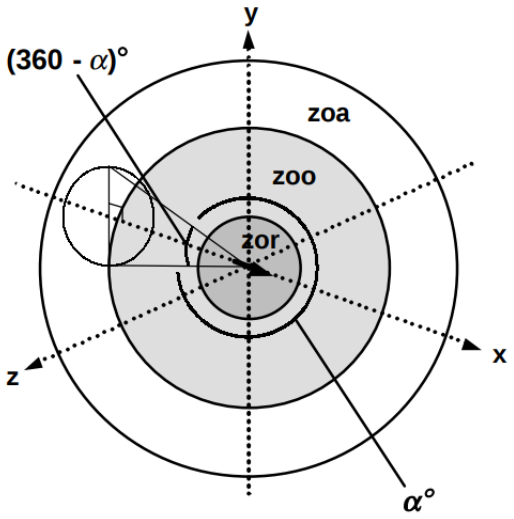

Figure 3.3: The partition of zones as shown in the (Couzin et al.)[13] paper. In the SimpleCouzin model these zone partitions are used. The agent is in the middle of the circle and zones are layered around him. In our model a 360 degree vision is assumed which means that α is set to 0, so that there is no blind spot.

- The agent has a 360-vision with a radius of 50 units.

Trails produced by this model can be seen in figure 3.4. Agents start moving in the same direction once they meet. If a single agent meets a group of agents he mostly adapts the direction of the group. After they meet they stick together, but single agents can be separated when another group with a different movement direction passes by or at some interactions with the walls.

3.5.3 VeloModel

This model is used to test whether agents can do tasks that require information from steps in the past. In this model the velocity is affected by the amount of agents seen in the last 40 time steps, hence the name VeloModel. The model has similar rules to the SimpleCouzin model with slight additions to the model. Here are the additions:

- If an agent in in either one of the three zones adjust the speed for the next 40 steps by gradually adding movement speed to the next future steps and then slowing down to base velocity of 1 again. (This effect stacks which means if in 10 consecutive steps an agent is nearby the velocity changes add up). The changes affect the next 40 steps.

A snapshot of agents moving according to these rules can be seen in figure 3.5. Agents again move around and avoid walls, like in the SimpleCouzin model. When agents meet they slowly

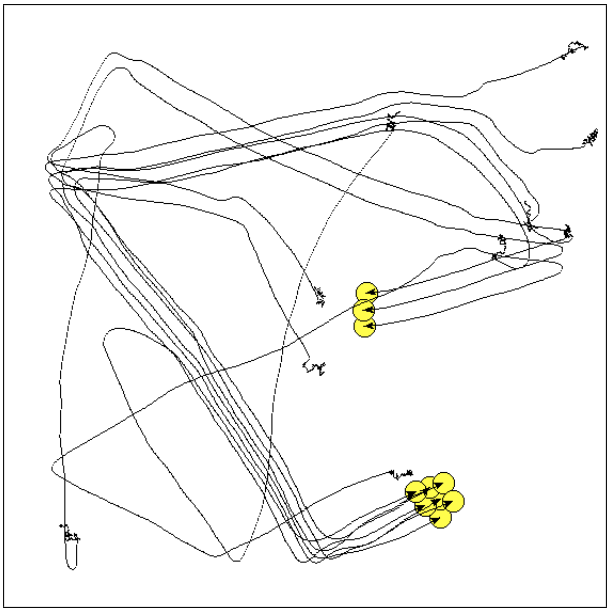

Figure 3.4: Snapshot of the simulation running with the SimpleCouzin model.

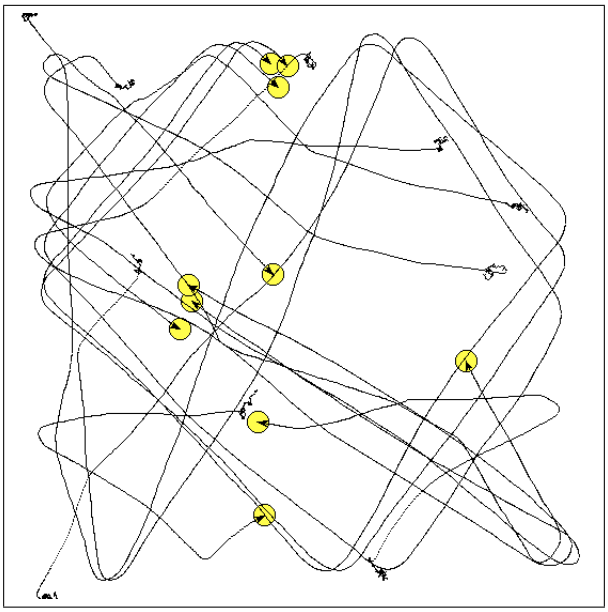

Figure 3.5: Snapshot of the simulation running with the Velocity model.

adjust their movement direction while also gaining speed. This can lead to them either starting to move together with high speed, or moving away from each other with a little acceleration. Once agents lose sight of neighbours they slow down again. This results in agents moving in smaller groups because it is harder for agents to attach to a fast moving group.

3.5.4 ComplexModel

In this model we add additional difficulty to the model by changing the size of the zones depending on distinct neighbours seen. This is hard for the Prediction model to learn since distinctions between neighbours do not exist in the data model. Here are the rules of this movement model:

- Agents walk in a straight line with a base velocity of 1.
- Assign all neighbours to one of three zones based on the distance.
- If at least one neighbour is in ZOR do the following:
  - For every neighbour is in ZOR (Zone of repulsion): Adjust the direction to move away from that neighbour
- If no neighbour was in ZOR and at least one neighbour is in either ZOA or ZOO do the following for every agent in the zones:
  - If the neighbour is in ZOA (Zone of attraction): Adjust the direction to move towards that neighbour
  - If the neighbour is in ZOO (Zone of orientation): Adjust the direction to move in same direction as the neighbour.
- If a wall is closer than 45 units gain an acceleration away from the wall. Acceleration speed is dependent on the distance to the wall. This can temporarily change the velocity of the agent, but when the wall gets out of sight the velocity will return to the normal velocity.
- If an agent in in either one of the three zones adjust the speed for the next 40 steps, by gradually adding movement speed to the next future steps and then slowing down to base velocity again. (This effect stacks which means if in 10 consecutive steps an agent is nearby the velocity changes add up)
- If in the last 40 timesteps at least 4 different agents are seen do the following:
  - The ZOR is extended to the whole visual range, hence ZOA and ZOO disappear.
  - The agent slows down slightly in the next step, but won't go slower than 0.5.

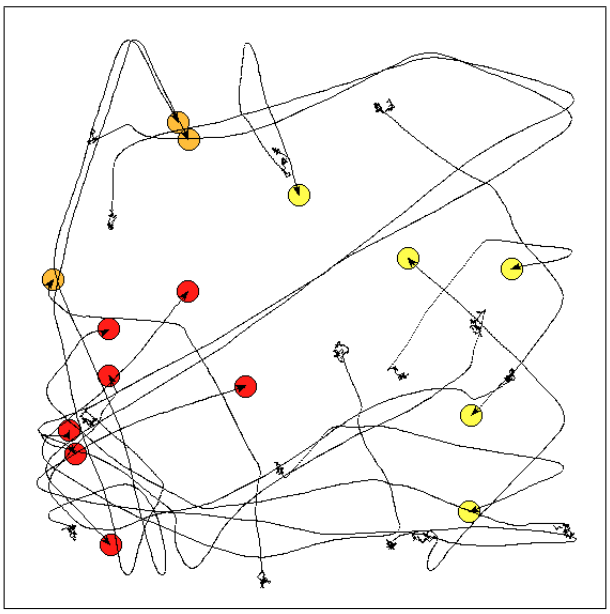

Figure 3.6: Snapshot of the Simulation running with the Complex model.
Agents are colored according to the distinct agents they met in the past 40 timesteps. Yellow means they did not meet any agents in the last 40 timesteps. Red stands for meeting at least 4 agents in the past 40 timesteps. Orange stand for 1-3 agents.

In figure 3.6 a snapshot of the simulation running the ComplexModel is shown. Again when agents move alone they act like in the SimpleModel. Once two agents meet they start acting like agents in the Velocity model. If groups are formed and agents see too many other neighbours they switch into repulsion mode, where they try to move away from every agent around them, while also slowing down. In figure 3.6 that is what happened to the group of red colored agents. Some of the agents already left the group while the others are trying to do the same. Two agents that met and act like in the Velocity model can be seen at the top and are colored orange. The yellow agents are moving without neighbours in sight.

3.6 Data generation

The simulation can be run with a variety of parameters. The main parameters include visual bins, number of agents, algorithm used for agents, height and width of the world, number of timesteps the simulation will run and how many agents will be part of the simulation.

To generate enough data for training and testing the simulation is run multiple times. The parameters used are the following:

- vision_bins = 16
- num_agents = Randomly chosen between 1 and 35

- num_timesteps = Randomly chosen between 300 and 1500
- borders = Randomly chosen between 300 and 600
- algorithm = Choice of one of the the models from last section

For each model a train set and a test set is generated. In order to generate the train set the simulation for each model is run around 300 times. The history of each agent is used to generate windows that include 40 consecutive timesteps. The same is done for the test set but with around 100 simulation runs. It is important to run the models multiple times. In the case of the Couzin model groups form within the first 100 timesteps. This means that after a while the simulation only consists of agents that are already in a group. But forming groups is a big part of the model and to get enough data that shows how groups form the simulation needs to be run multiple times. The data generated like this will be used to train the Prediction model.

The data that is used for further analysis is generated a little differently. For the Prediction model data sets every possible window was used. This means that two consecutive windows look like this:

$$window_t = df_{t-39}, df_{t-38}, ...df_{t-1}, df_t$$
$$window_{t+1} = df_{t-38}, df_{t-39}, ...df_t, df_{t+1}$$

This means the data of two consecutive windows looks very similar. To avoid this for the data that is used for analysis a lot of windows are skipped when generating train and test set for analysis. Only every twentieth window is included in the train and test set for analysis. We will refer to this data as T-SNE and UMAP data.

3.7 Predicting trajectories - Prediction model

In this section the model for predicting future trajectories is explained and how the training of the model is done.

3.7.1 Prediction model

The Prediction model is supposed to predict trajectories for each movement model described in this chapter. The layout of the model can be seen in figure 3.7. It consists of a two layered Long Short-Term Memory (LSTM) network and a Mixed Density Networks (MDN).

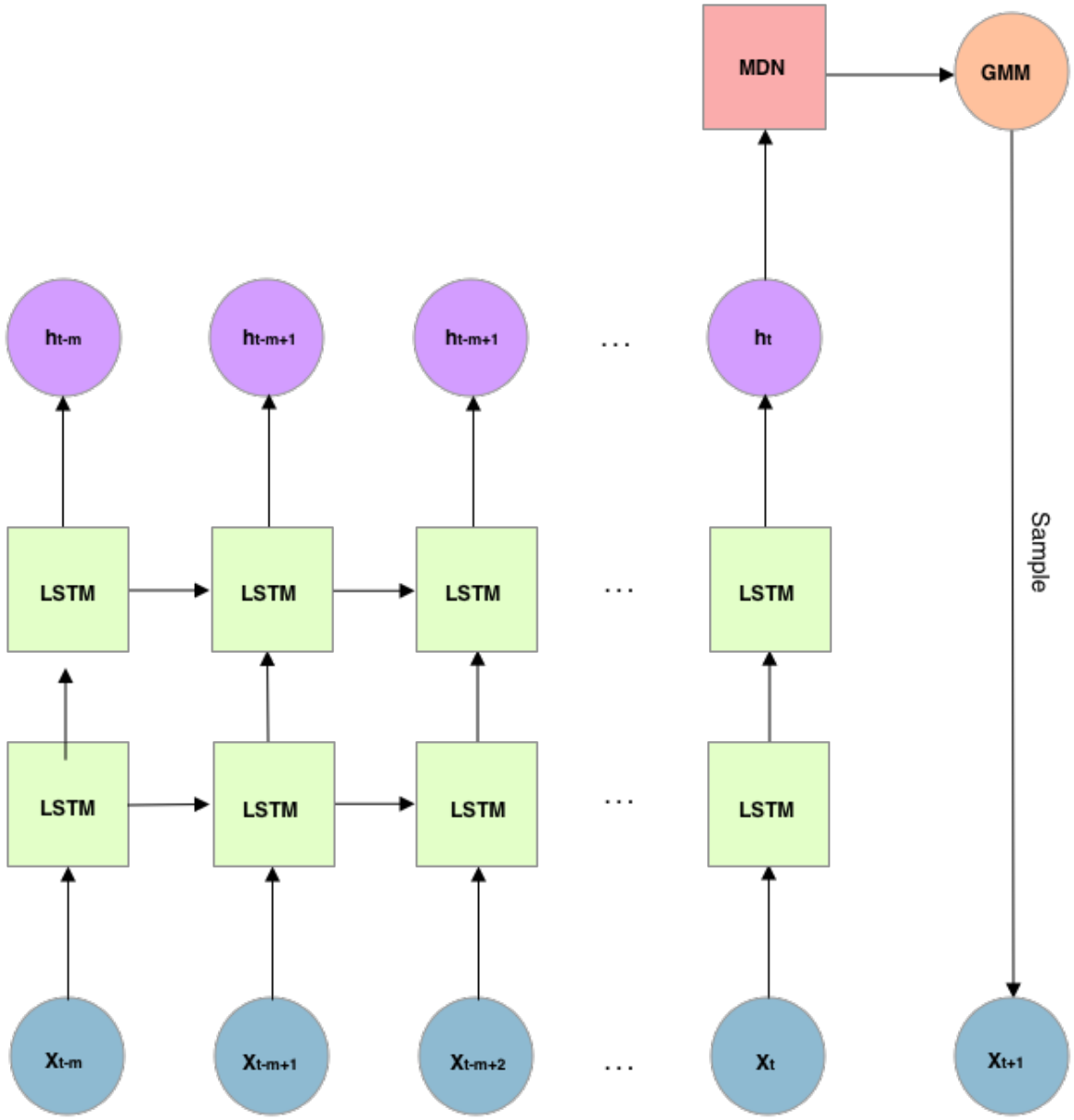

Figure 3.7: Trajectory predictor.
The layout of the neural network that will predict future trajectories. The input to this model is time series data x_{t-m} to x_t. The output is an Gaussian Mixture model (GMM), that can be used to sample the trajectory for the next timeframe x_{t+1}. The network conists of a two layer LSTM and an MDN.

The input to the model are the windows described in the last section. The output is the prediction of the movement vector and the rotation degree for the next step.

By using a Recurrent Neural Network (RNN) we are able to process time series data with arbitrarily many time steps. RNNs have a problem with vanishing gradients when applied to long time series data. Since the more complex models do have time dependencies that go back multiple timesteps and we expect that bee movement is dependent on input from the past we choose LSTMs. LSTMs were described by Hochreither and Schmidhuber (1997)[10] and are able to preserve long term memories. The 2-layered LSTM processes the input and calculates the hidden states h_{t-m} to h_t. Only h_t will be further used as the input for the Mixture Density Network (MDN).

MDNs were described by Bishop (1994)[11] and are used to predict a probability distributions. In our case we predict the parameters of a Gaussian Mixture Model, which gives us a probability distribution over all possible movement vectors and orientation changes. The GMM we use will predict the parameters for five Gaussian distributions and the weights each Gaussian distribution has in the GMM. This is done by using the hidden state as an input and then calculating all the components of a Gaussian mixture models.

One could also predict the trajectories directly. But the problem with that approach is that agents might have multiple valid choices at a time. An simple example of this is demonstrated in figure 3.8. There are other ways to predict probability distributions. In particular (Eyjolfsdottir et. al)[1] used equally sized bins to split all the directions the fly could move to in the next step. In the paper of (Graves, 2013)[12] Gaussian Mixture models together with RNNs were used to generate time series data. We use a very similar model.

The network can be trained by maximizing the probability of the real prediction inside the predicted Gaussian mixture model. This is done by minimizing the negative log-likelihood. The error function for the Prediction model will later be called MDNLoss.

One could also try to use Mean squared error loss (MSELoss) by using the mean prediction of the Gaussian mixture model. To see a problem with this approach lets look at figure 3.8 again. Lets assume the Gaussian predicts each of the two options described previously with a probability of 50%. The mean of that prediction would be in the middle of the two predictions, which means the error would be equally high to a probability distributions that predicts with a 100% probability to go straight through the obstacle. This is unsound and not a good error function to optimize this model. Nonetheless the MSELoss can be a good indicator for testing to see if the network is learning to go in the right direction, since negative log-likelihood is not as intuitive as a distance function.

After predicting the probability distribution it is easy to sample movements according to their probability and we are able to run the simulation using the network.

Figure 3.8: An illustration of an agent that has two possible trajectories.
The agent is represented as a yellow circle with an arrow. In this case the agent wants to go to the red cross and needs to avoid the grey obstacle. It needs to go around the obstacle which leaves two options, to pass the obstacle by going right or left. Both routes look equally good and agents might randomly do either one of them.

3.7.2 Training

As already stated we use the windows generated with the simulation for training. windows will be generated with 40 timesteps. The first 39 timesteps will be used as the input to the Prediction model, while the values in the 40. window will be predicted. We run the training with the whole train set, using a batch size of 32 and going through the train set at least once. During the training we use the test set to check the performance of the network at different stages of the training. This can help to see if the network is over fitting or learning to minimize the error at all.

3.8 T-SNE and UMAP

Using the hidden states of our Prediction model we can use T-SNE and see if certain classes get clustered. To do this we run our network on the test sets specifically generated for T-SNE and UMAP.

T-SNE is an algorithm to project high dimensional data on a 2D plane described in (Maaten & Hinton, 2008)[7]. The scipy implementation of T-SNE is used for the projections used in this work.

UMAP is also an algorithm to project high dimensional data into lower dimensions and is described in (McInnes, Healy & Melville, 2008)[8]. The umap-learn[9] implementation is used for the projections made in this work.

The data that will be projected are the values form the hidden states of the Prediction model. Again our data consists of windows that contain 40 timesteps each. The first 39 timesteps are used as the input for the Prediction model. We ignore the prediction and only use the hidden state h_t from figure 3.7. To color in the T-SNE and UMAP projection the label from the 40th timestep is used. The data used for T-SNE and UMAP are therefore only the hidden states and the labels from the last timestep.

With UMAP we want to achieve the same goal as with T-SNE. UMAP has the option to find a projections that separate the labels in the projection. By using the same projection on different data we can test whether the projected test data still shows separated labels. If so we assume that classes can be fairly easily separated.

3.9 Label Classification

The goal for the network was to use the hidden states to predict labels. In this section the advantages of having these hidden states are tested.

The task to solve is the following: Given trajectory data, predict a label for the next timestep. The label will be the number of timesteps in which an agent was inside one of the three zones. Only the occurrences in the last 39 timesteps will be counted for the label, therefore the maximum number of occurrences is 39. This means the task is a regression problem.

Four different models will be used for this. The first models uses the hidden states of the trained Prediction model. The hidden states are calculated and then used as the input for a simple classification model. The PyTorch model of the simple model looks like this:

```
self.model = torch.nn.Sequential(
            torch.nn.Linear(input_dim, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, output_dim),
```

Since the last hidden state from the RNN has dimension 256, the input dimension of this simple model is also 256. The output dimension is 1, because a single number is predicted. A linear network is used with 3 linear layers with ReLU activation functions in between.

The second model uses the raw window data. The model consists of an RNN and a linear network that is solely trained to predict the label and the code looks like this:

```
self.rnn = torch.nn.LSTM(n_features, 128, num_layers = 1)
self.out = torch.nn.Sequential(
    torch.nn.Linear(hidden_dim, 256),
    torch.nn.ReLU(),
    torch.nn.Linear(256, 128),
    torch.nn.ReLU(),
    torch.nn.Linear(128, n_outputs),
)
```

This model takes the same input as the Prediction model. The output is again a single number.

The third model also uses the raw window data. It is a very simple model and trained to predict the label and looks like this:

```
self.model = torch.nn.Sequential(
    torch.nn.Linear(input_dim, 256),
    torch.nn.ReLU(),
    torch.nn.Linear(256, 128),
    torch.nn.ReLU(),
    torch.nn.Linear(128, output_dim),
)
```

The input from the raw data is given as an 38*133 long vector. As can be seen from the code its just 3 linear layers with Relu activation functions.

The last model only uses the last timestep of the raw window data. The model looks like this:

```
self.out = torch.nn.Sequential(
    torch.nn.Linear(n_features, 256),
    torch.nn.ReLU(),
    torch.nn.Linear(256, 128),
    torch.nn.ReLU(),
    torch.nn.Linear(128, n_outputs),
)
```

The input to this model is a 133 long vector. The output is a single number.

The training and testing of these model will be done with the train and test set generated for UMAP and T-SNE. These tests are done to see if the hidden state of the Prediction model contains useful information about the windows. We also want to test the influence of the

amount of labeled data used for training. Our original goal was to use as little as possible labeled data, since getting labels is expensive. Therefore multiple tests with different amounts of training windows are done.

Chapter 4

Results

In this chapter the results of the training will be presented. First the performance of the Prediction Models on all the simulated models will be presented. Then the results of the training with bee data will be presented. The description of the training of the SimpleModel will be the most detailed. The error plots, T-SNE plots and UMAP plots for later models are generated in the same way, which is why explanations will be shortened.

In the end of the chapter two of the Prediction models are used to solve a classification task and compared with more traditional classification models.

4.1 Training of Simple Model

As described in chapter 3 the simple model just makes the agents walk in straight lines, while avoiding walls.

During the training phase the MDNLoss was saved. Since intuitively it is not easy to understand what the MDNLoss means the MSELoss described in chapter 3 was also tracked. The progression of the errors can be seen in figure 4.1. During the training the test set was used to track the performance of the network on different data. The progression of the error on the test set can be seen in figure 4.2.

Both plots show that the error continually went down. MSELoss reached almost 0, while MDNLoss reaches close to -14.5. This is a good value and means that the Prediction model predicts the right trajectory with a high probability.

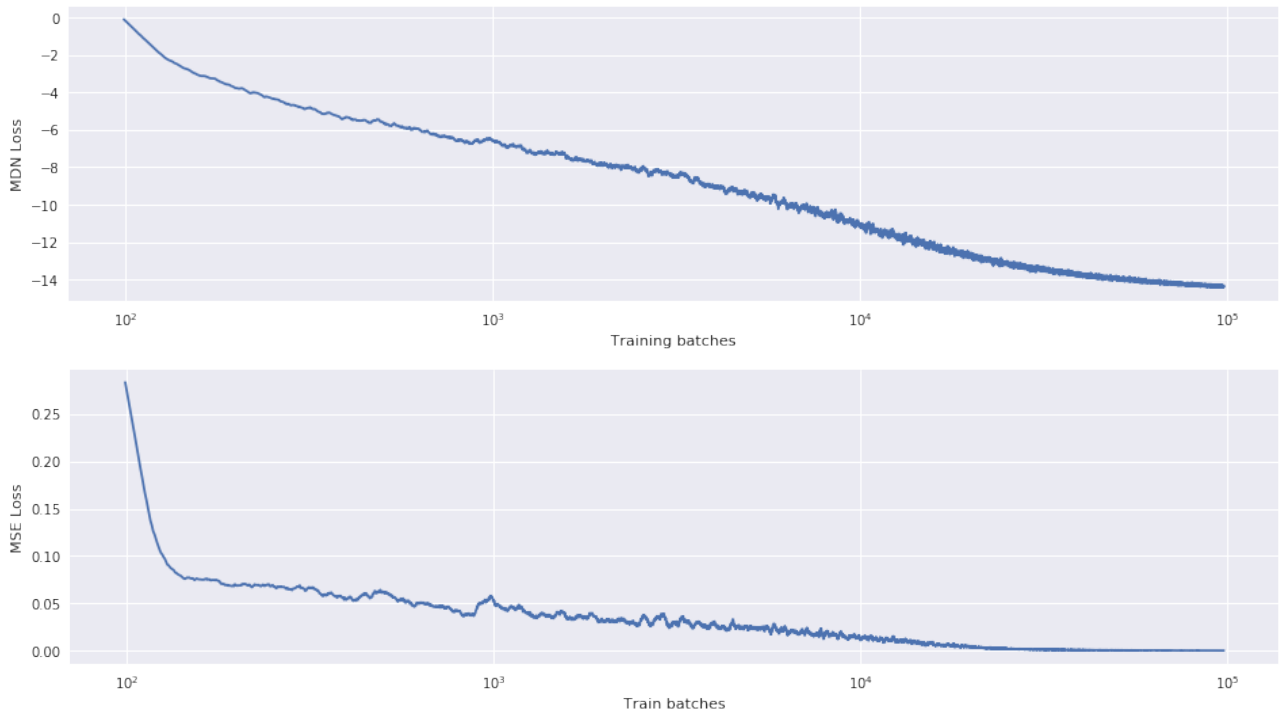

Figure 4.1: Training error while training the Prediction Model with SimpleModel data. Both use a logarithmic scale for the x-axis. The upper graph shows the MDNLoss and the bottom graph shows the MSELoss.

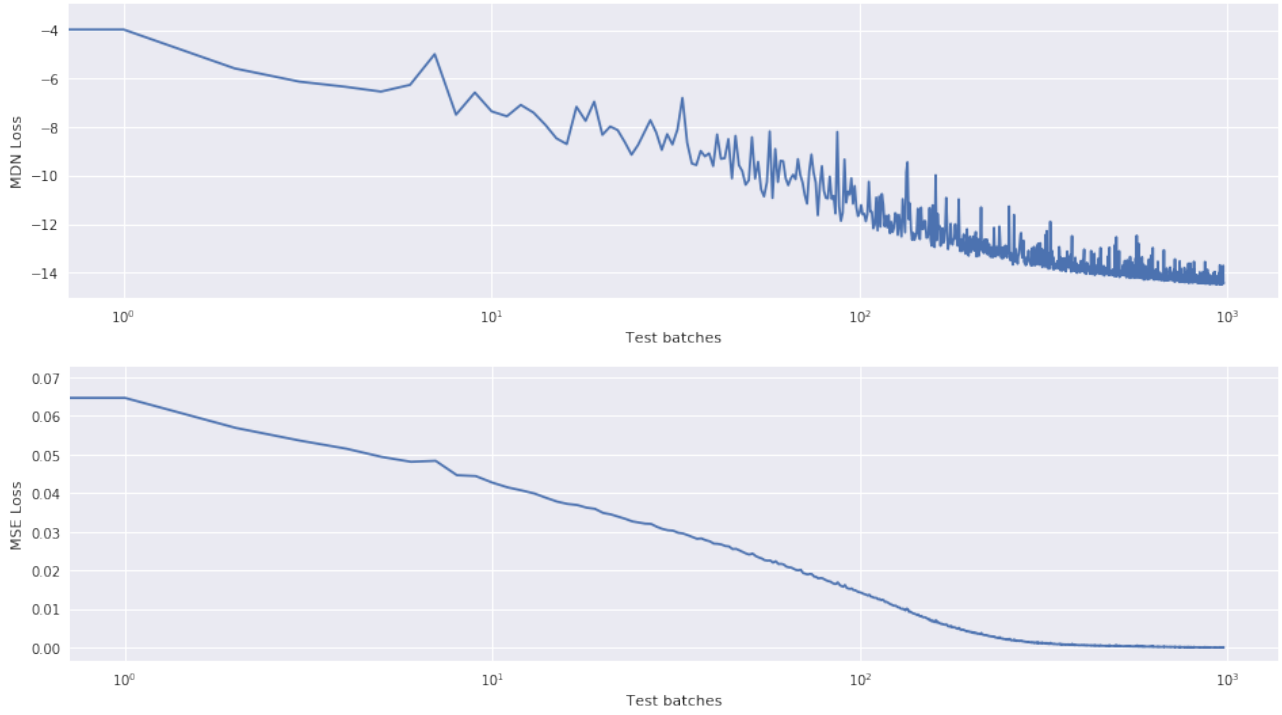

Figure 4.2: Testing error while training the Prediction Model with SimpleModel data. The two graphs show the losses on the test set during the training of the neural network. Both use a logarithmic scale for the x-axis. The upper graph shows the MDNLoss and the bottom graph shows the MSELoss. The loss was calculated every 100 training batches.

4.1.1 Running the Prediction model

To see the Prediction model in action the simulation was run with the trained Prediction model. In figure 4.3 snapshots of the simulation are shown. The agents are moving according to the trained Prediction model. Since the output is a gaussian Mixture model the future trajectories need to be calculated first. In the left image the agents calculate the trajectory by randomly sampling form the Gaussian mixture distribution. As can be seen the agents move in a similar way to the real simulated agents. The difference when compared to 3.2 is that the lines are not perfectly straight. They have a slight curvature. Also the avoidance of the wall shows that the incoming angle is not always close to the outgoing angle, which is the case in the Simple Model. Overall the network learned the most important features of the Simple Model. In the right image the agents move by using the mean of the Gaussian mixture distribution, which is calculated by adding the means of each Gaussian and then add them with the weights attached to each Gaussian distribution. This simulation run yields a similar result.

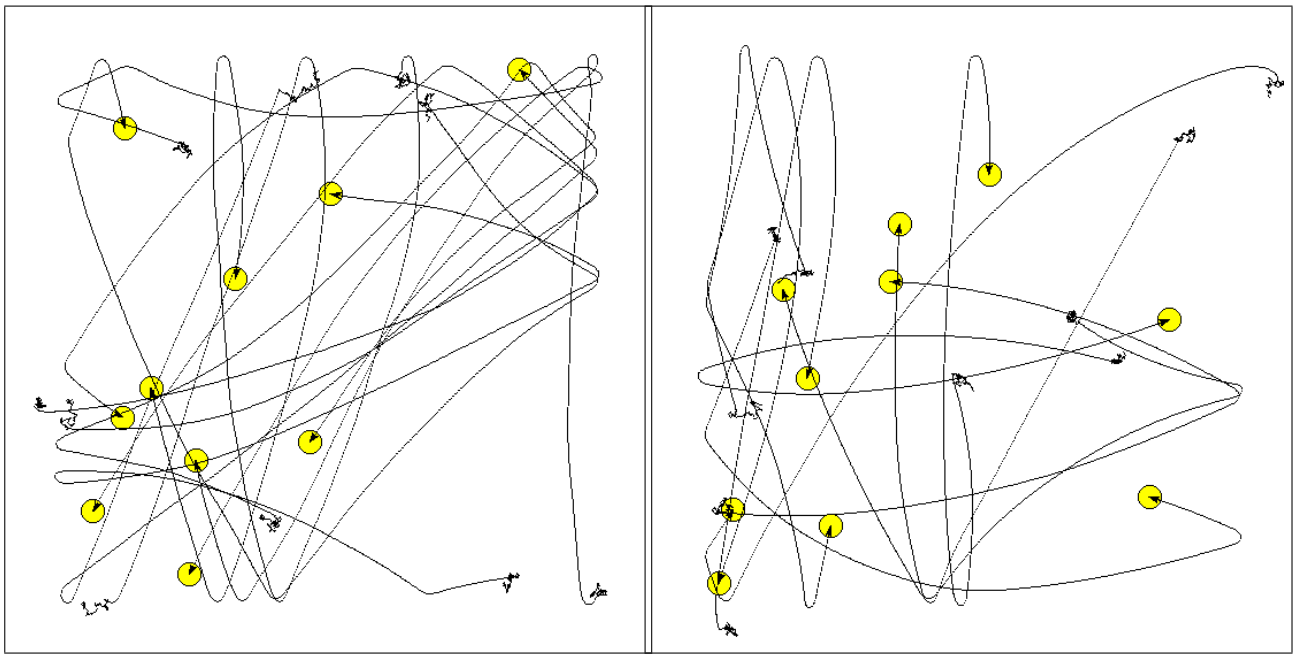

Figure 4.3: Simulation running the trained Prediction Model for SimpleModel.
In the left image the predictions are made by sampling from the probability distribution. In the right image the predictions are made by using the mean of the probability distribution.

4.1.2 T-SNE and UMAP

The Simple model does not offer a lot of attributes that could be interesting for T-SNE or UMAP. We will see how the walls are separated in the T-SNE projection and later try to train UMAP to separate all wall avoidance movement from normal movement.

Therefore we generate labels for that. Whenever a wall is within the maximum visual range

of an agent we label the step as a wall avoidance step. Otherwise it is labeled a normal step. T-SNE was run with 100000 windows, which means the Prediction Model predicted 100000 trajectories. Then the hidden state h_t from figure 3.7 was saved with the corresponding labels. Using only the hidden state the projection was made and after that the points were plotted and colored according to their label. The result in figure 4.4 show the T-SNE projection. As can be seen the the avoid wall label is separated very well from the normal walk label.

For UMAP a training and a test set was used. With the training set the projection was trained to separate hidden states according to corresponding labels of the windows. The test set was then used to see if the labels are still separated on unseen data. As can be seen in 4.5 this is the case. We got similar results as in T-SNE. Most labels are grouped at one place and there are a few red points in the mostly blue ring.

Looking at the predicted trajectories of the trained network it seems the most important features of the model were learned. The agents move with a constant velocity and avoid hitting the wall. There are slight problems with walking in a perfectly straight line and the wall avoidance does not produce trails with the correct in and out angles.

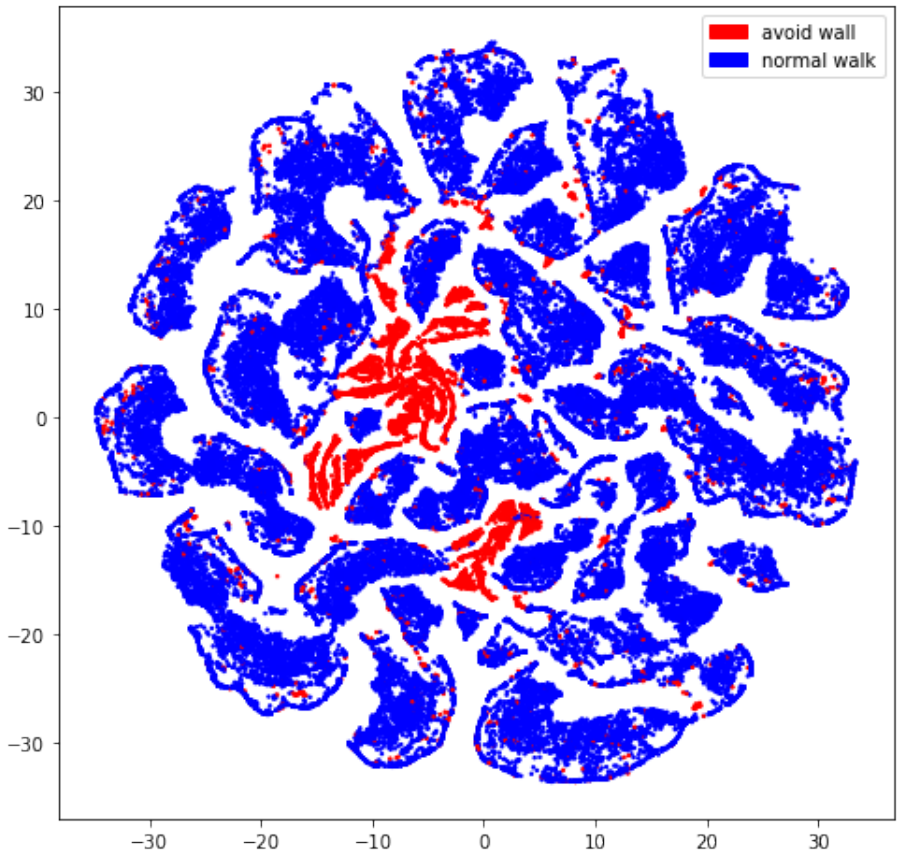

Figure 4.4: T-SNE of Simple Model
T-SNE projection run with 100000 windows. Blue points represent 'normal walk' windows and red points represent 'avoid wall' label.

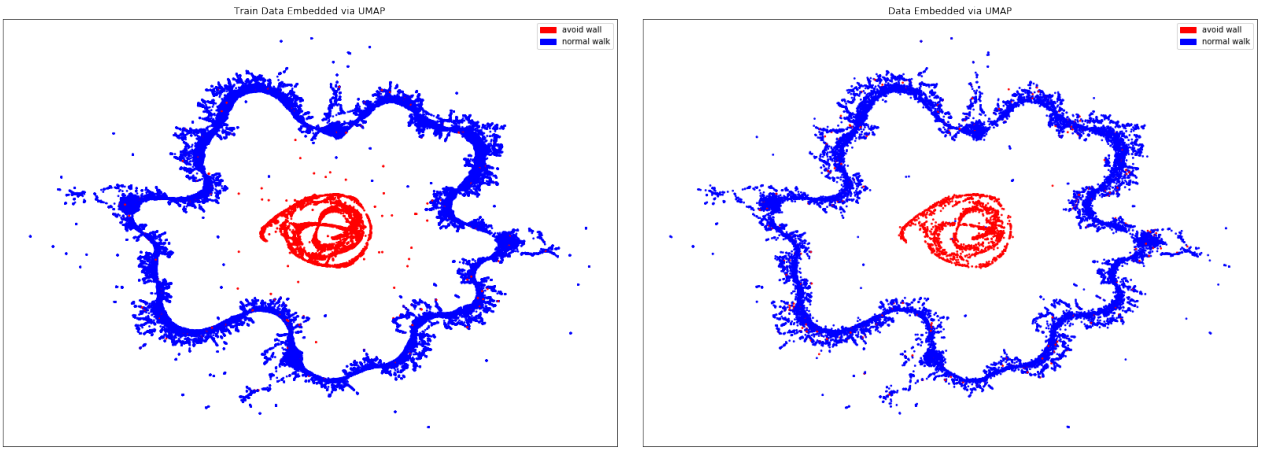

Figure 4.5: UMAP of Simple Model
The left data is plotted with the trained projection. On the right the projection of the test set is shown.

4.2 Training of Couzin Model

As for the Simple Model the MDNLoss and MSELoss during training phase are saved. The performance on the test set can be seen in figure 4.6. The error went down fairly quickly again but this time did not go as far down as for the Simple Model. The MSELoss reached almost 0 again.

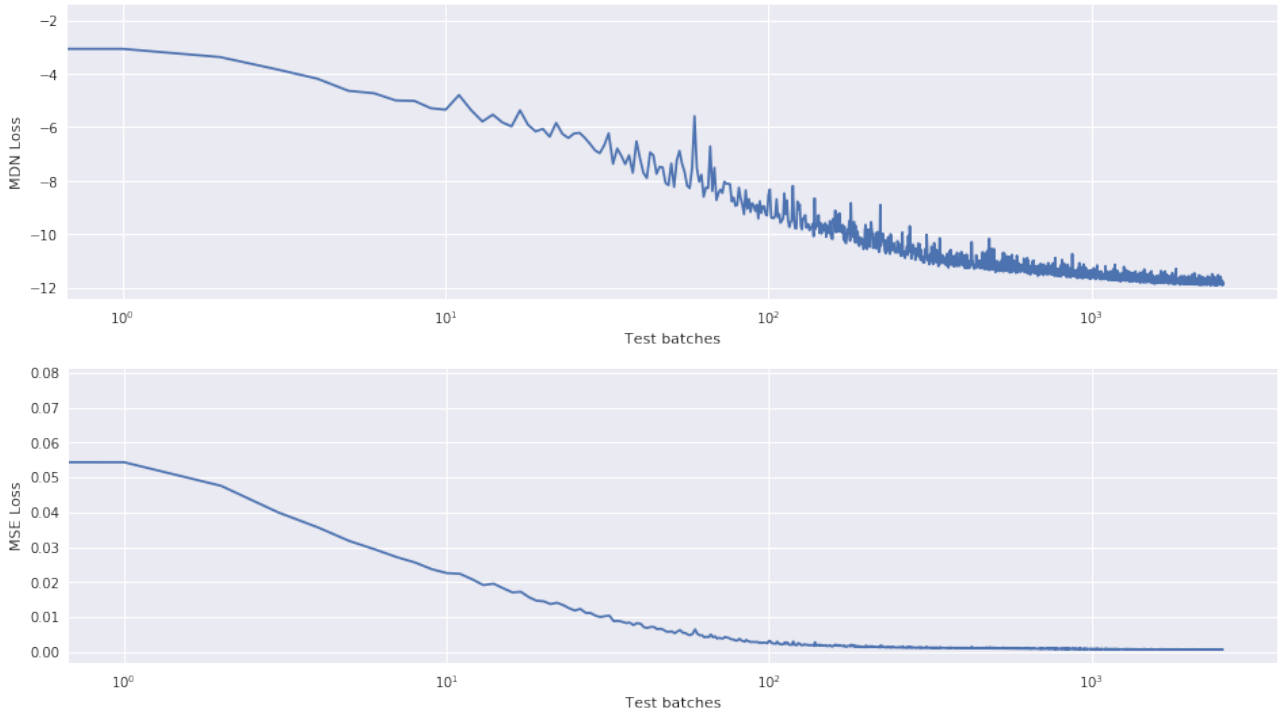

Figure 4.6: Testing error while training the Prediction Model with CouzinModel data.
The two graphs show the losses on a test set during the training of the Prediction model. Both use a logarithmic scale for the x-axis.

4.2.1 Running the trained network

In figure 4.7 we see snapshots of the simulated agents, that are now moving according to the trained network. Again we see that important properties were learned. The agents move in almost straight lines and avoid walls. When multiple agents meet they form a group and move together. In the Simple Couzin model a random noise was added to the movement, which explains why this time the trails do not look perfectly smooth and agents switch their movement direction. When looking at the mean prediction we see that agents seem to have a bias to move towards the right.

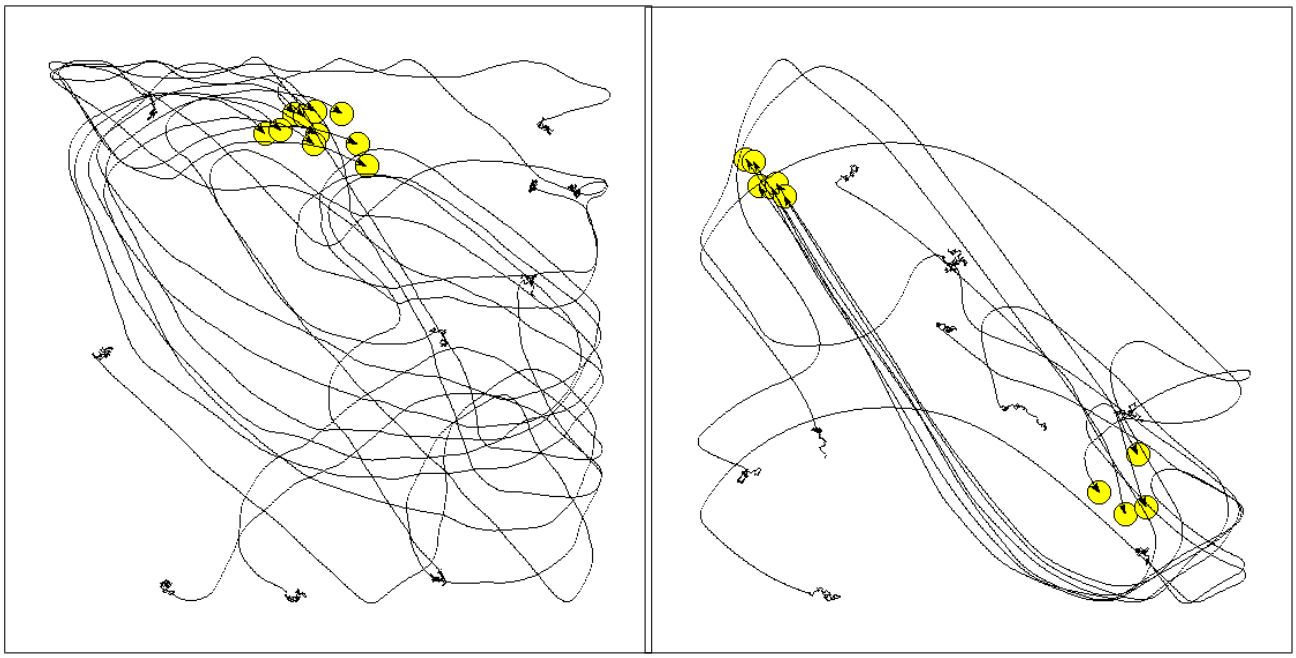

Figure 4.7: Simulation running the trained Prediction Model for CouzinModel.
Trained network simulating agents for the Couzin model. In the left image the predictions are made by sampling from the probability distribution. In the right image the predictions are made by using the mean of the probability distribution.

4.2.2 T-SNE and UMAP

In order to do T-SNE and UMAP labeled data is needed. We introduce new labels. If there an agent is avoiding the wall he gets an wall avoidance label. If a neighbouring agent is in the zone of repulsion, the agent gets a ZOR label. Equally we define labels for the zone of orientation and the zone of attraction. This means that we can have multiple labels per window, since an agent can be in all 4 states at the same time. This makes coloring a little harder since each point in the T-SNE projection could have multiple labels assigned to it. We encode the labels with 4 biniary digits as follows: ZOR - 1000, ZOO - 0100, ZOA - 0010, avoid wall - 0001. This way if multiple labels are present it is easy to represent them, e.g. if ZOR and avoid wall are active, the corresponding label is 1001. In figure 4.8 we can see the distribution of label

combination across the windows in the train and test set. The color coding of the barplot will be used for later T-SNE and UMAP projections. To see how the labels are distributed we do the T-SNE Projection and plot the projection multiple times to color in different labels. In 4.9 we see the projection 6 times. As can be seen there is some structure in the plots. The normal walk (0000) label is isolated. ZOR, ZOO and ZOA share a lot of space in the projection. In the last plot in the figure the combinations of the labels are used for coloring. Some regions of the plot are mainly occupied by single colors.

Again a UMAP projection is trained. This time all the combinations of labels are tried to be separated. In figure 4.10 we see the result of the UMAP projection. The projection of the test set yields a similar result. The normal walk label is still separated very well, while ZOO mixes a lot with ZOR and ZOA, which results in a lot of orange and green colored points.

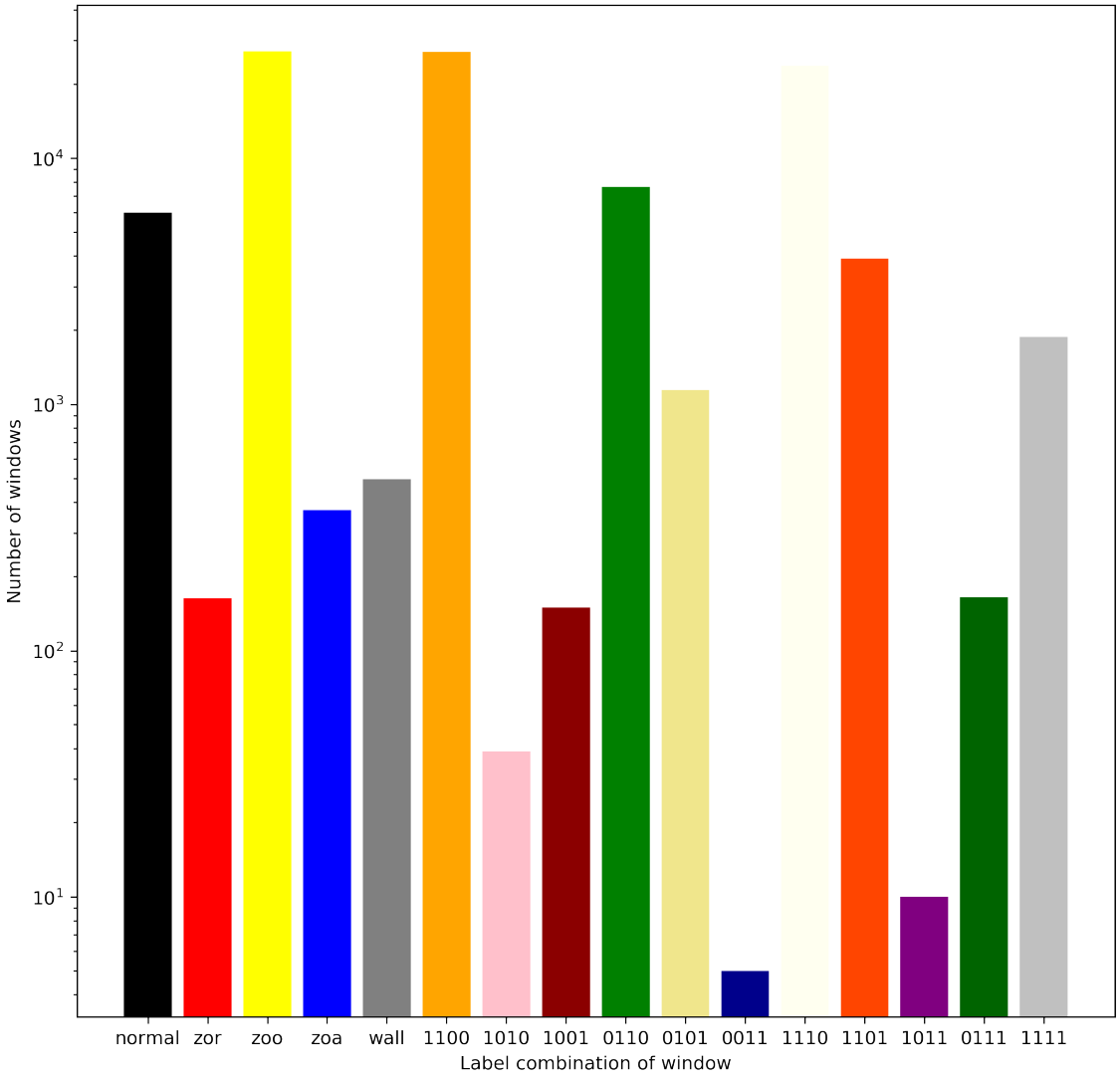

Figure 4.8: Histogram of label in train and test set for T-SNE and UMAP.
An histogram of all combinations of labels that the SimpleCouzin can have and how many windows with each labels are used for the train and test set. The color coding of the bar plot will be used in the T-SNE and UMAP projections to come.

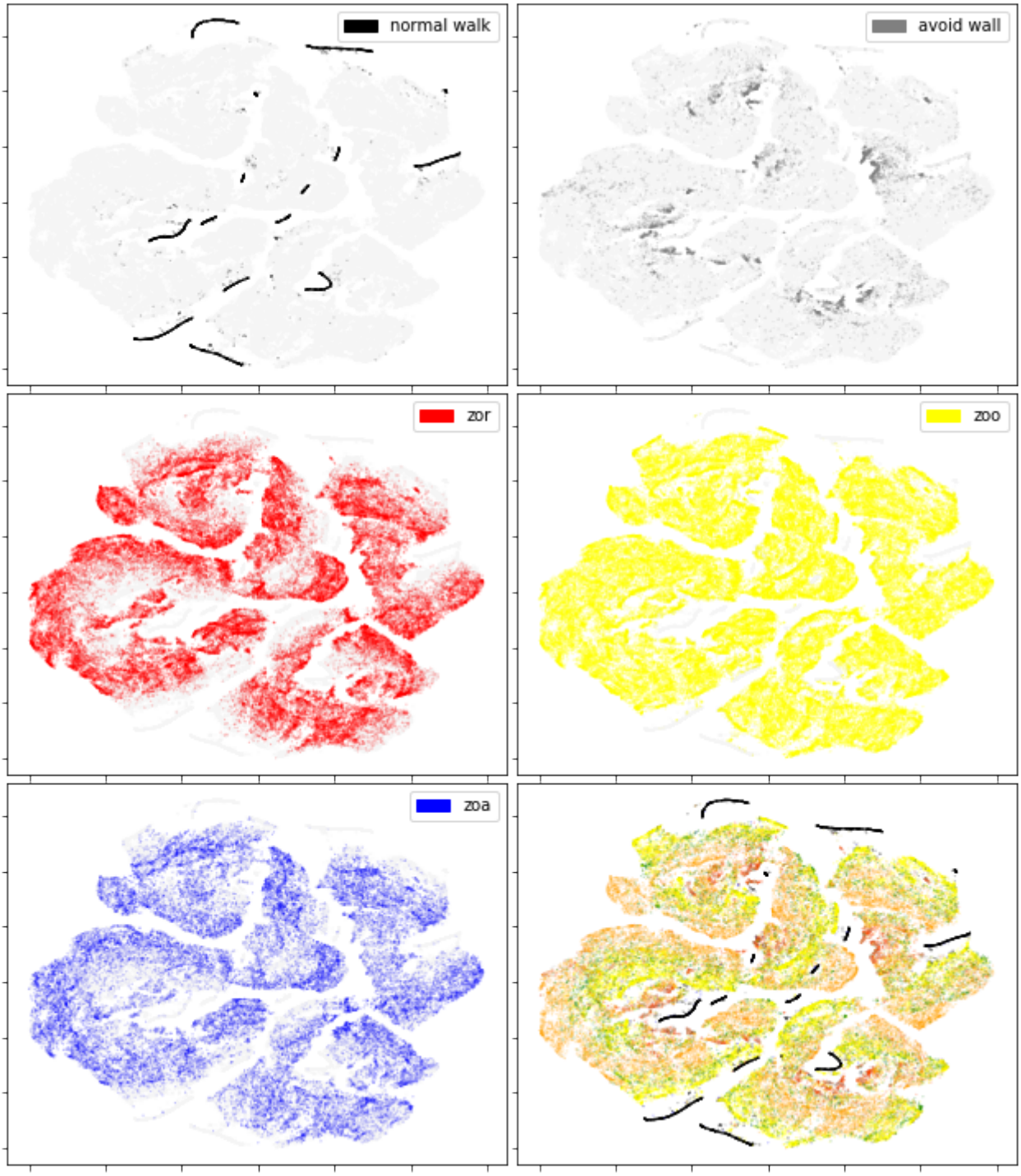

Figure 4.9: T-SNE of SimpleCouzin model.
Six plots with the same embedding. In the first five plots the base labels are colored in individually. In the sixth plot the combinations according to figure 4.8 are colored in with their respective colors.

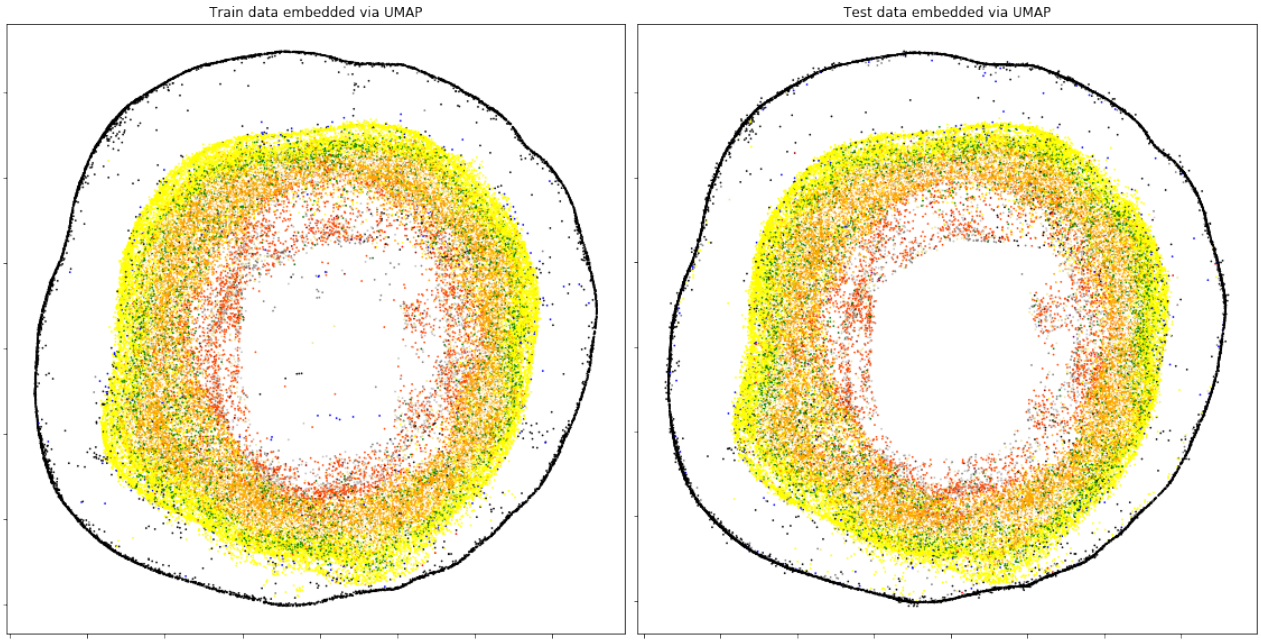

Figure 4.10: UMAP of SimpleCouzin model.
On the left the data which was used for training is projected and colored in, while on the right the test set was projected. Tge coloring was done according to the labels from figure 4.8.

4.3 Training of VeloModel

We approach the training the same way as for the previous two models. The error plots for the test set that were generated during training are shown in figure 4.11.

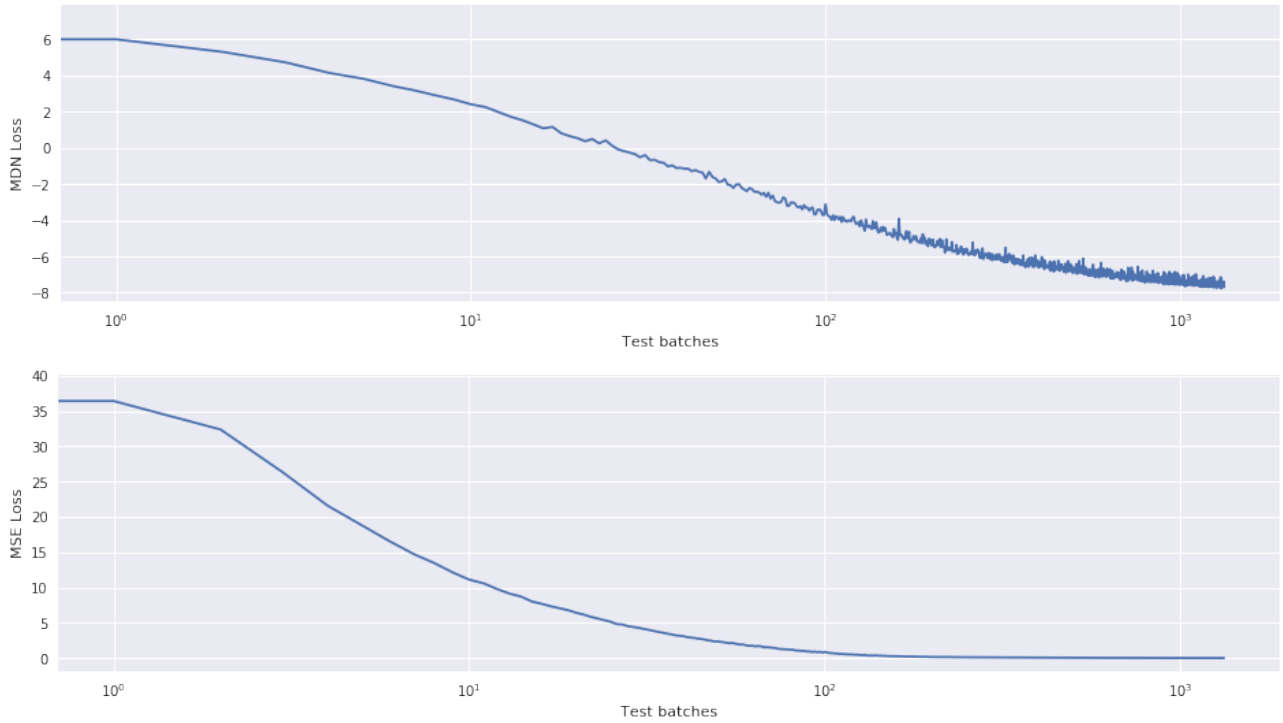

Figure 4.11: Testing error while training the Prediction Model with VeloModel data.

4.3.1 Running the trained network

Figure 4.12 shows the simulation running the trained Prediction model for the VeloModel. Agents still avoid walls and move around with constant speed if no other agents are around. Agents do change their speed when other agents get into their neighbourhood. When two agents meet they start accelerating and moving in a group. Bigger groups can also form but are rare, similar to the group behavior of the real model. Agents can lose groups due to interactions with walls or third agents. Once agents leave a group they slow down again. It seems the main features of the model have been learned, although the agents do not react as consistently to neighbours as in the real model. Sometimes agents can ignore a neighbour. There is also a bias in the trajectories, so that trails can get a curvature with influence of walls or other agents.

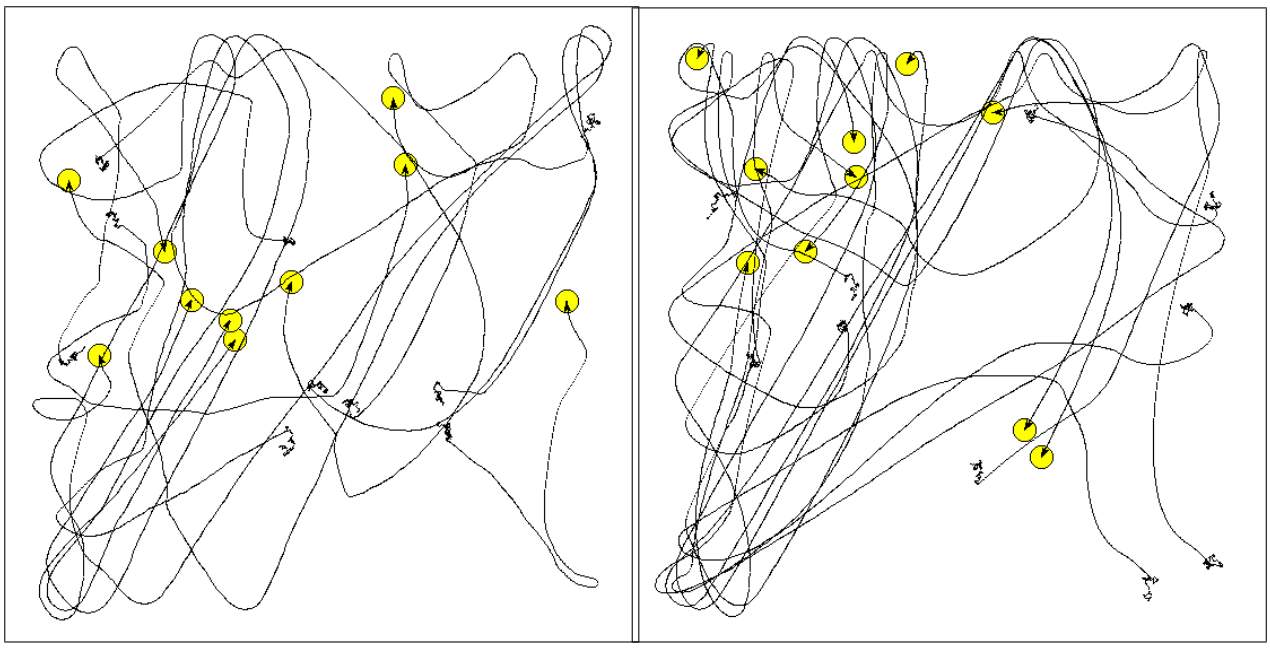

Figure 4.12: Simulation running the trained Prediction Model for VeloModel
Trained network simulating agents for the VeloModel In the left image the predictions are made by sampling from the probability distribution. In the right image the predictions are made by using the mean of the probability distribution.

4.3.2 T-SNE and UMAP

For the VeloModel the same combinations are used to color in labels as in the SimpleCouzin model. In figure 4.13 we see the T-SNE plots of the Couzin model. The normal walk is represented more than in the SimpleCouzin model, because in the VeloModel agents are less likely to form groups and can leave their groups again. Again ZOR, ZOO and ZOA share large sections of the projection. For the VeloModel we can generate new interesting labels. We generate new labels representing velocity at the end of the window, number of timesteps where neighbours have been present in the last 40 timesteps and distinct neighbours seen in the last 40 timesteps. The T-SNE projection colored in with these labels can be seen in figure 4.14. As can be seen in all three figures high label numbers can be separated from low label numbers.

Again a UMAP projection is trained with the combination of labels described in 4.8. The UMAP projections can be seen in figure 4.15. There seems to be some structure but mostly normal walk and wall labels can be separated from the other labels. The ZOR, ZOO and ZOA labels are mixed up in a large area inside the outer circle.

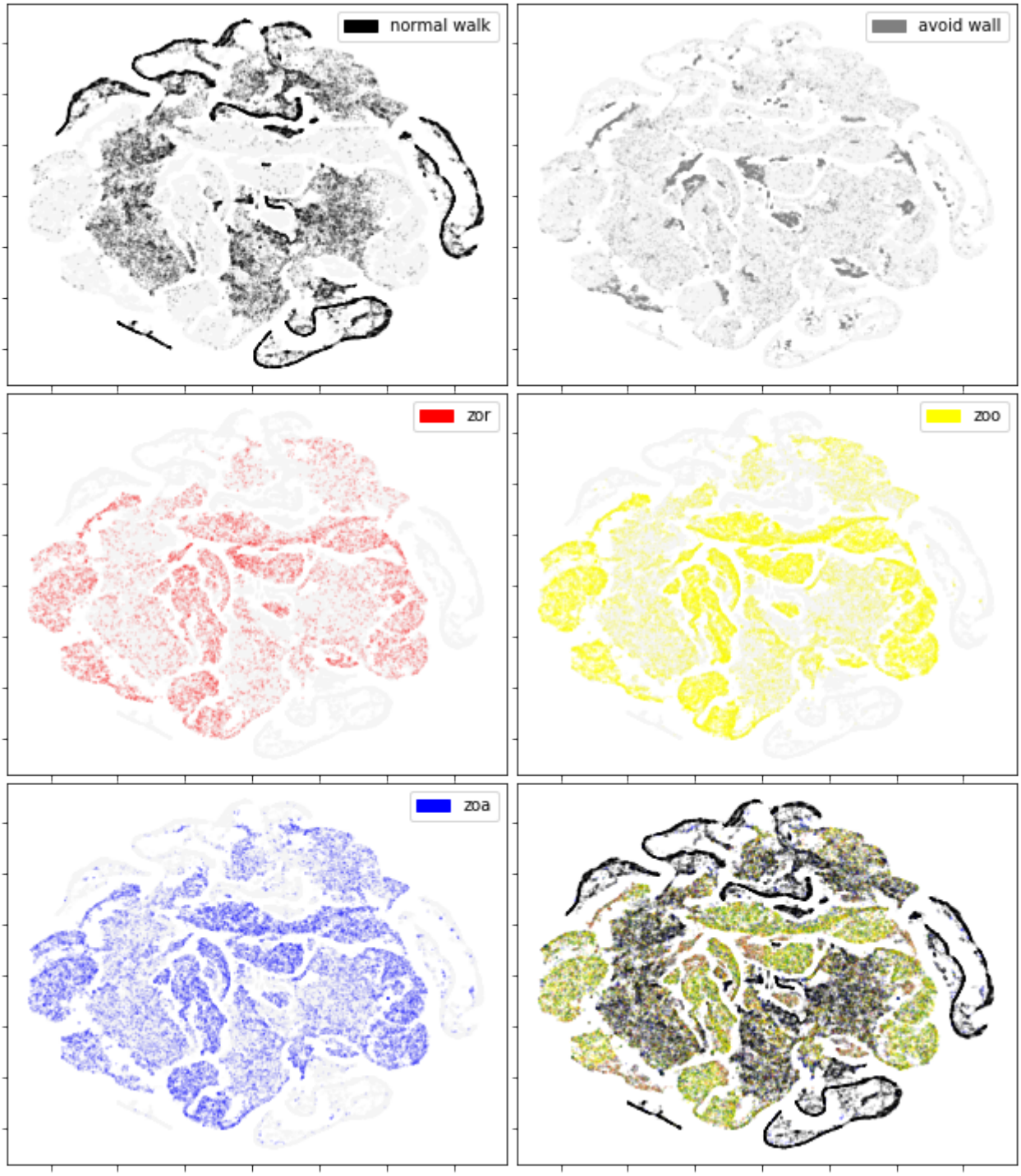

Figure 4.13: T-SNE Plots of the VeloModel
Six plots with the same embedding. In the first five plots the base labels are colored in individually. In the sixth plot the combinations according to figure 4.8 are colored in with their respective colors.

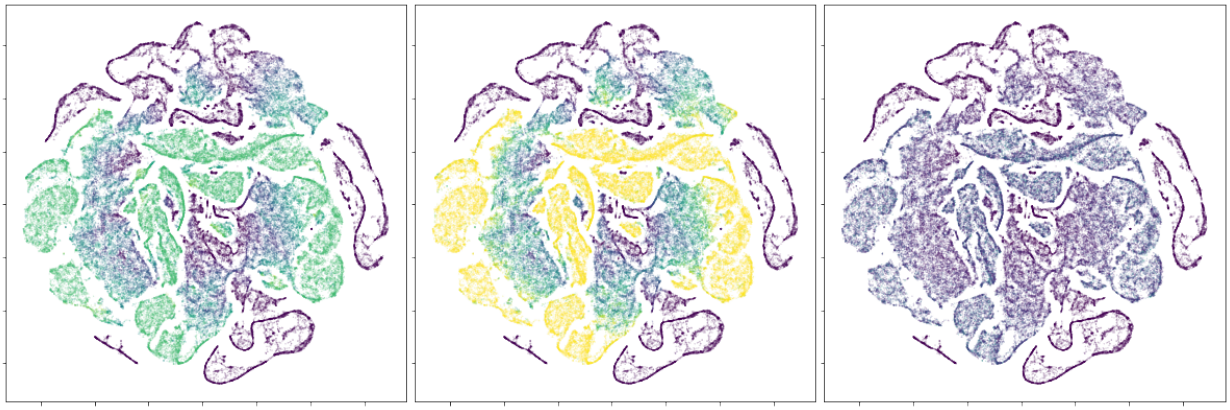

Figure 4.14: T-SNE plots for VeloModel with velocity and neighbour labels.
The left plot was colored in according to velocity labels, the middle one according to the number of timesteps where an agent was present and the right one according to the distinct neighbours in the last 40 timesteps. The color mapping for each of the images is the same where purple represents low values and high values are represented as yellow. Green is for values in between.

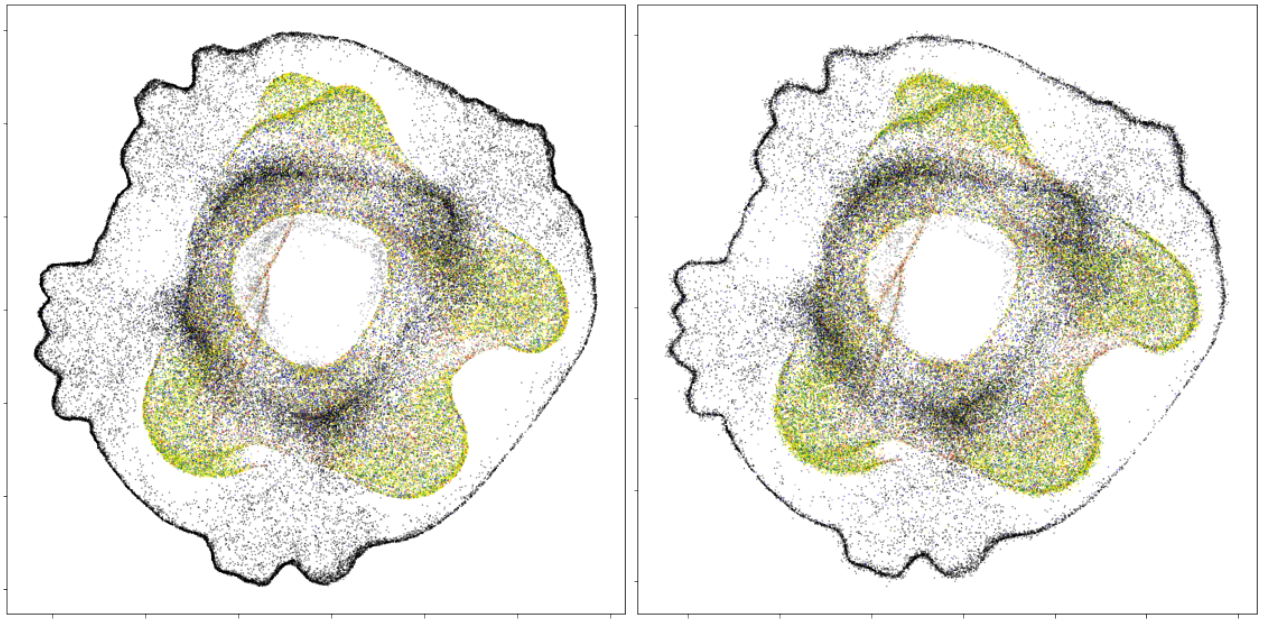

Figure 4.15: UMAP of VeloModel with zone labels.
On the left the data which was used for training is projected and colored in, while on the right the test set was projected.

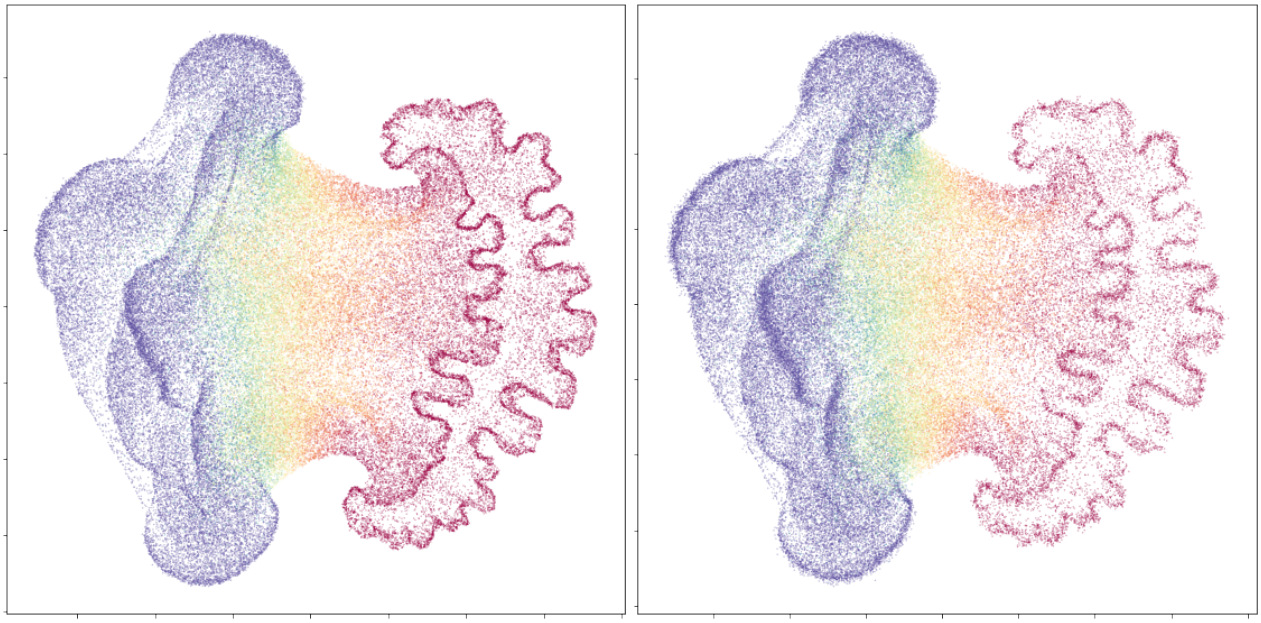

Figure 4.16: UMAP of VeloModel with number of neighbour occurrences in last 40 steps. On the left the data which was used for training is projected and colored in, while on the right the test set was projected.

4.4 Training of ComplexModel

The training is again done the same way. The error plots during training can be seen in figures 4.17 and 4.18. As can be seen the error fluctuates more than before. but is slightly lower than in the VeloModel. This will be discussed in chapter 5.

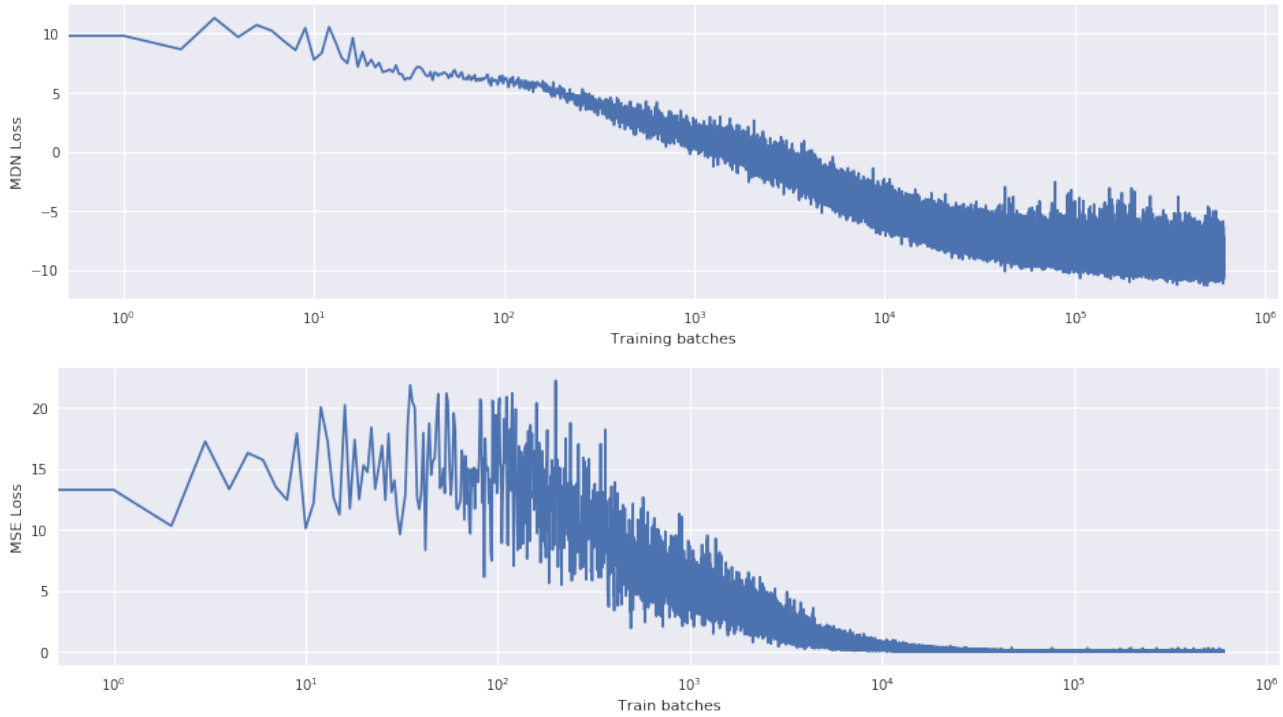

Figure 4.17: Training error while training the Prediction Model with ComplexModel data.

4.4.1 Running the trained network

In figure 4.19 a snapshot of the simulation running the trained Prediction model for the ComplexModel is shown. Again all the basic rules are learned, e.g avoiding walls and moving normally when no other agents are nearby. We color in labels according to the number of distinct neighbours in the last 40 steps of each agent. Red means at least 4 agents have been seen and yellow means no agent has been seen. Again agents change speed when other agents appear in neighbourhood and try to form small groups. In complex couzin agents in groups should only slow down if at least 4 distinct agents are seen in last 40 timesteps. But the network did not learn that perfectly. Sometimes an agent slows down, even though only one neighbour was seen in past 40 timesteps. It is also not apparent that the repulsion zone extends to the full vision. Agents maintain small distances between each other, but once repulsion mode should start they usually just slow down.

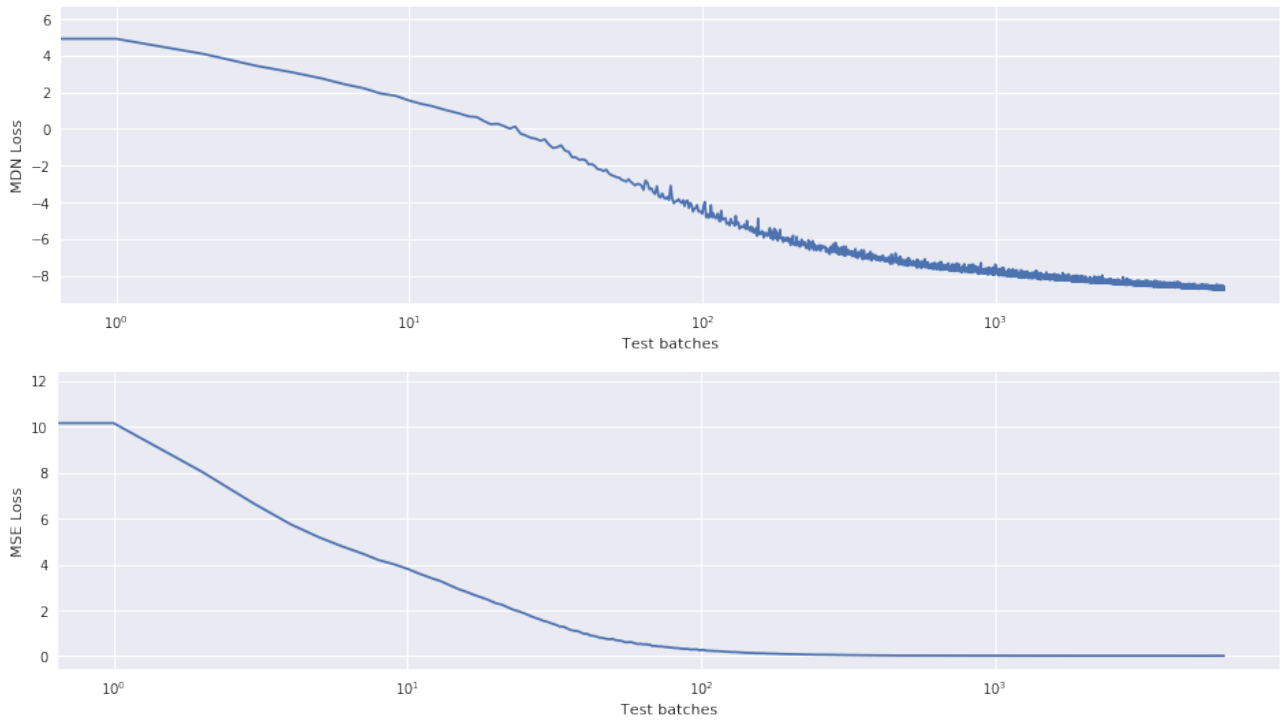

Figure 4.18: Testing error while training the Prediction Model with ComplexModel data.

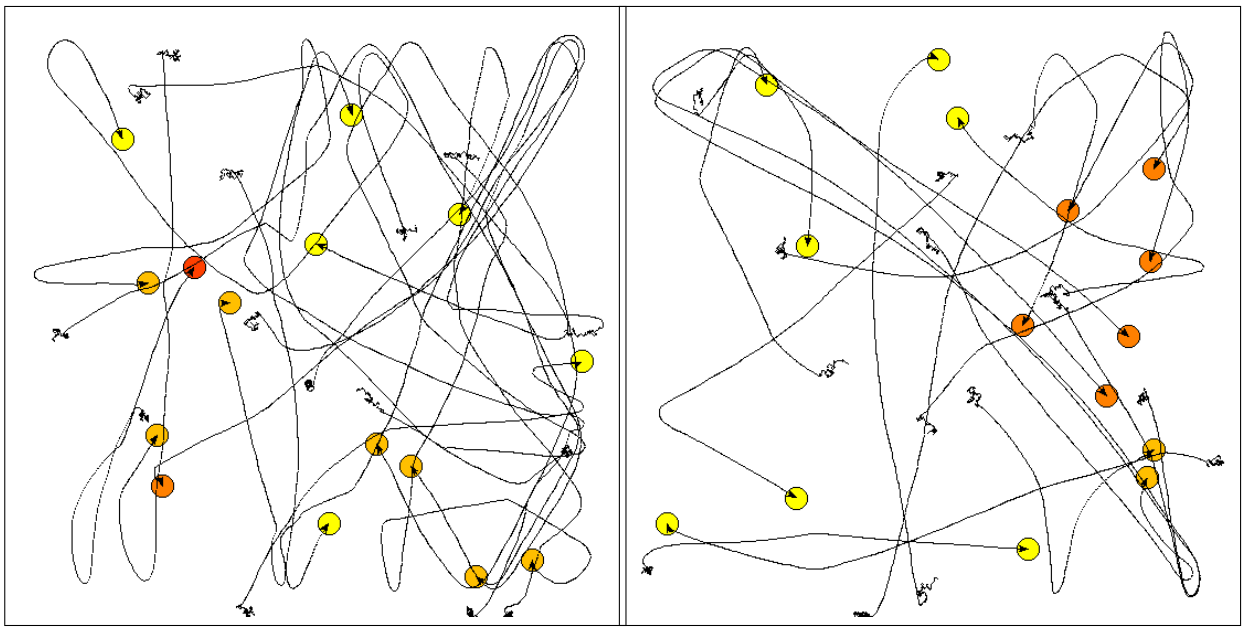

Figure 4.19: Simulation running the trained Prediction Model for ComplexModel
Trained network simulating agents for the ComplexModel In the left image (a) the predictions are made by sampling from the probability distribution. In the right image (b) the predictions are made by using the mean of the probability distribution.

4.4.2 T-SNE and UMAP

Looking at the T-SNE plots in figure 4.20 we see the distribution of labels in the projections. Compared to the SimpleCouzin and VeloModel the ZOR label appears more often alone. This

is due to the fact ZOR label can not appear at the same time as ZOO and ZOA labels in the movement model. ZOR and normal walk labels are isolated quite well in the T-SNE projection.

Looking at figure 4.21 we see that velocity, number of distinct neighbours and steps in which a neighbour was present can each separate high from low values.

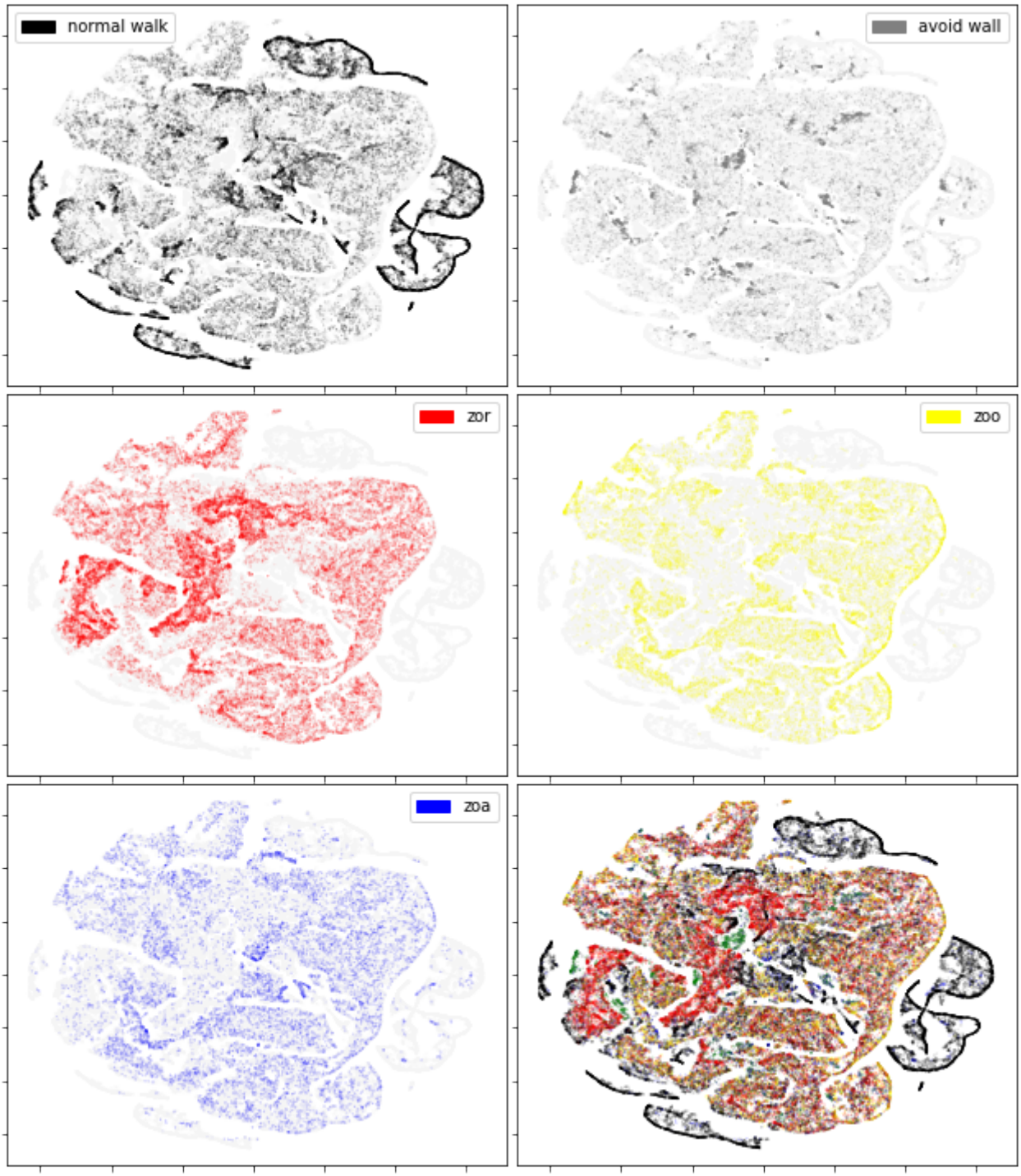

Figure 4.20: T-SNE Plots of the ComplexModel
Six plots with the same embedding. In the first five plots the base labels are colored individually. In the sixth plot the combinations according to figure 4.8 are colored in with their respective colors.

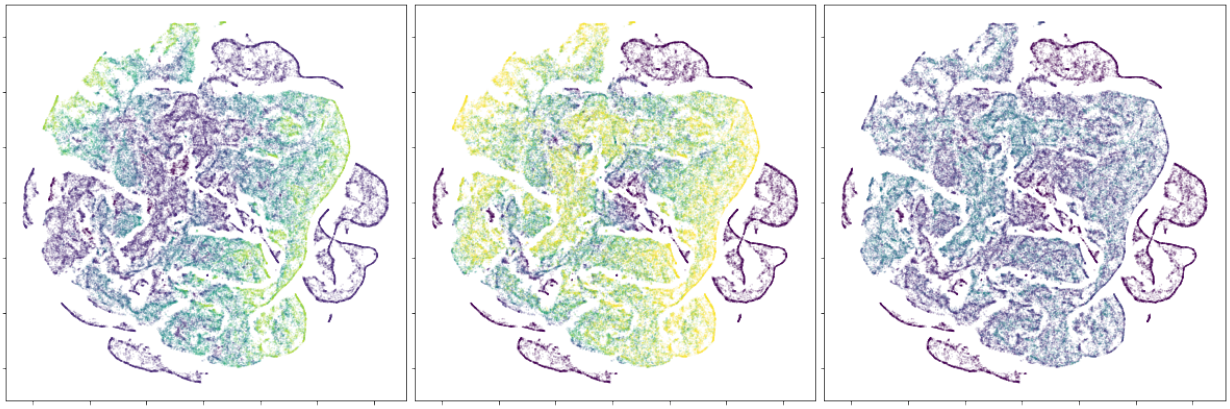

Figure 4.21: T-SNE plots for ComplexModel with velocity and neighbour labels.

We calculate the same UMAP projections as for the VeloModel. In figure 4.22 we see that the red ZOR labels form a ring in the middle, while normal walk labels are spread around the left side of the plot. The other labels are spread all around the ring.

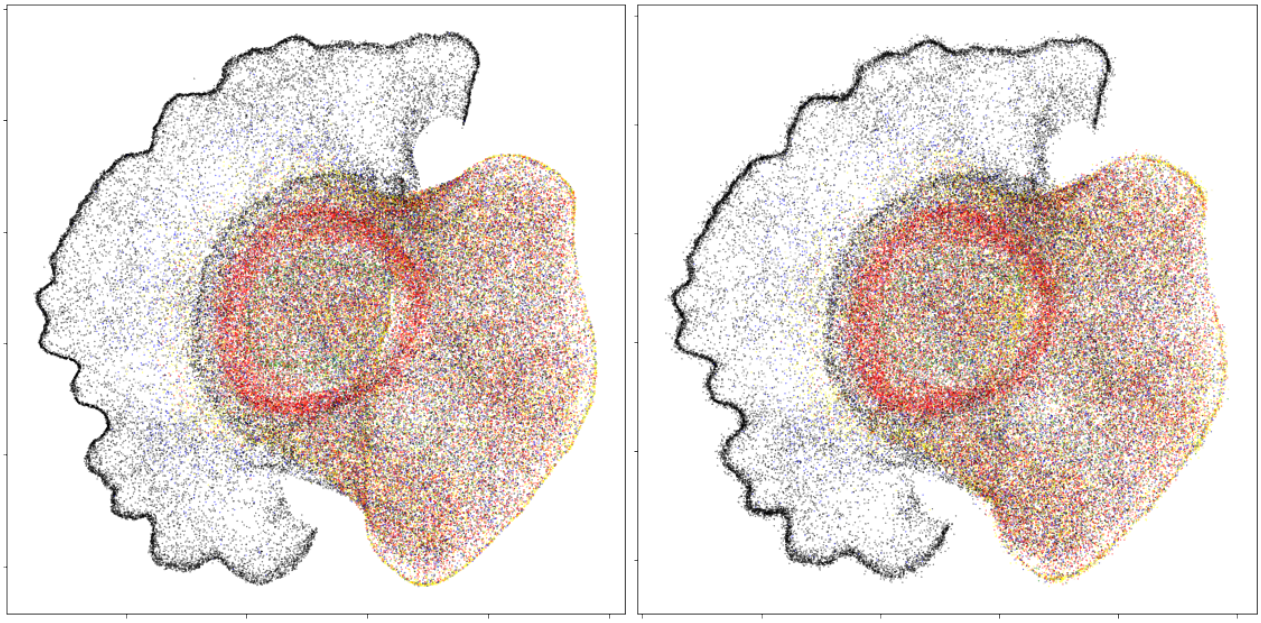

Figure 4.22: UMAP of Complex Model with zone labels
On the left the data which was used for training is projected and colored in, while on the right the test set was projected.

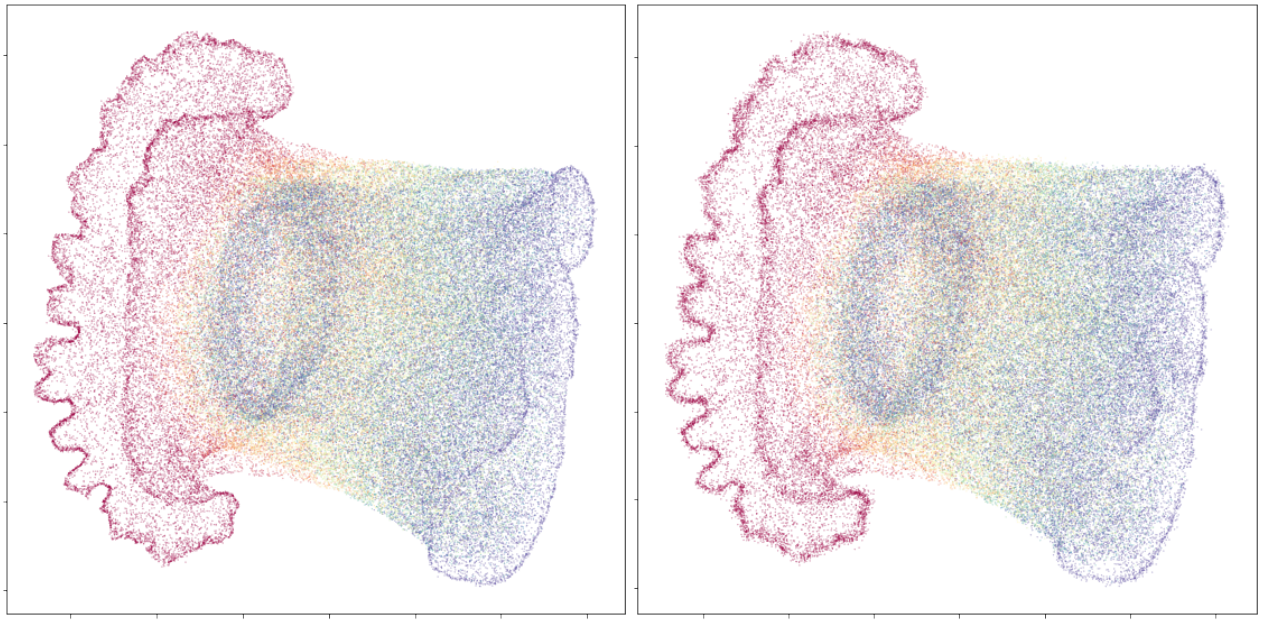

Figure 4.23: UMAP of ComplexModel with number of neighbour occurrences in last 40 steps.

4.5 Training of Bee Model

Finally we look at the training of the Prediction model with bee data. The errors are plotted in figure 4.24 and figure 4.25. The fluctuations of the error that already have been in the error plots of the ComplexModel increased.

4.5.1 Running the trained network

Running the bee network inside the simulation shows that trajectories look unreal. In the case of sampled trajectories there is a strong bias to go to the right. Bees walk into the borders, but simulation prohibits to go outside the borders. What can not be seen in the image is that the orientation constantly changes and the agents move very slow. In the trajectories that were calculated by getting the mean prediction the agents move even slower. They keep walking very slowly in the same direcction, and almost stand still, while the orientation changes in every step so that agents appear to wiggle.

4.5.2 T-SNE and UMAP

In figure 4.27 the projection of the hidden states with T-SNE is shown. The points are colored in according to the age of the bee that belonged to the window. There are no regions where single colors are dominant, instead its a very colorful mix everywhere. In figure 4.28 the same

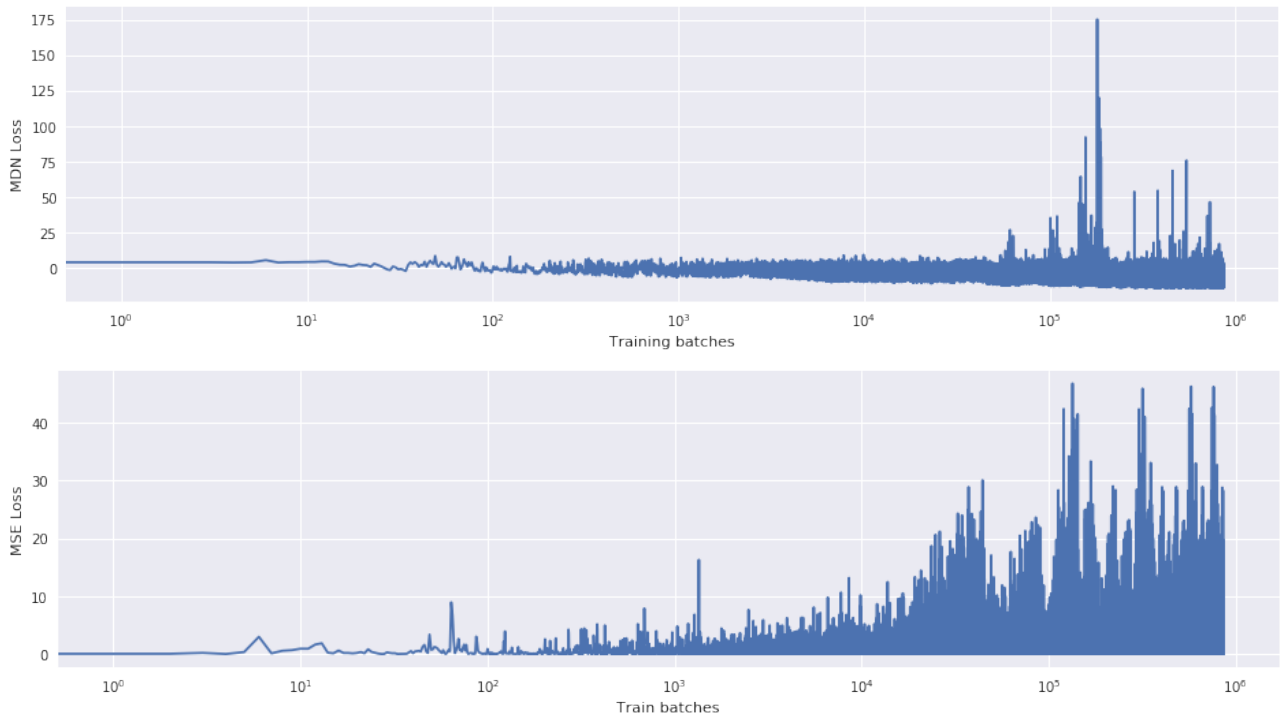

Figure 4.24: Training error while training the Prediction Model with bee data.

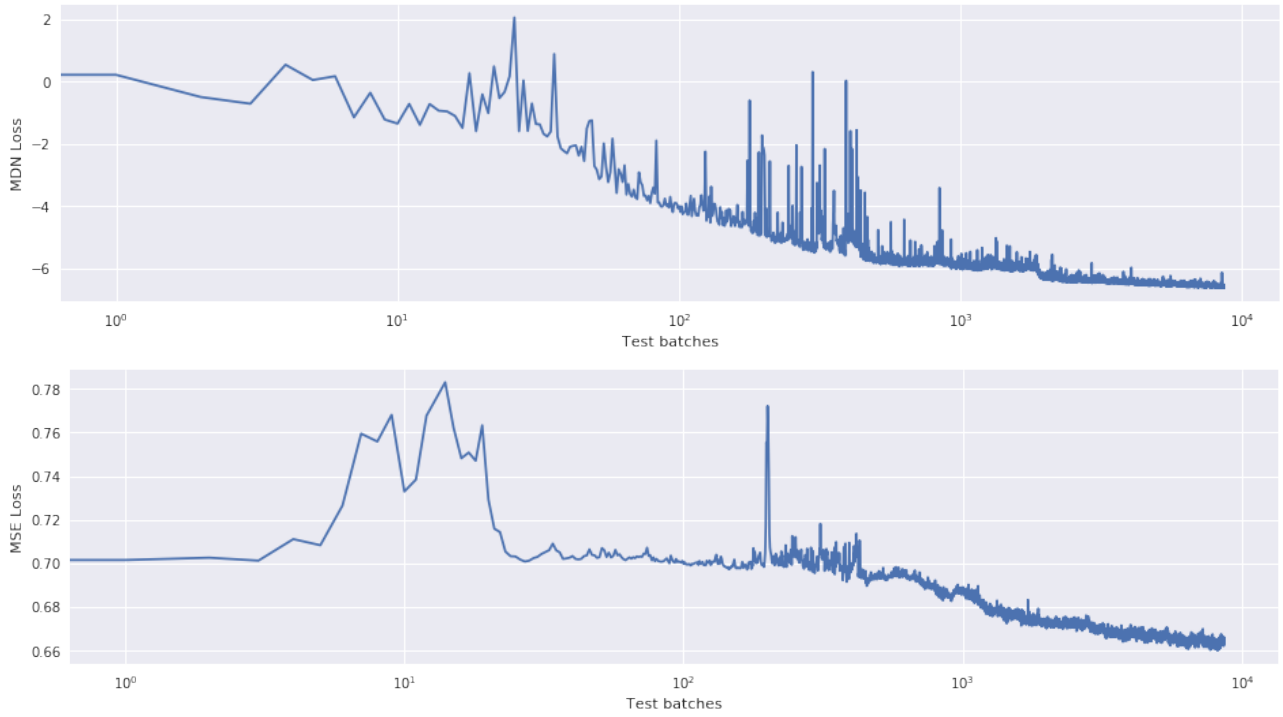

Figure 4.25: Testing error while training the Prediction Model with bee data.

projection is shown. This time the points are colored in according to the speed of the bee in the next timestep. It seems that resting bees can be separated quite well from slowly moving bees. In general most of the windows from the database are slowly moving or resting bees.

The UMAP projections in figure 4.29 do not reveal any additional information about the hidden

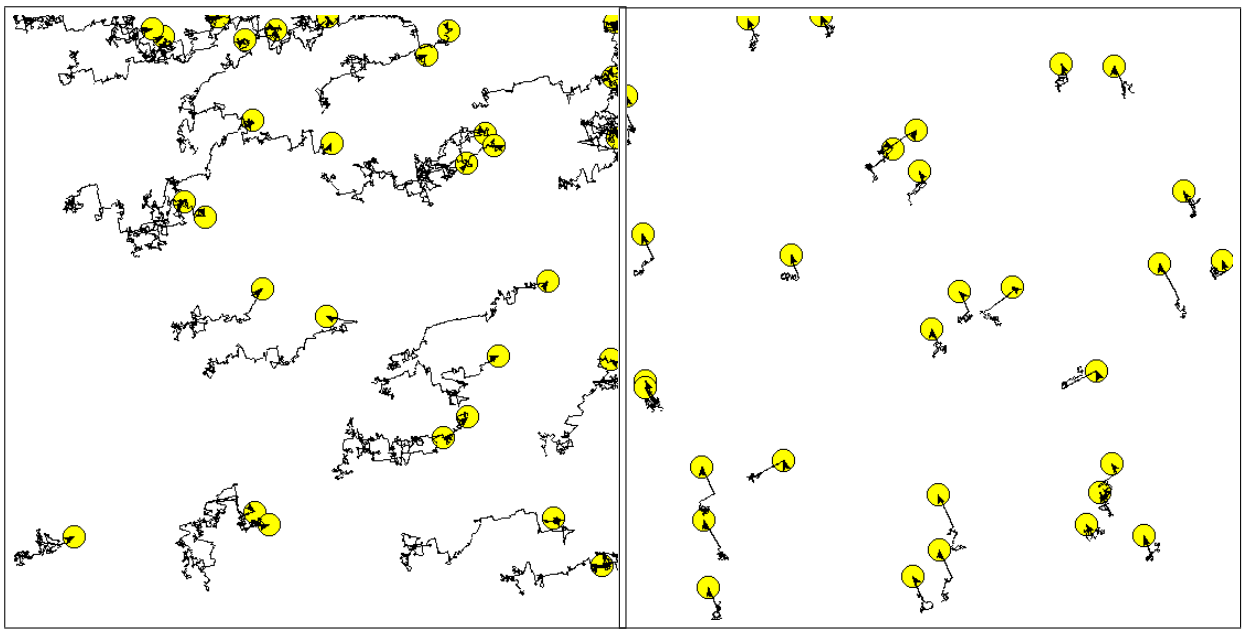

Figure 4.26: Simulation running the trained Prediction Model for BeeModel. On the left the trajectories were sampled from the GMM. On the right the mean of the GMM was used to predict trajectories.

states.

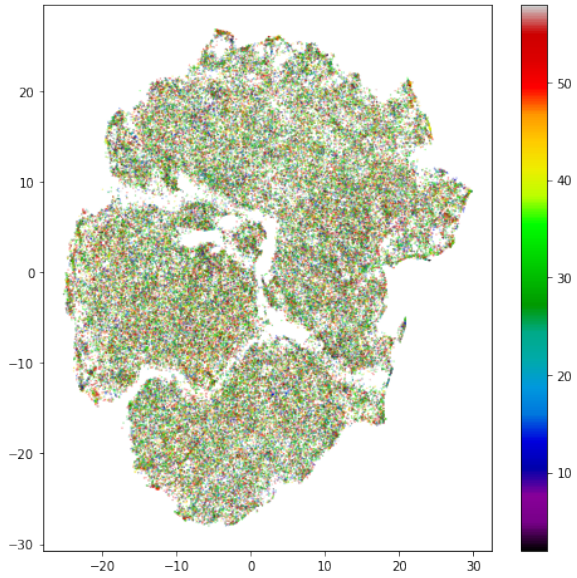

Figure 4.27: T-SNE Plot of the BeeModel with age labels
The T-SNE projection is colored according to the age of the bee. Next to the projection there is a color bar that shows how the colors are assigned to different ages.

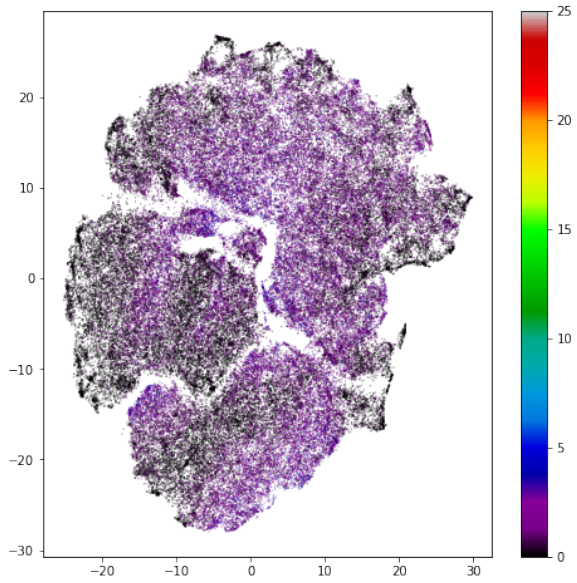

Figure 4.28: T-SNE Plot of the BeeModel with velocity labels
The T-SNE projection is colored according to the velocity of the bee. Next to the projection there is a color bar that shows how the colors are assigned to different velocities.

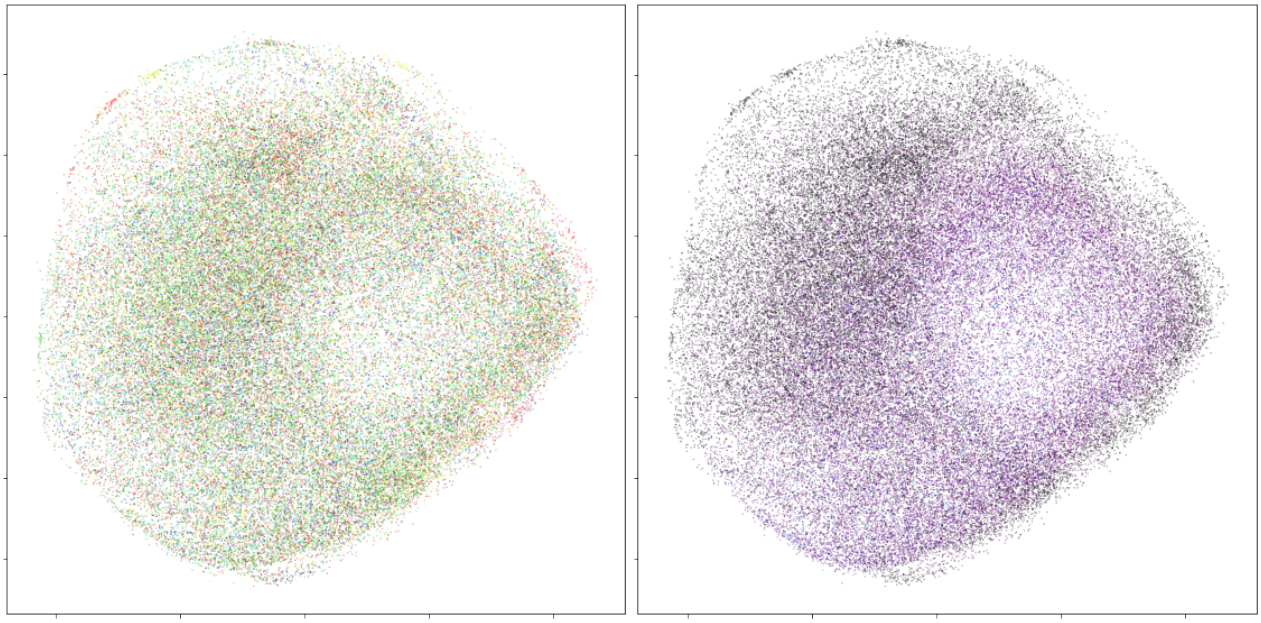

Figure 4.29: UMAP of BeeModel with age labels and velocity labels
On the right the projection that was trained with age labels is used. On the right the projection that was trained with the velocity labels is used. Both images show the projection of the test set.

4.6 Label Classification

In this section labels will be classified based on different classification models. The four models used are described in chapter 3.

Since the SimpleModel and the CouzinModel seem rather simple the tests will not be done with them. The BeeModel seemed to complex for our PredictionModel to fully learn it and the analysis of the hidden states revealed that no structure about the labels we have seems to be present. That is why in this section the focus is on the VeloModel and the ComplexModel.

One of the interesting labels that both models share is the number of timesteps where neighbours occurred in the last 40 timesteps.

Each of the four models is trained with different amounts of training data. Since this is an regression task where the each model needs to predict a single number we can plot an average error and an accuracy. The average error shows the average across the test set of how close the prediction is to the actual label. The accuracy shows how often the exact number was predicted.

First we look at the VeloModel. For training the same data was used as for T-SNE and UMAP. We will only show the performance of the trained model on the test set. Each model was run multiple times to compare the effect of having a small amount of training data and a lot of training data.

In figure 4.30 we see the average error for different amounts of training data used. The experiments were done with as little as 40 samples and up to 40000 samples. The x-axis at each point in the graph shows how many training samples have been used in that run. For each point the model was trained 10 times, and for each of those training runs new training data was sampled. This is to check how stable each model is and to get an average performance for this amount of data.

In the graph it is clear that the model that uses the hidden states from our previously trained Prediction model is by far the best, as long as no more than 40000 samples were used. A similar result can be seen in 4.31 with the accuracy. Using the hidden states the model achieves good result with very few data points. At 100 data points the average error already fell below 2, which means that the number of steps where an agent was seen can be estimated extremely good with a few labeled samples.

The other models improve a lot with more labels, while the model with the hidden states starts with a very good value and only improves slightly. The average error almost seems constant. The accuracy does have an upwards trend with more labels.

The same experiment was done for the ComplexModel. The results can be seen in figure 4.32 and in figure 4.33. Here the hidden states of the Prediction model did not seem to help a lot. The model that uses the hidden states managed to be only slightly better for the first few setups, but once the number of training samples reached around 250 the RNN model that uses the raw window input surpassed it and with more labels got better and better. The hidden states managed to stay on par with the Simple model that uses the raw data. The model that

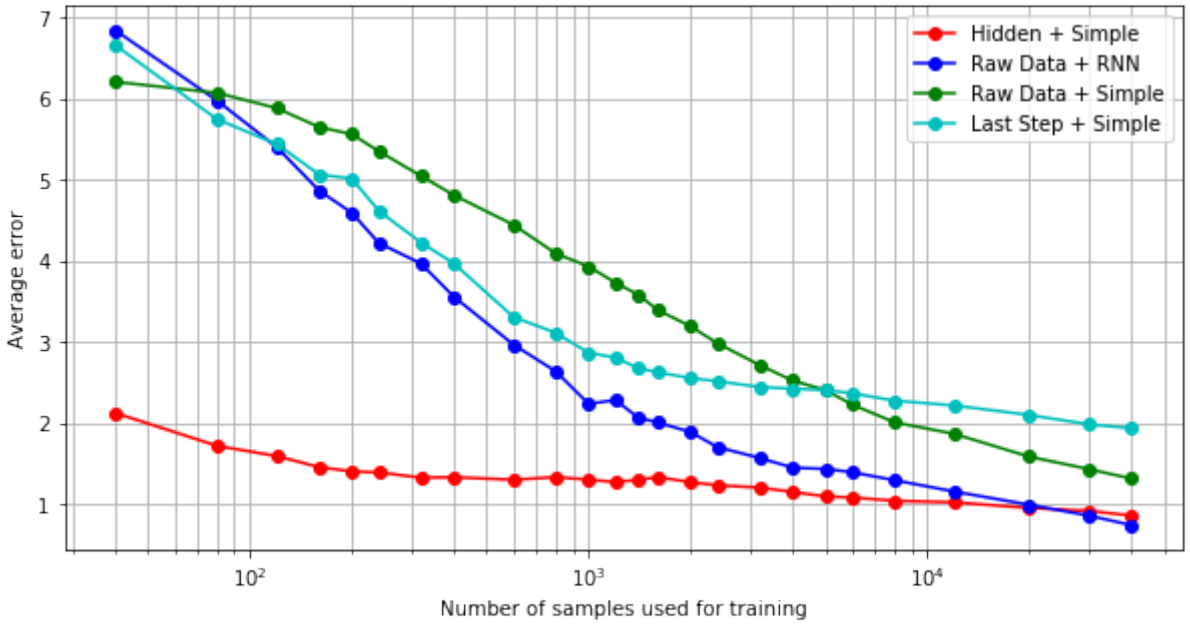

Figure 4.30: Average errors for runs with VeloModel data.
The red curve represents the simple network that is trained using the hidden states of the Prediction model. The blue curve represents the RNN trained on the raw window data. The green curve represents the simple network that is trained on the raw window data. The cyan curve represents the simple network that only uses the last window for classification. Each point on the curve represents a test run that is repeated 10 times. The average error displayed in the graph is an average of the average errors of each test run.

only used the last step for classification had the worst performance. This was expected, since the label needs all timesteps to be calculated accurately.

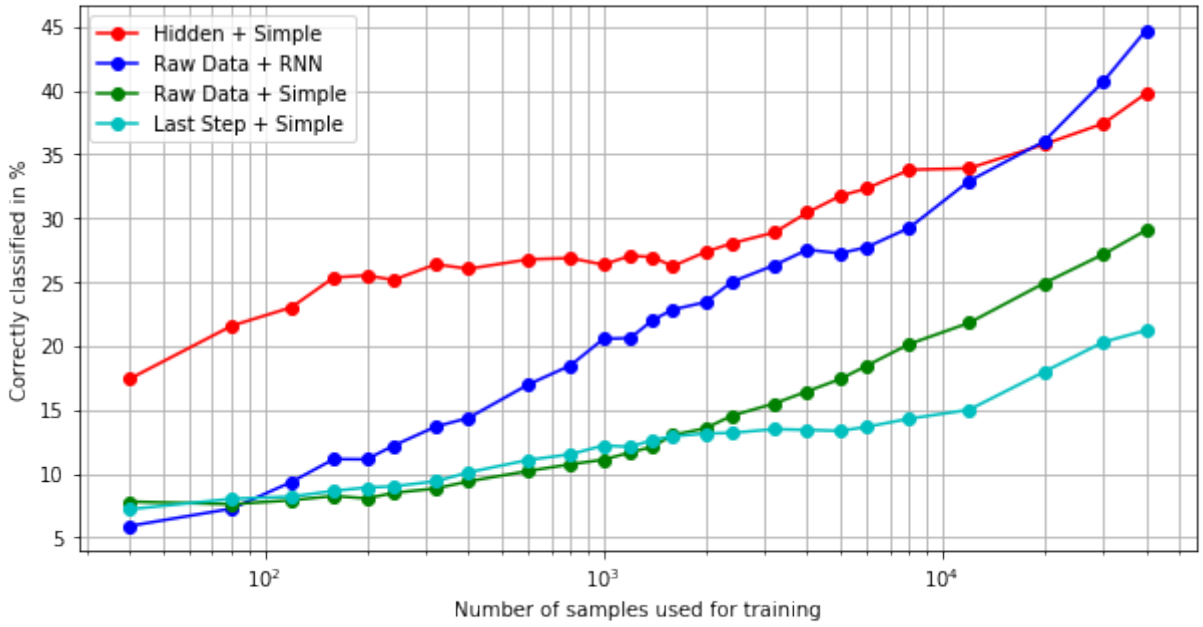

Figure 4.31: Accuracy for runs with VeloModel data.
The curves represent the same models as in figure 4.30. Each point on the curve represents a test run that is repeated 10 times. The accuracy displayed in the graph is an average of the accuracy of each test run.

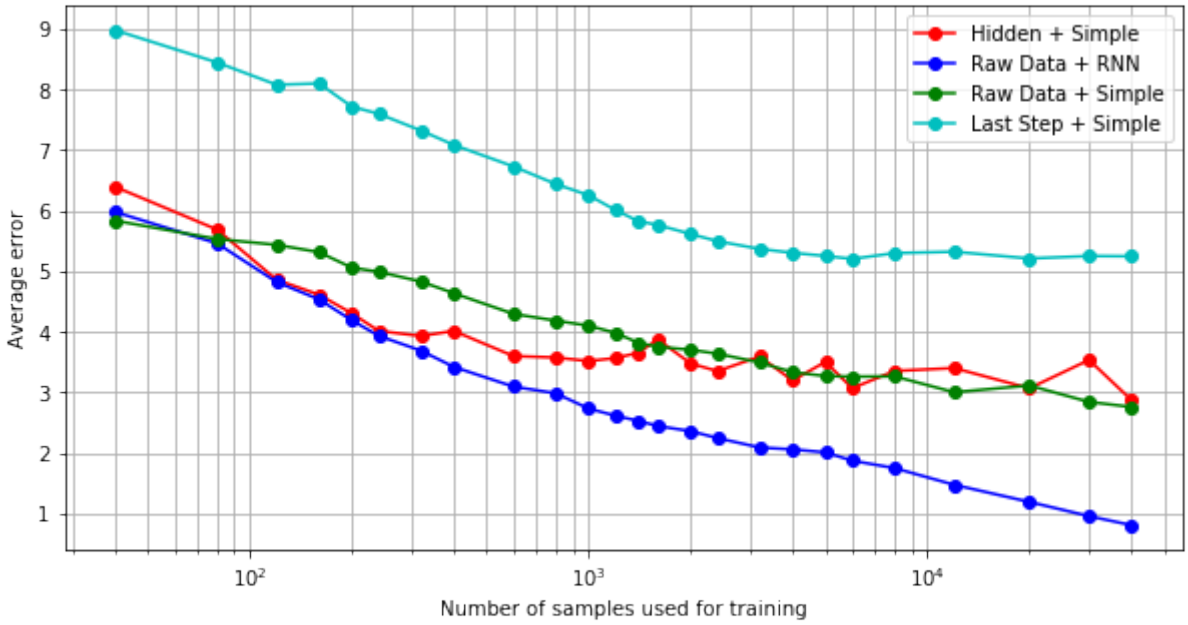

Figure 4.32: Average errors for runs with ComplexModel data.
Same experiment as in figure 4.30 with ComplexModel instead of VeloModel.

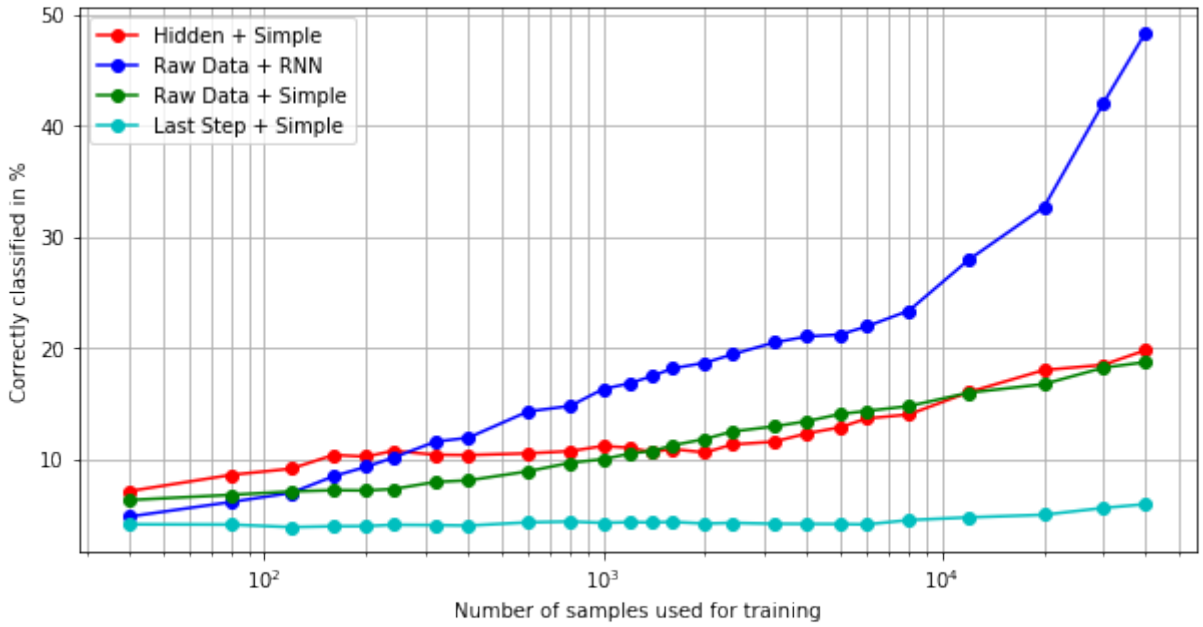

Figure 4.33: Accuracy for runs with ComplexModel data.
Same experiment as in figure 4.31 with ComplexModel instead of VeloModel.

Chapter 5

Discussion

In the last two chapters we described our objective, how we want to try to achieve it and what the results of our work were. In this chapter these results will be discussed.

5.1 Was the Simulation with all the movement models worth it?

So lets look at the simulation first. The simulation managed to generate data according to movement models that are not as complex as bee trajectories, but complex enough to see some interesting results. The visualization was informative enough to compare the implemented movement models to their trained Prediction network. Seeing the agents move revealed some of the problems the Prediction model had when generating trails, e.g. the bias.

For comparing bee trajectories with generated bee trajectories the simulation was not good enough. One of the reasons for this is that the simulation is not fast enough to simulate a whole bee hive with thousands of bees in real time. Another problem is that the simulation does not handle data with gaps in it very well. This leads to the question what should the simulation do if a bee disappears for a few seconds. Should the bee be removed or left in the place last seen. This can affect the neighbourhood of other bees around it and lead to unrealistic data. Since in this theses there never was the need to simulate huge amounts of bee data, these problems weren't answered. The simulation is able to play through single tracks from bee data, but can not nicely play through multiple bees at the same time.

The choice of the movement models was arbitrary. I needed movement models to test my model on and started with the simplest I could imagine. Remembering a course from university I came to the idea to reuse the Couzin paper and added complications to the Couzin model when I needed them. These models challenged my Prediction model to the extend that new

problems emerged, witch was the whole purpose of having the simulation. Therefore having these movement models was a success.

The simulation was a huge success in another way. Being able to test code is very important when experimenting with new data and new methods. This way development time can be improved significantly because problems are easier to see. Being able to play through single bee trajectories can already help to understand the nature of the data. Testing the implementation of different data models, to see if they preserve all the necessary information about trajectories. All this helps development time a lot.

5.2 Predicting trajectories

The goal of the Prediction model was to predict trajectories based on past trajectories. The training was done for all the models and worked for most of them.

The training of the SimpleModel can be considered a success. The trails produced by the Prediction model look very similar to those of the SimpleModel. Analyzing the hidden states shows that the two main states an agent can be in are separated almost perfectly. Although one drawback of ray casting showed up. While the movement model assumes a perfect vision and uses that, the data model can't catch every information in the neighbourhood of an agent. Thus the real model can react to a wall, while the ray casting data shows no sign of the wall. This can happen if the wall appears at the maximum visual range and is exactly between two rays and both don't reach the wall. Also there was a slight bias in the predicted trajectories. Where the bias comes from is not quite clear. My assumption is that it is a problem with the Prediction model itself. We optimize a probability distribution and have to optimize a loss function that uses probabilities. Since multiplying a lot of small probabilities can cause numeric problems for the computer, some parameters inside the loss function were set to not get smaller than a minimum number. This has the effect that the Prediction model sometimes can't reach perfect results and might be the reason that the small bias exists. Nonetheless the training with the SimpleModel yielded positive results.

The training of the SimpleCouzin model also worked quite well. The trajectories produced by the Prediction model looked very similar to those of the movement model. The key features of the SimpleCouzin model were learned. Agents avoided walls and started moving in the same direction as their neighbours. One problem was seen when running the simulation multiple times. Again a small bias was learned which caused agents to turn preferably to one side. Another problem was that agents seemed to react late sometimes. Agents that already saw another group of agents started to adjust their movement direction a few steps after the neighbours were seen. Using T-SNE and UMAP to separate the the label combinations displayed in figure 4.8 was still fairly good. A problem appeared with labels that are similar in nature and

often appeared at the same time. Those were harder to separate from one another.

The training of the VeloModel was also a success. Again the main features of the movement model were learned by the Prediction model and the generated trajectories looked very similar to the real ones. Adjusting the speed based on past events showed that the agent can learn time relevant behaviors. UMAP and T-SNE again yielded good results for the zone labels, but also for the newly introduces neighbour and velocity labels. Using the hidden states for classification boosted classification performance. This will be further discussed in the next section.

First problems started to appear at the training of the ComplexModel. While the training and testing error during the training phase went down continuously the Prediction model differed from the ComplexModel. Again the general features of the model, like avoiding walls, moving in small groups and accelerating when other agents are in neighbourhood were learned. But when it came down to slowing down only when enough distinct neighbours were seen, the Prediction model failed. It slowed down even when only a few distinct neighbours were seen. The ZOR did not work like in the original model. There might be multiple reasons for this. One explanation could be that the data model does not allow to differentiate between different agents. Every agent looks the same in the data model, but that is the data that the Prediction model gets in the input. Despite this disadvantage I am convinced that based on the vision more sophisticated models could predict better trajectories.

The last training was done with the BeeModel and again the training and testing errors went down. But for the first time the MSELoss did not go anywhere near 0. Comparing the generated bee trajectories with real bee trajectories is hard. T-SNE and UMAP only revealed that we can differentiate between moving and resting bees, which is trivial. There could be multiple reasons for this. The model might not be good enough to learn the complex movement of bees. Maybe the labels tested with T-SNE and UMAP can't be captured by 40 movement steps. Guessing the age of a bee based on a trail that was generated in a few seconds does sound like a hard task. Maybe the tracking data of bees has too many tracking errors and noise in it, which affects the training in a negative way. In my opinion it is a combination of those three things.

In summary the Prediction model worked to some extend on all the generated movement models. Only on the bee data it was not shown that the Prediction model learned anything meaningful.

5.3 Classification of labels

In this section the results from the Label classification in chapter 4 are discussed.

The results of the experiments with the VeloModel suggest that using the hidden states gen-

erated by the Prediction model can boost the classification performance. For small amounts of labeled data the classification based on the hidden states was significantly better than the performance of the other models.

But the results of the experiments with the ComplexModel do not show the same success. The simple model with the hidden states was at no point significantly better than the RNN that used raw window data. It could be argued that the Prediction model was not able to learn the important features of the ComplexModel and that is why the classification of labels didn't yield better results. It would seem that in order for an Prediction model to help with classification, the Prediction model itself has to be great at predicting trajectories that the movemen model would choose.

5.4 Problems and possible improvements

The simulation could be improved further. A real time visualization of larger amounts of agents and a better visualizations are definitely possible. The ray casting could be improved by giving agents and bees a "body" in the data model. Simulation could be run with more complex movement models for further testing that include more complex behavior.

The data model chosen as the input to the Prediction model only consist of positional data. A lot more could be added to it, like age of the bee, bee hive environment, temperature, size of the bees and interactions that can not be captured by positions alone, e.g trophallaxis.

Looking at the bee data from the database some problems still exist. There are inconsistencies in the database where bee ids are detected at the same time at multiple locations, detections of bee ids at times where the bee did not even hatch yet. Little tracking errors that make movement seem unnatural sometimes. The choice of the bee data used for training can also be improved, e.g. only using bees with a high activity, so the Prediction model doesn't have to bother with too many inactive trajectories. One could try to filter all the detections that seem to have errors in them. More labeled data is needed to evaluate the Prediction model further. Using age to test if the model works is ambitious.

The Prediction model could be improved. In the work of (Eyjolfsdottir et. al)[1] they used classification of labels and generation of sequences simultaneously. This could improve the quality of hidden states.

5.5 Conclusion

Predicting bee trajectories remains an open problem. Our results on the simulated movement models suggest that the chosen methods can work but still some problems need to be solved.

More labeled data for testing the bee model needs to be used, in order to evaluate the model properly. The tracking data needs to be analyzed further, to know what kind of tracking errors need to be considered. Further preprocessing on the bee data should be done, so that the quality of the trajectory data used for training is improved. Very inactive tracks can be excluded from training, since inactive tracks probably do not show interesting behavior anyways. Experiments with window sizes beyond what was tested in this work should be done. Additional features can be added to the bee model, e.g. hive entrance or honey comb texture. Further information about the bee itself, e.g. age and size. The simulation and the movement models could be used for further research, testing different Prediction models and other movement models. Adding more complex movement models could help with further testing.

5.6 Outlook

Since the prediction of bee trajectories is still an open problem more research should be done on it. But even if a Prediction model that works is found some interesting questions await us. Can such an model just be used to boost behavior classification, or could further applications be found. Is it possible to use such an model to gain insights about so far unknown behaviors or states of bees. Can reasons for certain behaviors be understood better when analyzing such a network. Can assumptions about bees be tested with a Prediction model that can simulate a bee hive. Finding answers to all these questions could improve the way bee research is done.

Bibliography

[1] Eyrun Eyjolfsdottir and Kristin Branson and Yisong Yue and Pietro Perona. Learning recurrent representations for hierarchical behavior modeling. 2016. arXiv:1611.00094 URL https://arxiv.org/abs/1611.00094

[2] Andrew M. Dai and Quoc V. Le. Semi-supervised Sequence Learning. 2015. arXiv:1611.00094 URL https://arxiv.org/abs/1511.01432v1

[3] Benjamin Wild and Leon Sixt and Tim Landgraf. Automatic localization and decoding of honeybee markers using deep convolutional neural networks 2018. arXiv:1802.04557 URL http://arxiv.org/abs/1802.04557

[4] Franziska Boenisch and Benjamin Rosemann and Benjamin Wild and Fernando Wario and David Dormagen and Tim Landgraf. Tracking all members of a honey bee colony over their lifetime. 2018. arXiv:1802.03192 URL https://arxiv.org/abs/1802.03192

[5] Fernando Wario, Benjamin Wild, Raúl Rojas, Tim Landgraf. Automatic detection and decoding of honey bee waggle dances. 2017. PLoS ONE. doi 10.1371/journal.pone.0188626. URL https://arxiv.org/abs/1708.06590

[6] Tom Burgert. Learning about Bee Behavior by Predicting the Future. 2018. https://www.mi.fu-berlin.de/inf/groups/ag-ki/Theses/Completed-theses/Bachelor-theses/2018/Burgert/Bachelor-Burgert.pdf

[7] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. 2008. Journal of Machine Learning Research. URL http://www.jmlr.org/papers/v9/vandermaaten08a.html

[8] Leland McInnes and John Healy and James Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. 2018. arXiv:1802.03426 URL https://arxiv.org/abs/1802.03426

[9] Leland McInnes and John Healy and Nathaniel Saul and Lukas Grossberger. UMAP: Uniform Manifold Approximation and Projection. 2018. The Journal of Open Source Software vol. 9, no. 29, pp. 861

[10] Sepp Hochreither and Jürgen Schmidhuber. Long Short-Term Memory. 1997. MIT Press. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735

[11] Christopher M. Bishop. Mixture density networks. 1994. URL http://publications.aston.ac.uk/id/eprint/373/1/NCRG_94_004.pdf

[12] Alex Graves. Generating Sequences With Recurrent Neural Networks. 2013. arXiv:1308.0850. URL https://arxiv.org/abs/1308.0850

[13] Couzin, I. D. and Krause, J. and James, R. and Ruxton, G. D. and Franks, N. R. Collective memory and spatial sorting in animal groups. 2002. Journal of Theoretical Biology, 218(1), 1-11. doi:10.1006/jtbi.2002.3065. URL https://pdfs.semanticscholar.org/0280/ea23d90ed2482ac6642011bbe8aa3e0ef9ad.pdf

[14] Songrit Maneewongvatana and David M. Mount. Analysis of approximate nearest neighbor searching with clustered point sets. 1999. CoRR cs.CG/9901013 https://dblp.org/rec/bib/journals/corr/cs-CG-9901013

[15] Eric Jones and Travis Oliphant and Pearu Peterson and others. SciPy: Open source scientific tools for Python. 2001. http://www.scipy.org/

[16] J. D. Hunter. Matplotlib: A 2D Graphics Environment. 2007. Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95

[17] Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam. Automatic differentiation in PyTorch. 2017. NIPS-W URL https://pytorch.org/

[18] Zelle, John. Python Programming: An Introduction to Computer Science 2nd Edition. 2010. Franklin, Beedle & Associates Inc.

[19] S. Lawrence, C. L. Giles, Ah Chung Tsoi and A. D. Back. Face recognition: a convolutional neural-network approach. In IEEE Transactions on Neural Networks, vol. 8, no. 1, pp. 98-113. 1997. doi: 10.1109/72.554195

[20] Yonghui Wu and Mike Schuster and Zhifeng Chen and Quoc V.Le and Mohammad Norouzi and Wolfgang Macherey and Maxim Krikun and Yuan Cao and Qin Gao and Klaus Macherey and Jeff Klinger and Apurva Shah and Melvin Johnson and Xiaobing Liu and Łukasz Kaiser and Stephan Gouws and Yoshikiyo Kato and Taku Kudo and Hideto Kazawa and Keith Stevens and George Kurian and Nishant Patil and Wei Wang and Cliff Young and Jason Smith and Jason Riesa and Alex Rudnick and Oriol Vinyals and Greg Corrado and Macduff Hughes and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. 2016. arXiv e-prints. arXiv:1609.08144, URL https://arxiv.org/abs/1609.08144

[21] Mariusz Bojarski and Davide Del Testa and Daniel Dworakowski and Bernhard Firner and Beat Flepp and Prasoon Goyal and Lawrence D. Jackel and Mathew Monfort and Urs Muller and Jiakai Zhang and Xin Zhang and Jake Zhao and Karol Zieba. End to End Learning for Self-Driving Cars. 2016. arXiv e-prints. arXiv:1604.07316 URL `https://arxiv.org/abs/1604.07316v1`

[22] BioroboticsLab of Freie Universität Berlin. Blog of the BioroboticsLab. URL `http://berlinbiorobotics.blog/`

List of Figures