Learning structured exploration strategies through learned neuromodulated plasticity rules

Christos Georgiades

University of Cyprus

Department of Computer Science

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Bachelor of Science

at the

University of Cyprus

APPROVAL PAGE

Bachelor of Science Thesis

LEARNING STRUCTURED EXPLORATION STRATEGIES THROUGH LEARNED NEUROMODULATED PLASTICITY RULES

Presented by

Christos Georgiades

Research Supervisor ______________________________
Professor Christodoulou Christos

Committee Member ______________________________
Dr. Vassiliades Vassilis

University of Cyprus

June, 2021

ABSTRACT

If the end goal of the field of Artificial Neural Networks is to create an artificial mind then it should be able to explore an environment, an innate behaviour in almost all living beings. This leads to the project's main question "Can structured exploration behavior be encoded in an artificial neural network (ANN)?". Using embedded neuromodulated plasticity rules and imitation learning methodology we want to find out how an ANN trained this way learns, adapts and performs and how it compares against a long short-term memory network which is a state-of-the-art model that can be used for sequence modeling. We create non-stationary environments with a changing reward function, T-mazes, where our human participants will generate optimal/structured exploration behavior as they explore. Using those traces we train both of the networks and attempt to reproduce similar traces. We conclude that neuromodulated plasticity rules and imitation learning can provide a powerful solution to exploration problems and can outperform other conventional models.

ACKNOWLEDGMENTS

The completion of this study would not have been possible without my supervisor, Professor Christos Christodoulou, for his continuous support and guidance, along with allowing me to research and inquire deeper into my fields of interest.

I want to extend my gratitude to my co-supervisor Dr. Vassilis Vassiliades, who generously provided me with his knowledge and experience throughout the undertaking of this project, as well as the countless hours he set aside to assist me.

I want to thank my dear mother and father, who provided me with their wisdom and moral support during the whole of my academic career. I sincerely appreciate their patience and the fact they stood by me every step of the way.

Last, but not least, I want to to dedicate this to my brother, Demetris, who taught me that few things matter in life as long as you are happy.

Table of Contents

Table of Figures

Index of Tables

Chapter 1

1.1 Introduction

Exploration is a challenging problem for all fields of Neural Networks and in this project we will attempt to solve it from the aspect of using recorded human actions to teach Neural Networks. The question that underlies the topic of this thesis is whether structured exploration behaviour can be taught and encoded in an Artificial Neural Network (ANN).

The objective of the project is to see if it is possible and viable to train an ANN using human traces and to study how the performance of such a network compares in performance to other conventionally trained networks. Via collecting traces of human agents trying to navigate through non-stationary environments (T-mazes) and collect the reward and providing those traces to an ANN. We will use a differentiable plasticity model along with an extension of that model, backpropamine, which is a biologically inspired model that incorporates neuromodulated plasticity in our ANN. We want to find out how an ANN trained this way learns, adapts and performs and how it compares against a long short-term memory [1] network, which is a state-of-the-art model that can be used for sequence modeling.

By proving that Neural Networks can be taught to solve simple exploration tasks in controlled yet randomized envrironments we can extend our finding and pave the way to more advanced exploration problems of unknown topography.

1.2 Related Work

The problem is based on animal cognition experiments and behavioral science, where rats are tested for their ability to learn and memorise paths in physical mazes. Exploring T-Mazes has been studied from the scope of Neural Networks before albeit with different forms of NNs. Using Continuous-time recurrent neural networks [2] and a different training method to the one proposed here, it has been managed to train a robot to traverse Single depth T-Mazes successfully but had questionable results with Double T-Mazes. The study did not investigate larger depth T-Mazes and homing tasks. Another example of a study using T-mazes is the work of [3] which focused on how Neuromodulated neurons were able to achieve better learning and memorisation in comparison to standard neurons. Furthermore, Vassiliades and Christodoulou (2016) [4] introduced the novel computational model of switch neurons that were capable of switching between behaviors in response to changes in the environment. These neurons were trained in T-mazes very similar to the ones we are using for this dissertation

1.2 Thesis outline

The thesis consists of five chapters. In the first chapter we explain the topic of the thesis and mention related work. The second chapter fleshes out the background surrounding the field and the topic that helps grasp the perspective of this work. The third chapter includes details of the methodology that was followed and the implementation of the work. The results and their analysis are discussed on the fourth chapter. The last chapter is a summary of the conclusions and possible future work, around this thesis.

Chapter 2

Epistimological Background

2.1 Exploration in Neural Networks

Exploration is a problem the field of Neural Networks and Reinforcement Learning has yet to give a conclusive solution to with various methods and models proposed throughout the years attempting to solve this challenge. From Eplsilon-Greedy[5] which attempts to balance exploitation, using known data and repeat actions that have worked well, and exploration, making novel decisions in attempt to achieve greater rewards, to Deep-Reinforcement Learning (DRL) [6]. Proposed solutions suffer from various shortcomings either from sparse or deceptive rewards, known as a Hard Exploration Problem or from various Noisy-Tvs [7] where in the presence of uncontrollable & unpredictable random noise the agent effectively becomes a “couch potato” as it obtains new rewards consistently, failing to make any meaningful progress. Using human traces we hope to develop a method of teaching NNs to explore known and unknown environments that will circumnvent such problems.

2.1.1 T-Mazes

T-Mazes are environments based on behavioral science and used in animal cognition experiments [8]. T-mazes are used to study how the rodents function with memory and spatial learning through applying various stimuli. With the insurgence of Neural Networks the T-Mazes were naturally adapted to be used in navigation and memorisation experiments. A T-Maze is made up of a T-Junction where the agent, physical or not, has to navigate and make a choice of either turning left or right to find the reward places at the end of the two perpendicular corridors.

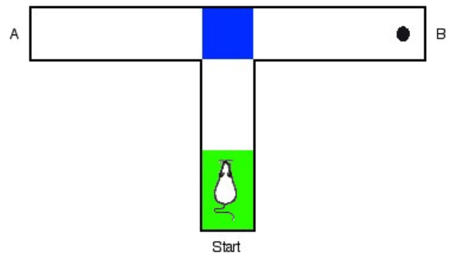


*Figure 1: T-Maze layout. Single depth. Coloured are the start **(green)** and a turn **(blue)**.*

2.2 Differentiable Plasticity

The main ideas of how our ANN learns and works are based on the work of Miconi et al. (2018)[9] on Differentiable Plasticity where we have a system of Nodes which are trained with backpropagation which consists of a nonlinear forward pass and a linear backward pass and are able to change the inner values of the network. Following this approach it is possible to calculate parameters for the rules via stochastic gradient descent and effectively our Neural Network is able to learn.

$$x_j(t)=\sigma\{\sum_{i \in inputs}[w_{i,j}x_i(t-1)+\alpha_{i,j}Hebb_{i,j}(t)x_i(t-1)]\}. \tag{1}$$

In differentiable plasticity a connection between any two neurons i and j, has both a fixed component and a plastic component. The fixed part is a weight $w_{\mathrm{i,j}}$ while the plastic part is stored in a Hebbian trace $Hebb_{\mathrm{i,j}}$. The “plasticity” of a connection can vary from fully fixed to fully plastic based on the Plasticity coefficient $\alpha_{\mathrm{i,j}}$ (Eq. 1).

$$Hebb_{i,j}(t+1)=\eta x_i(t-1)x_j(t)+(1-\eta)Hebb_{i,j}(t). \tag{2}$$

The Hebbian Trace (Eq. 2) or the Hebbian Rule is responsible for providing simultaneity to our ANN stemming from biological neurons. If interconnected neurons become active very close in time during a particular event, their connection strengthens and “a memory” of this event is formed, in simpler terms “neurons that fire together, wire together”.

In Hebbian rules there is often a weight decay term η (Eq. 3), to prevent explosive positive feedback on Hebbian traces. However, in the lack of input Hebbian traces often decay to zero. More complex Hebbian rules can maintain values indefinitely in the absence of stimulation, thus allowing stable long-term memories, while still preventing runaway divergences (Oja, 2008).

$$Hebb_{i,j}(t+1)=Hebb_{i,j}(t)+\eta x_j(t)(x_i(t-1)-x_j(t)Hebb_{i,j}(t)). \tag{3}$$

Using embedded plasticity rules in the Neural Network architecture it is possible to calculate the values of the learning rules via stochastic gradient descent and let a NN learn on its own and we expect it to be a powerful tool when learning from demonstration.

σ :	Non-linear function, usually tanh.
$w_{\mathrm{i,j}}$:	Weight between neurons i, j. Non-plastic compoment connecting neurons i,j.
x :	Neuron output
$\alpha_{\mathrm{i,j}}$:	Plasticity coefficient
$Hebb_{\mathrm{i,j}}$:	Hebbian trace, plastic compoment connecting neurons i,j.
η :	Weight decay value

2.3 Backpropamine

Backpropamine is an expansion of differential plasticity, showed that neuromodulated networks can be trained with gradient descent. Using a mechanism called eligibility trace, which keeps track of which synapses contributed to recent activity, a dopamine signal controls how the eligibility traces are transformed into plasticity changes.

Replacing Eq. 3 above with the following two equations,

$$Hebb_{i,j}(t+1)=Clip(Hebb_{i,j}(t)+M(t)E_{i,j}(t)). \tag{4}$$

$$E_{i,j}(t+1)=(1-\eta)E_{i,j}(t)+\eta x_i(t-1)x_j(t). \tag{5}$$

The function $Clip(x)$ in Eq. 5 is any function or procedure that constrains $Hebb_{\mathrm{i,j}}$ to the$[-1,1]$ range and replaces Oja's rule that was used in Differentiable Plasticity.

Here $E_{\mathrm{i,j}}(t)$ is a simple exponential average of the Hebbian product of pre- and post-synaptic activity, with trainable decay factor η . $Hebb_{\mathrm{i,j}}(t)$, the actual plastic component of the connection (Eq. 4), simply accumulates this trace, but gated by the current value of the dopamine signal $M(t)$. Note that $M(t)$ can be positive or negative, approximating the effects of both rises and dips in the baseline dopamine levels [11]. The calculation of the dopamine signal can be done in various ways. Our implementation uses a sclara output of the network that is then passed through a meta-learned vector of weights.

$E_{i,j}(t)$	:	Eligibility trace at connection i, j.
$M(t)$	:	Dopamine signal.

2.4 LSTM

A common Long Short-Term Memory [1] model works tremendously well on a large variety of problems, and is now widely used. An LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. Relative insensitivity to gap length is an advantage of LSTM over RNNs which suffer from vanishing and exploding gradient problems.

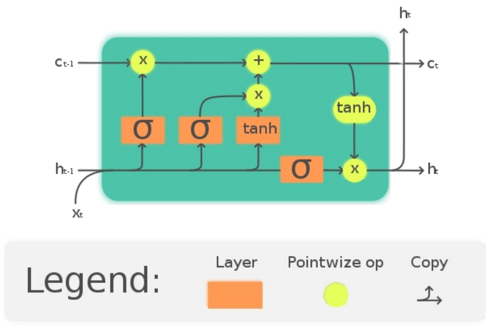


Figure 2: LSTM model diagram
Author: Guillaume Chevalier

2.5 Adam Optimization

The Adam optimization algorithm [12] is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing. It is an algorithm that can be used instead of the classical stochastic gradient descent to update network weights based on training data. It is appropriate for problems with non-stationary objectives and has little memory requirements

The Adam algorithm adapts the parameter learning rates based on the average first moment (the mean). Adam also makes use of the average of the second moments of the gradients (the uncentered variance). Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.

2.6 One-Hot Encoding

In machine learning, One-Hot Encoding [13] is a frequently used method to deal with categorical data. Because many times, machine learning models need their input variables to be numeric, categorical variables need to be transformed. An integer encoded variable is removed and a new binary variable is added for each unique integer value. Thus for x different categories, x different binary variables are needed. At each moment only one binary variable is activated and its value set to 1, while all other variables are set to 0

2.7 Class Imbalance

An imbalanced classification problem is an example of a classification problem where the distribution of examples across the known classes is biased or skewed. The distribution can vary from a slight bias to a severe imbalance where there is one example in the minority class for hundreds, thousands, or millions of examples in the majority class or classes.

Imbalanced classifications pose a challenge for predictive modeling as most of the machine learning algorithms used for classification were designed around the assumption of an equal number of examples for each class. This results in models that have poor predictive performance, specifically for the minority class. This is a problem because typically, the minority class is more important and therefore the problem is more sensitive to classification errors for the minority class than the majority class.

Our project is a classical example of an imbalanced classification problem. Since our T-Mazes naturally contain more straight corridors than turns. While corridors are undemanding in their traversal, the more rare turns presented in the environment are complicated, requiring the agent to orient themselves and memorise which paths they visited in the past.

Chapter 3
Design and Implementation

3.1 Training Methodology

The training of the ANN was split into two phases, first and foremost was generating the human traces of optimal/structured exploration behavior in non-stationary environments, where the reward function changes. Within the margins of this project we used randomized T-mazes of varying depth. In the second phase we used imitation learning-like methodology to train the backpropamine ANN and attempt to reproduce similar traces as well as the LSTM which stands as a comparison point for our ANN.

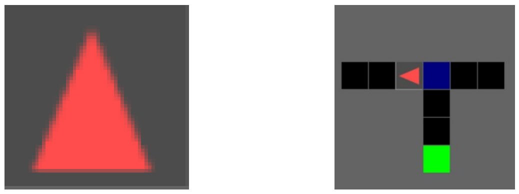
3.2 Agent Environment

The agent environment is parameterized and randomly generated and similar to the specifications other research used [4], specifically the agent's visibility was restricted. The agent's actions are "move forward", "turn left" or "turn right", and the agent was allowed to change orientation only when choosing a turn action at a turning point, or reverses automatically in the homing task when reaching the end of a corridor. Each action was a single step in the Maze and in the trace and invalid actions were not recorded and did not increment the step counter in order to reduce junk data fed into the NNs.

3.2.1 Agent Visibility

The visibility of the agent was an integral part of the project, as it determined what data was stored and encoded into the trace which was to be used to train our ANN. As part of the experiments we allowed for a flexible impementation that makes it possible to restrict the human observation to:

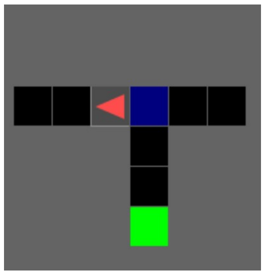
- **Semi-full observability:**
  - the full maze with the rewards obscured
- **Partial observability:**
  - limited agent observation

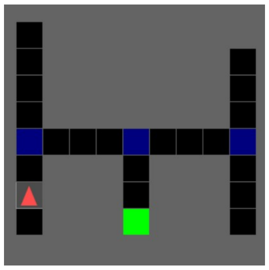

*Drawing 1: Partial Observability**(left)**. Semi-full observability**(right)***

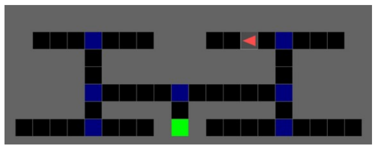
Later on Partial Observability in human experiments was scrapped since it proved to be a difficult task for humans to solve the mazes using only the information of a single cell and that was reflected in the quality of the traces collected.

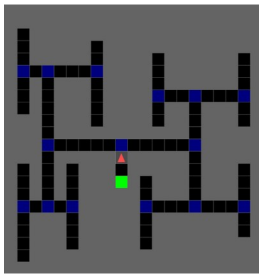
3.3 Maze Design

For the construction of the mazes we used the Gym (https://gym.openai.com) and Minigrid (https://github.com/maximecb/gym-minigrid) frameworks since they provided the necessary tools to build our virtual rat maze as well as simple and intuitive ways that allowed us to interact with our agent and the outside world. The level design consists of T-Mazes, which are corridors connected in a way that forms a T-junction. The depth of a maze determines how many choices an agent must make in order to reach an end point of the maze. Furthermore, tasks were divided to non-homing and homing, where the agent has to return to their original starting position after collecting the reward. This is designed to test the agent's ability to memorise its path and retract its steps.


Drawing 2: Single Depth T-Maze


Drawing 3: Double Depth T-Maze


Drawing 5: Triple Depth T-Maze


Drawing 4: Quadruple Depth T-Maze

Each tile in the maze was colour coded to represent its function and the available movements. The Start tile was green, and in the case of homing tasks, when the agent picked up the reward and had to return to its initial position, the tile would change to red. T-Junctions, where an agent would turn left or right, were coloured blue while corridors were black and walls were grey.

The different tasks appeared in the following order:

Table 1: T-Maze task order

1. Single, non-homing, partial
2. Double, non-homing, partial
3. Triple, non-homing, partial
4. Quadruple, non-homing, partial
5. Single, non-homing, semi-full
6. Double, non-homing, semi-full
7. Triple, non-homing, semi-full
8. Quadruple, non-homing, semi-full
9. Single, homing, semi-full
10. Double, homing, semi-full
11. Triple, homing, semi-full
12. Quadruple, homing, semi-full

- Tasks 1-4
    - will test the human ability to solve the task when presented with input similar to the one the agent observes.
- Tasks 5-8
    - provide more information to the user - the traces from these tasks are expected to have better overall performance.
- Tasks 9-12
    - are about the homing task - the user will be provided with more information and would determine if an agent learns to remember (by reversing its path).

Each task was repeated a number of times, with the same reward location, to allow the agent not only to explore many ends of the maze but also exhibit memorisation after finding the reward. Depending on the depth of the maze, each task was repeated for a number of episodes, with Single T-Mazes repeating for 4-8 episodes. Double for 8-12 episodes, Triple for 16-20 episodes and quadruple for 32-36 episodes. This means that for all experiments the human participants were asked to provide a trace containing 12 different tasks which contain 180 to 228 episodes.

The reward locations were randomized and selected from one of the possible corridor ends of the maze. For each task the reward location changed once after a random number of episodes to stipulate exploration and encourage the agent to revisit the Maze's ends. Care was taken to allow the agent to explore for at least two to three episodes for each reward location.

For each action taken, points were awarded to the agent. A step penalty was applied for every step to avoid unnecessary moves and encourage optimal behaviour strategy. An additional penalty was applied when the agent failed to find the reward in the maze or, in the case of homing tasks, the agent does nor manage to successfully return to the start of the maze.

In the case of a Single-Depth Non-Homing T-Maze the agent begins from the start of the T junction and the rewards are found at either end of the two perpendicular corridors. The agent's task was to nagivate down the corridor, turn either left or right, and reach the end of the corridor, where it found the reward or in the absence of it, turned around to explore the rest of the maze. The imposed maximum number of steps allowed the agent only to explore only one end of the maze.

3.3.1 Maze Generation

The generation of the T-Mazes occured in multiple steps. First and foremost, the interface *[Appendix A]* invokes the corresponding task and passes along the seed, and if needed, the location of the reward in the form of an integer which determined which maze end held the reward of the maze. Depending on the depth of maze, the number of corridors of the T-Maze and the maximum number of steps the agent was allowed were determined. In the case of a homing task the amount of steps were doubled.

Subsequently, the lengths of the corridors were generated in pseudorandom fashion, the corridors were then checked for overlap to avoid circular paths. Then, based on the generated corridor lengths the square size of the maze is calculated. In the last step of validity checking, we calculate the negative offset of the grid. This offset is used for Triple and Quadruple depth mazes which might extend past the initial starting position of the agent.

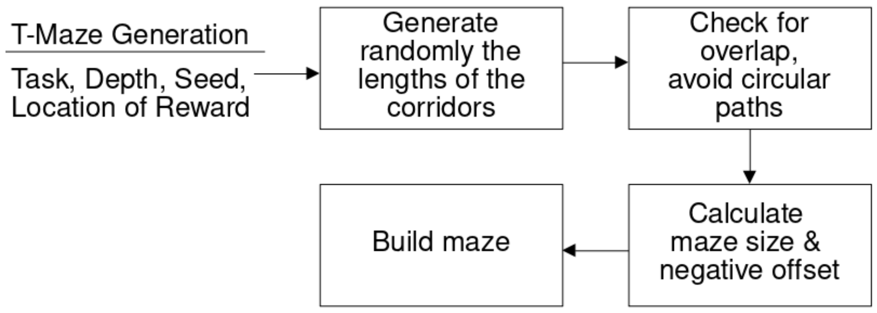

T-Maze Generation
Task, Depth, Seed, Location of Reward
Generate randomly the lengths of the corridors
Check for overlap, avoid circular paths
Calculate maze size & negative offset
Build maze

Figure 3: Maze generation process

Afterwards the corridor lengths are generated and checked. The maze is build from the ground up. The square grid is created and all the tiles are set to be blocked. Afterwards using a recursive builder we iterate through the corridor lengths list and place alternating horizontal and vertical corridors in the grid. The secondary role of this recursive builder is to return the coordinates of all the maze ends and all the turning points in the maze. Using the two lists of maze ends and turning points we place all the coloured tiles at the appropriate coordinates and enable the movements allowed at those locations. i.e. the automatic reversal of direction, and allowing the agent to turn at the turning points. Lastly the starting tile, the reward and the agent are placed in the maze.

3.4 Collecting Traces

To train our Neural Networks we needed to gather data from human participants. The mazes were distributed as a zip file with the required python code, and used bash or shell scripting, depending on the host ecosystem, to handle installing any additional plugins and running the maze experiments. Human agents were presented with the mazes discussed above and were instructed to solve the mazes and find the reward as efficiently as possible. The human agents were given as much guidance as possible without exposing the underlying systems of the maze to assist them in creating efficient and accurate traces.

The overall architecture of the human maze environment, was comprised of a handler that was the entry point of the system and responsible for the configuration of the mazes and the number episodes which were later passed on to an interface. The interface was responsible for the creation of the actual maze and coo diff_plasticity.main(0.01, 0.001, 20, epochs=epochs, save='dp')rdinating human inputs and the environment and creating traces with the outputs from the maze.

A trace file was comprised of multiple tasks and episodes. When a new episode started the type of the maze along with its seed was recorded in the trace file in the two seperate lines. Following that was the recording of its step, comprised of the observation of the agent, which indicated the type of tile the agent was located on, the human action, and the points awarded by the maze for the action. After an episode was finished the keyword “done” was recorded in the trace file, to signify the correct termination of the episode. Additionally the same was done when all tasks were finished to signify the end of the trace file.

3.4.1 Trace File Format

A single step in the trace file was recorded with the following format;

$$[obs_{agent}]\,action_{human}\,reward \tag{7}$$

The agent obeservation was comprised of a One Hot Encoding comprised of five bits of the possible different tile types the agent can be occupying. At each point only one of the five bits was enabled and set to 1 and the rest were set to 0.

$$[obs_0, obs_1, obs_2, obs_3, obs_4] \tag{8}$$

Where each obs_i signifies which kind of tile the agent is located on, as such:

obs_0	Corridor Tile
obs_1	Start Tile
obs_2	Turning Point
obs_3	Maze End, Dead end of a corridor
obs_4	Maze Reward

In the later stages of the project the agent observation was expanded to include three more points of data in addition to the original (8) One Hot Encoding (OHE). Specifically the NN agent exhibited some difficulty when traversing turns and recognizing which directions of movement were available. As such three more binary digits were added that represented on which sides the agent was surrounded by walls.

$$[obs_{agent}, obs_5, obs_6, obs_7] \tag{9}$$

obs_5	Wall to the left
obs_6	Wall to the right
obs_7	Wall in front

The human actions were represented by a single integer from 0 to 2, for each action the agent can perform, turning left, turning right, going forward, accordingly. While the reward was a floating point number. A penalty of -0.05 was applied for each step, and an additional penalty of -0.4 was given if the agent did not manage to find the reward or return to the starting tile, in the case of a homing task. If the agent managed to pick up the reward during a homing task, a reward of 0.5 was given, if the agent also managed to return successfully back to the start, it was also rewarded with an additional reward of 1.0. This reward also applied for non-homing tasks when the agent reached the reward.

3.5 Evaluation

Before training or testing the traces were split to two different sets. A training set with a size of 80% and a test set of 20% and their order was shuffled once after being retrieved by the system file manager. After training the models were evaluated by their performance in the T-Mazes, based on the amount of rewards and pickups that were acquired in addition to how few times the model was stuck in a destructive situation.

3.6 Long Short-Term Memory Model

The LSTM was the conventional model we used to solve the T-Maze exploration problem. The model was readily available from the PyTorch library. The single, unstacked

LSTM was initialized with an input dimension, dim_{input}, equal to the size of the obs_{agent} and an additional bias value that was always active, specifically 6 for the original observation format which only included tile information and 9 later on using the expanded observation format that encodes the sides where walls surround the agent as well. When used as a classifier the LSTM must have a hidden dimension equal to the population of the classes. Thus we used a hidden dimension, dim_{hidden}, of 3, equal to the population of available actions. Furthermore the hidden h and cell state c were initialized as a tensor of zeroes.

The LSTM worked in conjuction with a Loss Function or criterion, either Cross Entropy Loss or the combination of the Softmax and NLLLoss, and an Adam optimizer. The output of the network was compared to the OHE of the $action_{human}$ to calculate loss and propagate it backwards through the network and adjust the LSTM accordingly.

3.6.1 Long Short-Term Memory Training

The training of the LSTM was conducted with providing it with the human traces collected in the first stage of the project. Before starting training, we clear the accumulated gradients of the LSTM model and the Optimizer improving the training and testing performance of our model. The traces from the train set would be given one by one to the LSTM where they were parsed line by line. Lines that contained maze or seed information were discarded, while lines containing the recording of a step were parsed. To the obs_{agent} a bias bit was added and the whole arrangement was converted to a tensor and reshaped to a (1, 1, dim_{input}) tensor. That is, a single batch, single sequence tensor with size equal to the dimension of the input. This tensor format was used as the input for the NN at every step.

After retrieving the LSTM output, the hidden and cell states, h, c, were saved to be given to the LSTM on the next step of input. While the OHE of the $action_{human}$ was given to the loss function. The resulting loss was backpropagated through the LSTM. During the training of the LSTM the sum of the loss and the percentage of hit for all actions, as well as the percentage of hit specifically for left or right actions were calculated. The two different percentages were to ensure the NN did not favour one action over the other, as our data sets were naturally biased towards forward actions, since our mazes contained more straight corridors than turns.

3.7 Backpropamine Network Model

The Backpropamine network has a similar structure to the LSTM network, albeit with different internal algorithms. The model is based on Miconi's work for UberAi. The Backpropamine network was initialized with the dim_{input} similarly to the LSTM [3.6] and the size of the hidden layer, $hidden_{\mathrm{size}}$.

The optimizer was initialized with the learning rate, η, and a weight decay value, wd. Furthermore the loss function used weight values, w_{l0}, w_{l1}, w_{l2}, that were applied to the loss of the left, right, forward, action classes. Higher value w_l work to counteract skewness towards a majority class, since, just by its sheer majority the Neural Network pays more attention to that class.

3.7.1 Backpropamine Network Model Training

Once more the Backpropamine Network, similarly to the LSTM, used the human traces as input and the human action as target output. Before starting training the gradients for the Network and the Optimizer were zeroed and the hidden state and hebbian traces, h, c cleared. The recorded steps from the Trace were parsed. The observations of the agent, obs_{agent} along with a bias bit were converted to a (dim_{input}) tensor. Given that tensor, $input_{tensor}$, the hidden h, and hebbian traces, c the ANN would output the raw scores of the possible actions, y, the out value or its "chosen action", y, as well as the changed hidden state and hebbian traces, h', c'. y and a tensor with the integer value of $action_{human}$ are then given to the loss function which calculates the loss, which is then backpropagated through the network. At the end of the trace we stepped the optimizer and zeroed once more the gradients of the Network and the Optimizer

3.8 Softmax function

A Softmax function takes as input a vector z of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. Within the bounds of our project that means taking the real numbers a Neural Network produces after being given an input and converting them into probabilities that represent which action is most likely to be the correct one.

The softmax function is defined by the formula

$$\sigma(z)_i = \frac{e^{zi}}{\sum_{j=1}^{K} \mathrm{e}^{zj}} \text{ for i = 1,...,K and z = } (z_1, \ldots, z_k) \in \mathbb{R}^K \tag{10}$$

$\sigma(z)_i$	:	Softmax set
e^{zi}	:	Exponent function applied to value z_i
$\sum_{j=1}^{K} \mathrm{e}^{zj}$	:	Sum of exponentials

3.9 Normalizing Loss

Not all mazes are created equal, some are larger, more complex, and repeated for more episodes, while others are smaller, simpler. Human participants did not always find the most efficient path towards the reward. As a result some Traces were much larger than others and Loss would be accumulated much more easily even though the NN would make less mistakes than in previous batches it was given. To combat this problem the loss value would be normalized according to the length of the trace file it originated from with the following function:

$$l_{\text{j norm}} = l_j * \left(\frac{\sum_{i=1}^{n} len_i}{n}\right) / len_j \tag{11}$$

l_j : The original loss value that originated from passing the Trace file j through the training phase.

$$\frac{\sum_{i=1}^{n} len_i}{n} \tag{12}$$

:The average length of all Trace files use in the training phase.

$l_{\text{j norm}}$: The Normalized loss value.
len_j : The length, in lines, of the Trace file j

3.10 Weighted Loss

To combat class imbalance of a naturally skewed dataset a weight was applied for each action at the loss function. Using the Inverse Number of Samples (INS) function which is a simplistic and popular weighting scheme. For each action its weight was equal to 1 over the Number of Samples in Class c.

$$w_{\text{n,c}} = \frac{1}{\text{Number of Samples in Class c}} \tag{13}$$

The logic behind this function is that the loss of a large Class matters less than the loss of a small Class discounting it and giving more attention to classifying correctly the small Class. Since the large Class will be repeated many times, the Neural Network will equally learn how to handle it.

Chapter 4
Results and discussion

4.1 Long Short-Term Memory Learning

The LSTM was relatively simpler than the Backpropamine Network. Since the Network was unstacked we only needed to test different values for the learning rate η and the weight decay $L2$ that the Optimizer would use.

4.1.1 LSTM Learning Rate

Several different values of η stood out when training the LSTM network. Namely 0.15, 0.2, 0.3 which displayed the best accuracy in categorizing the observations given to the Network.

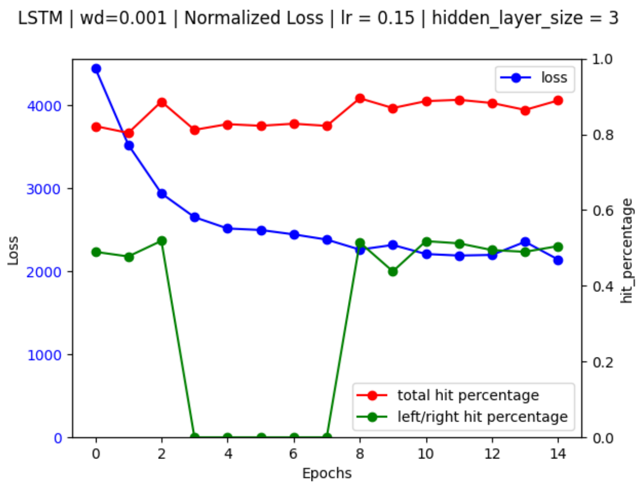


Figure 4: LSTM Loss and hit percentage during training. Training Traces=15 Learning Rate η=0.15. Weight Decay L2=0.001, Hidden Layer size h_{dim} *=3, Weighted Loss = [1.0, 1.0, 1.0]*

The LSTM with Learning Rate value η=0.15 [Fig. 4] displays a good fit Learning Curve. It approximately manages to categorize 90% of the all the actions in the training set, and ~50% of all the Left/Right actions.

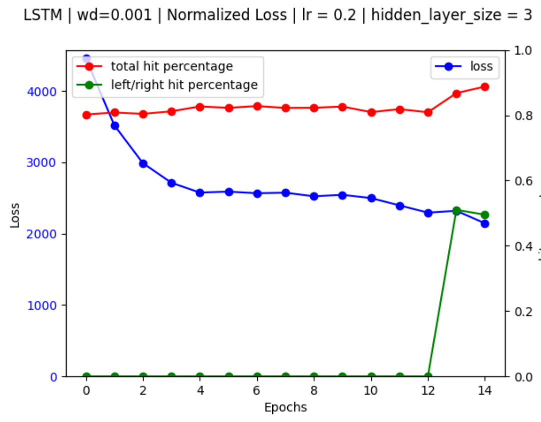


Figure 5: LSTM Loss and hit percentage during training with 15 different traces with Learning Rate η=0.2. Weight Decay L2=0.001, Hidden Layer size=3, Weighted Loss = [1.0, 1.0, 1.0]

Likewise with an η=0.2 [Fig. 5] at the end of the training, the LSTM manages to achieve ~90% accuracy in all actions and ~50% accuracy in categorizing Left/Right actions. The late jump of accuracy in the hit percentage of the Left/Right actions might mean that the η might be too large and the NN has a hard time settling down on the correct values for its weights.

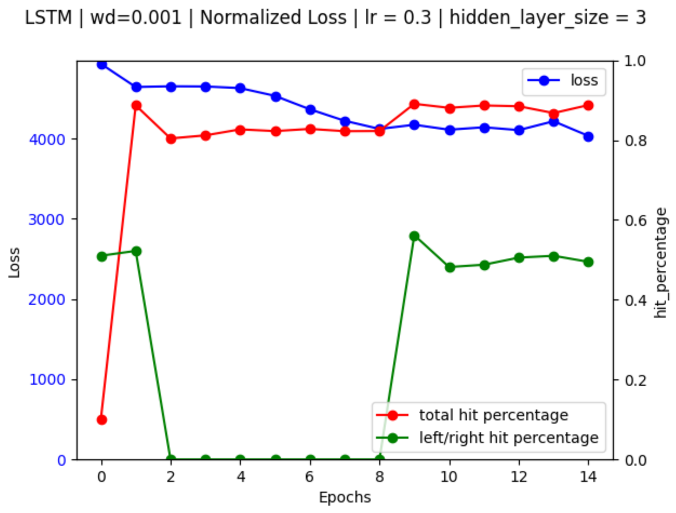


Figure 6: LSTM Loss and hit percentage during training with 15 different traces with Learning Rate η=0.3. Weight Decay L2=0.001, Weighted Loss = [1.0, 1.0, 1.0]

With a Learning Rate η=0.3 [Fig. 6], the training results in a similarly accurate results with the two other values yet does not seem to properly reduce Loss as well as with a Learning rate of 0.15 or 0.2

From the above figures it seems the most appropriate and accurate NN is created with a Learning Rate η=0.15 as it combines both the best Loss Curve and the best action hit percentages.

4.1.2 LSTM Weight Decay

We tested various various values of Weight Decay on a logarithmic scale from 0 to 0.1 in order to understand how it affected the training of the NN. As our data in general was not noisy, Weight decay did not affect the training of the Network very much. Nevertheless values larger than 0.001 resulted in bad training fitment.

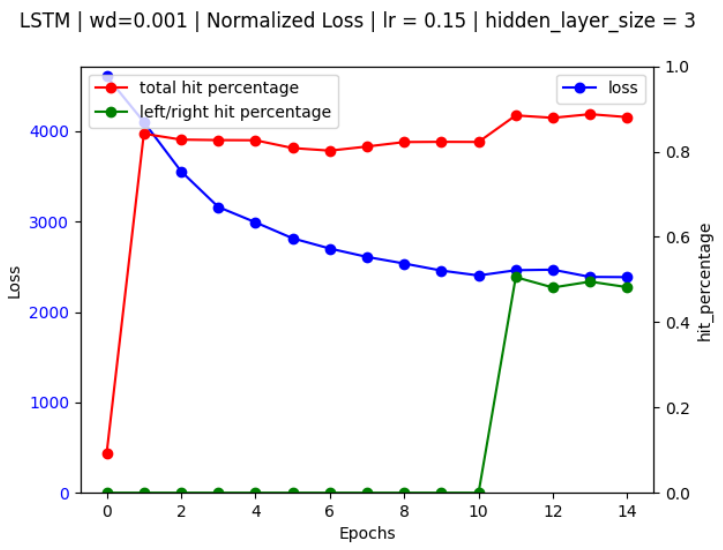


Figure 7: LSTM Loss and hit percentage during training with 15 different traces with Weight Decay L2=0.001. Learning Rate η=0.15, Weighted Loss = [1.0, 1.0, 1.0]

The LSTM in figure 4, shows a good loss curve and unimpacted action hit percentages.

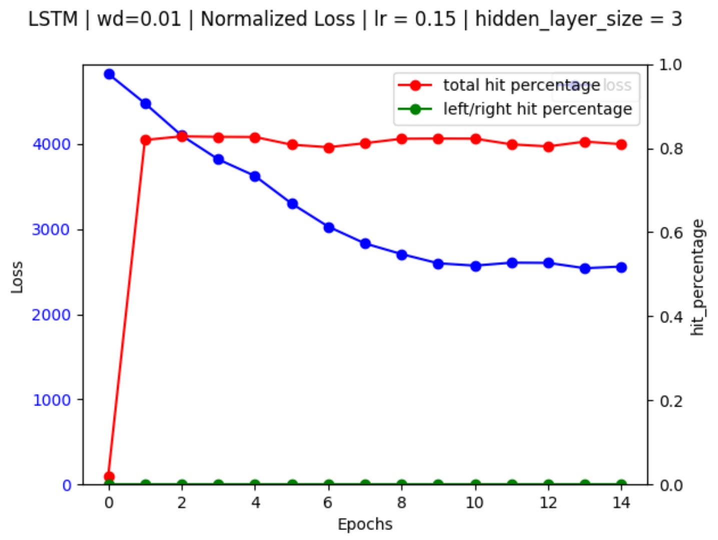


Figure 8: LSTM Loss and hit percentage during training with 15 different traces with Weight Decay L2=0.01. Learning Rate η=0.15, Weighted Loss = [1.0, 1.0, 1.0]

The LSTM in figure 5, shows that the loss curve plateaus later than in figure 4, while also not learning how to handle turns.

4.1.3 Weighted vs Unweighted Loss - LSTM

In an attempt to improve the LSTM's Left/Right action accuracy we applied weights to the loss function using the formula above and comparing them with unifrom Loss Weights while keeping the rest of the NN parameters the same.

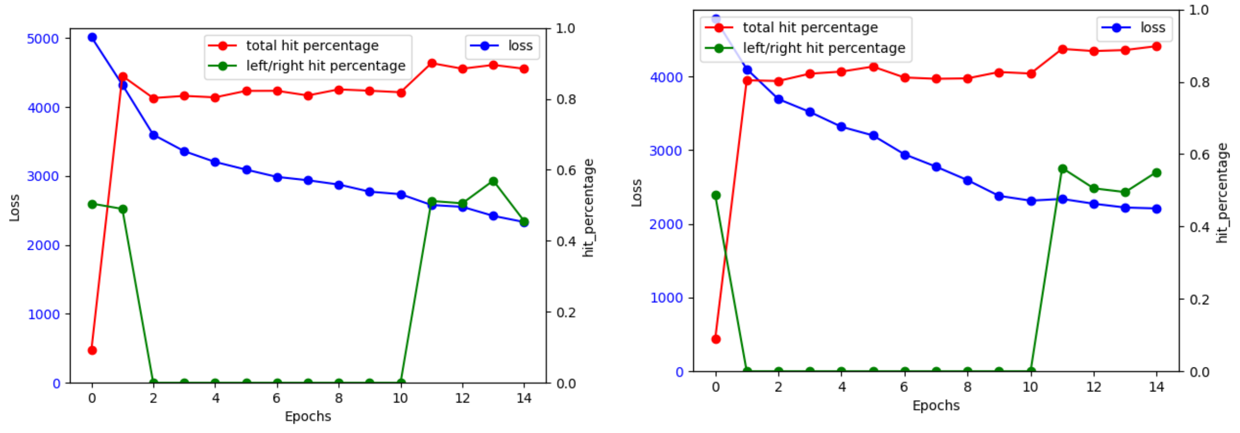


Figure 9: Loss and hit percentage for 15 different traces. ***(left)*** *Weighted Loss = [1, 1, 1]*

(right) *Weighted Loss = [1/6, 1/6, 1/53]*

From Fig. 6 we can deduce that the unweighted and weighted loss have similar results given that the rest of the parameters remain the same. The one minor difference worth noting is that the loss of the LSTM with the Weighted Loss Function plateaus earlier. Academic literature supports our findings and states that a Weighted Loss Function is needed when dealing with generally noisy data [14].

4.2 Backpropamine Network Learning

For the Backpropamine Network we had to ascertain the optimal values for η , $L2$ similarly to the LSTM [4.1.1, 4.1.2, 4.1.3] as well as the size of the Hidden Layer. A larger HiddenLayer usually exhibits greater loss, as there are more neurons to be corrected for each step, but might achieve greater accuracy as it can encode information better.

4.2.1 Backpropamine Network Learning Rate

When testing the Backpropamine Network we use an η in the range of 0.001 to 1.0. From those values, the learning rates that seemed to perform the best were 0.01, 0.02. In the following figures we will analyse how they affect the networks loss and hit percentage.

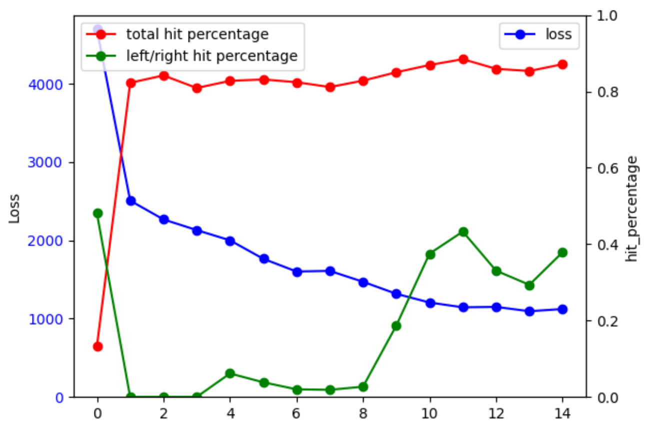


Figure 10: BPN Loss and hit percentage during training with 15 different traces with Learning Rate η=0.01. Weight Decay L2=0.001, Weighted Loss = [1.0, 1.0, 1.0]

Analysing the results in fig. 7 the NN displays a good loss curve though is slow to plateau. Nevertheless the network achieves a hit percentage of ~90% and and a Left/Right hit percentage of ~40%.

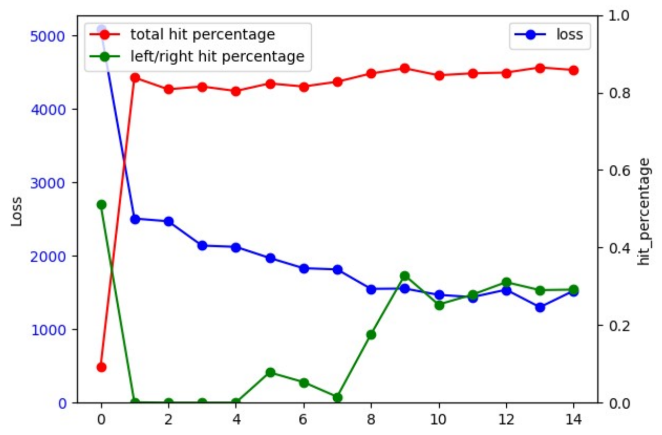


Figure 11: BPN Loss and hit percentage during training with 15 different traces with Learning Rate η=0.02. Weight Decay L2=0.001, Weighted Loss = [1.0, 1.0, 1.0]

Similarly with a Learning Value η=0.02 [Fig. 11] the NN ties with the previous execution in terms of final action hit. The network only has a hit percentage of ~85% and and a Left/Right hit percentage of ~30% yet it manages to achieve that value much more consistently. Meanwhile Loss is not being decreased harmoniously with every trace.

4.2.2 Backpropamine Network Weight Decay

Similarly to the LSTM the Weight decay values we tested ranged on a logarithmic scale from 0 to 0.1. Values lower than 0.001 did not seem to impact performance while values higher than that hindered the Neural Network's ability to learn.

4.2.3 Backpropamine Network Hidden Layer Size

The size of the Hidden Layer drastictly affected the capabilities of the Backpropamine Network. Various sizes from 1 to 100 were tested but the results showed that smaller to medium sized networks performed better and were easier to train.

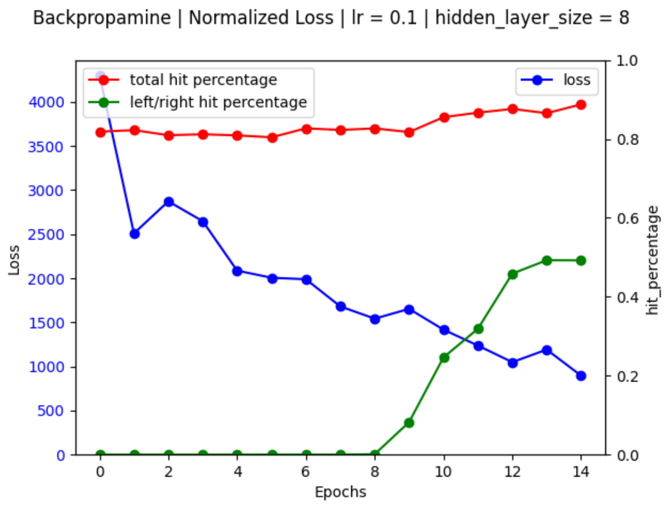


Figure 12: BPN Loss and hit percentage during training, using Hidden Layer Size equal to 8. 15 different traces. Learning Rate η=0.1. Weight Decay L2=0.001, Weighted Loss = [1/6, 1/6 1/53]

Small networks (Hidden Layer < 10) [Fig. 12] proved that they can still learn the rules of the maze and could classify properly the actions from the traces. They did not present the same Hit Percentage drop that larger mazes did and were less computationally intensive when training. While the results are good on paper they were not consistent, thus on some runs the network failed to learn the correct action for some part of the classification. Resulting in an agent that in most cases gets softlocked in the testing phase.

A larger Hidden Layer [Fig. 14] helped ameliorate the problem where agents would get softlocked and produced more consistent learning. Though would often display some "forgetfullness" until its performance balances, this could be a result of the vanishing or exploding gradients.

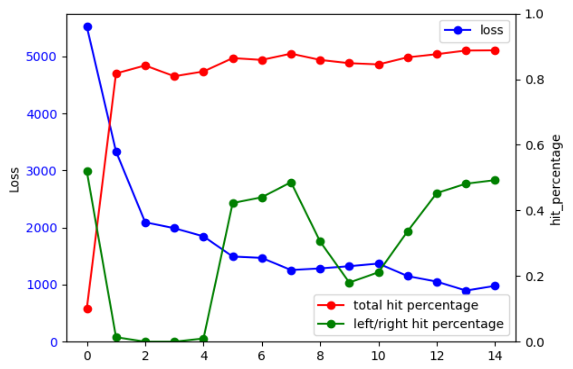


Figure 14: BPN Loss and hit percentage during training, using Hidden Layer Size equal to 20. 15 different traces. Learning Rate η=0.1. Weight Decay L2=0.001, Weighted Loss = [1/6, 1/6 1/53]

4.3 Training on traces Vs Training on Epochs

Up until now the Mazes were trained on individual Traces, the whole epoch repeating only once. This while creating partially successful results and making the network easy to train and avoiding overfitting does not produce consistent results and the Neural Network sometimes does not manage to learn all the “rules” of the maze. Deciding on the number of Epochs is a difficult task since we need to balance things in such a way that our Network does not lose the ability to generalize yet still manages to learn in the whole course of training as best as possible.

For this, we pass the Training Traces through the Neural Network multiple times, and testing with our Testing Traces once. We tested for how many Epochs each Network should be trained for. Metrics for the optimal number of Epochs include the Loss, Hit percentage, L/R Hit percentage we used earlier. From the resulting figures we will then determine when we need to stop training early depending on how the networks perform in the testing phase.

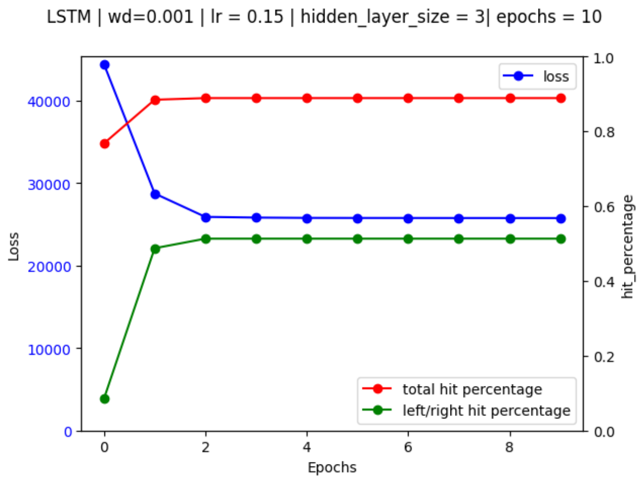


Figure 15: LSTM loss and hit percentage. Training traces=12. Epochs=10, Weight Decay L2=0.001. Learning Rate η=0.15, Weighted Loss = [1.0, 1.0, 1.0]

From the fig. 15 above we can see that the LSTM Network after 3 training epochs still displays a flat trend regarding the Loss and Hit percentage values. This trend continues for the rest of the epochs tested. For a bigger number of epochs the LSTM displays the same performance without any flunctuations in the resulting Hit percentages.

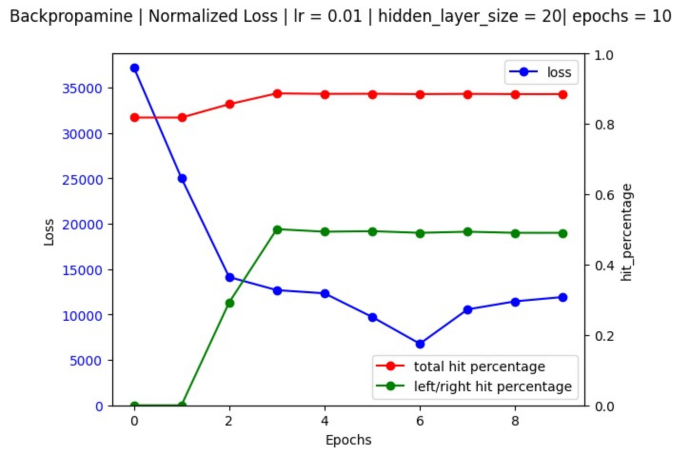


Figure 16: Backpropamine Train loss and hit percentage. Training traces=12. Epochs=10, Weight Decay L2=0.001. Learning Rate η=0.15, Weighted Loss = [1.0, 1.0, 1.0]

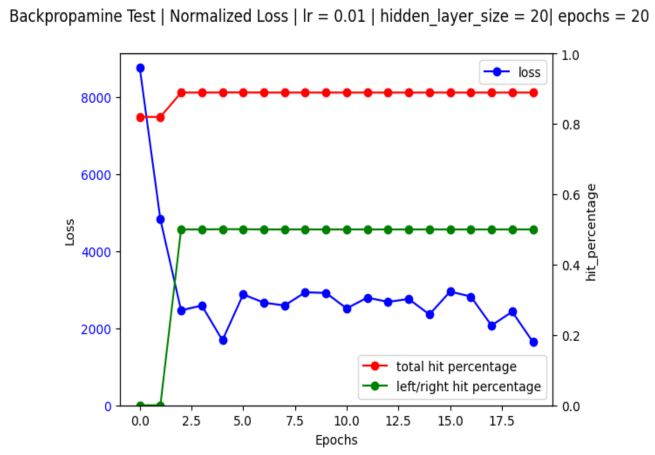


Figure 17: Backpropamine Test loss and hit percentage. Training traces=12. Epochs=10, Weight Decay L2=0.001. Learning Rate η=0.15, Weighted Loss = [1.0, 1.0, 1.0]

Regarding the Backpropamine Network [fig. 16, 17]the number of epochs requires more finetuning in comparison to the LSTM. When trained with 3 Epochs the Network seems to be stops learning how to classify the Traces given to it. Hit percentages display an upwards trend while Loss displays the opposite. After 7 epochs loss seems to display a reverse peak as a greater number of epochs displays a detrimental effect on the loss of the network. Futhermore we can see from the test results we can see that the network achieves almost identical results in training and testing.

4.4.1 Long Short-Term Memory Performance

With the initialization values we gathered above, we setup our LSTM to be trained and then tested in actual T-Maze environments. We gathered how many times the agent managed to collect the reward or a pickup.

Table 2: Average(r=3) LSTM network performance. $\eta=0.15$, $L2=0.001$, Training Traces=12, Testing Traces=3, Epochs=4.

Maze	Rewards	Pickups	Total Episodes
Tmaze	0	N/A	22
DoubleTMaze	0	N/A	30
TripleTMaze	20	N/A	60
QuadTMaze	0	N/A	104
TmazeHoming	0	4	15

DoubleTMazeHoming	0	0	29
TripleTMazeHoming	0	0	53
QuadTMazeHoming	0	0	101

The LSTM Network displays lackluster performance [Table 2] in the various tasks it is presented with. While it displays some basis of learning [Fig. 16,17] when provided with the traces it does not manage to apply that experience in the maze and collect the rewards. Its success when finding the reward in the Triple Non-Homing maze seems to be based on luck.

Table 3: Average(r=3) LSTM network performance compared to humans. $\eta=0.15$, $L2=0.001$, *Training Traces=12, Testing Traces=3, Epochs=4. Optimal reward is an estimation based on the maximum reward-(half of the allowed steps * step penalty).*

Maze	Human average	LSTM average	Minimum reward	Optimal reward
TMaze	0.80	-0.29	-0.9	0.75
DoubleTMaze	0.62	-1.35	-1.4	0.5
TripleTMaze	-0.04	-1.23	-2.15	0.13
QuadTMaze	-0.79	-3.85	-3.9	-0.75
TMazeHoming	0.93	-0.79	-1.8	1
DoubleTMazeHoming	-0.34	-2.35	-2.8	0.5
TripleTMazeHoming	-1.75	-3.72	-4.3	-0.25
QuadTMazeHoming	-4.83	-7.29	-7.8	-2

Comparing to the human averages we can see that the LSTM does not manage to achieve that. While our human participants are close to the optimal reward estimation the models inability to collect rewards hinders it a lot.

4.4.2 Backpropamine Network Performance

The performance of the Backpropamine [Table 4] was questionable. The agent displayed some elements of understanding of how to navigate the maze and some sort of exploration strategy but would often get stuck repeating the same invalid actions Sometimes such hiccups would sort themselves out while others it would bring the whole computer system to a kneel. Memorisation was mediocre. While sometimes displaying memorisation of the reward's previous location, the would often revisit the same empty corridors walking back and forth from the two end points.

Table 4: Average(r=3) Backpropamine network performance. Hidden Layer Size = 20, $\eta = 0.01$, $L2 = 0.001$ *, Training Traces=12, Testing Traces=3, Epochs=1.*

Maze	Reward	Pickups	Times Stuck	Total Episodes
Tmaze	6	N/A	0	22
DoubleTMaze	0	N/A	17	30
TripleTMaze	0	N/A	52	60
QuadTMaze	1	N/A	103	104
TmazeHoming	4	4	6	15
DoubleTMazeHoming	0	0	29	29
TripleTMazeHoming	0	0	44	53
QuadTMazeHoming	0	0	101	101

Comparing the average of backpropamine [Table 4] to that of the LSTM network [Table 2] the results aren't much better. Backpropamine displays more diversity, even achieving to solve the single depth homing task successfully but it does not achieve consecutive rewards in the same way the LSTM does in regards to the triple depth non-homing task.

Table 5: Average(r=3) Backpropamine network performance compared to humans. $\eta = 0.01$ *,* $L2 = 0.001$ *, Hidden Layer Size=20, Training Traces=12, Testing Traces=3, Epochs=4. Optimal reward is an estimation based on the maximum reward-(half of the allowed steps * step penalty).*

Maze	Human average	BP average	Minimum reward	Optimal reward
TMaze	0.80	-0.18	-0.9	0.75
DoubleTMaze	0.62	-0.87	-1.4	0.5
TripleTMaze	-0.04	-2.10	-2.15	0.13
QuadTMaze	-0.79	-3.42	-3.9	-0.75
TMazeHoming	0.93	-1.14	-1.8	1
DoubleTMazeHoming	-0.34	-2.03	-2.8	0.5
TripleTMazeHoming	-1.75	-3.85	-4.3	-0.25
QuadTMazeHoming	-4.83	-7.29	-7.8	-2

In terms of average rewards [Table 5] backpropamine scores similarly to the LSTM. It does achieve a better score when tested in single depth non-homing mazes but it does not achieve a value close to the optimal reward.

4.5 Expanding The Observation

As stated before [3.4.1] after the initial round of training the Networks the first Observation was expanded to include three distinct boolean variables that specified if there was a wall, left, right, or infont of the agent. This was done in order to counteract a flaw both networks exhibited, turning towards walls when passing a turn the agent already visited and sometimes getting stuck ad infinitum. One reason for that could be that agent is not aware of their orientation when approaching a turn, as seen in fig. 18 and thus fails to properly classify the observation and pick the correct action. We tested if an expansion to the observation would help the two Networks avoid softlocking while also achieving better results when tested in the virtual T-Mazes.

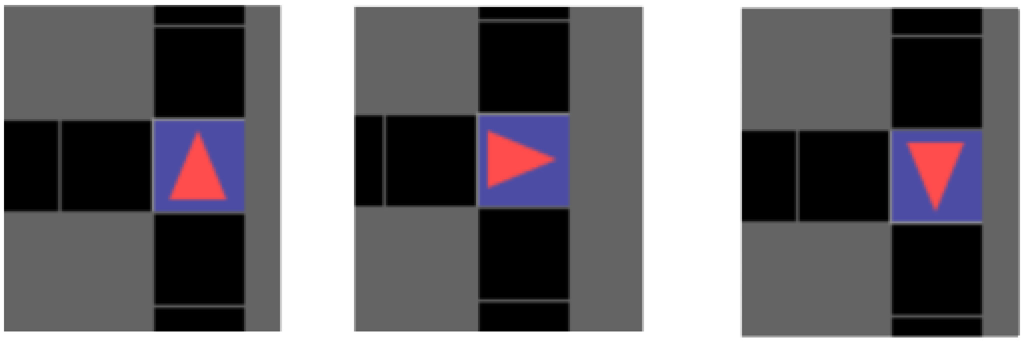


Figure 18: All these cases produce the same observation for the agent, yet the avialable actions are different.

There were seven different Human Traces collected in comparison to the fifteen Human Traces that were used in the Original Observation Model. Nevertheless the Backpropamine Network improved in performance even with the reduction in the amount of Trace Files that it was trained on previously.

Table 6: Backpropamine Testing Performance. Epochs=7, Learning Rate=0.01, Hidden Layer Size=20, 5 Training Traces, 2 Testing Traces, r=3

Backpropamine Maze Performance	**Reward**	**Pickups**	**Times Stuck**	**Total Episodes**
Tmaze	4	N/A	0	11
DoubleTMaze	12	N/A	0	21
TripleTMaze	17	N/A	1	35
QuadTMaze	35	N/A	15	70
TmazeHoming	2	4	0	9
DoubleTMazeHoming	0	9	0	17
TripleTMazeHoming	0	5	26	37
QuadTMazeHoming	16	17	33	65

From the above table we can conclude that the Backpropamine Network manages to reach the reward around half of the episodes in Non-Homing Tasks in comparison to its previous performance without the expanded observation [Table 1].

We can also see that in Homing Tasks it also manages to pickup the reward for a number of episodes. Although for the Double or Triple depth tasks it does not succeed in retracing its steps back to the original position. This does not apply to the Single or Quadruple depth tasks where the ANN agent returns to their original position most of the time.

While the results of the Backpropamine Network are good, the extended Observation does not manage to eliminate the possibility of the agent getting stuck while exploring the larger mazes.

Table 7: LSTM Testing Performance. Epochs=7, Learning Rate=0.15, Hidden Layer Size=3|

5 Training Traces, 2 Testing Traces, r=3

LSTM Maze Perfomance	Reward	Pickups	Times Stuck	Total Episodes
Tmaze	4	N/A	0	11
DoubleTMaze	6	N/A	0	21
TripleTMaze	0	N/A	0	35
QuadTMaze	0	N/A	0	70
TmazeHoming	0	4	0	9
DoubleTMazeHoming	0	4	0	17
TripleTMazeHoming	0	0	0	37
QuadTMazeHoming	0	0	0	65

The LSTM network does not benefit as much from the adjustment of the Observation Format. The LSTM until now proved to be more resistant to getting stuck in a T-Maze than the Backpropamine Network and its results in regards of achieving the Maximum Reward or Picking up the Reward in Homing Tasks does not significantly improve.

4.6 Network Viability – Comparison

The Backpropamine Network performs better than the LSTM in almost all of the various tasks. Although the Network suffers from repeating the same actions or behaviours in certain situations that softlock it. It is much more flexible when tested in a T-Maze environment and much more resistant to overtraining.

The LSTM's ease of use and availability do not outweight its mediocre performance in the T-Mazes. It fails to consintently navigate towards the reward in mazes of depth larger than Double and does not exhibit the same memorisation in the homing tasks that the Backpropamine Network does.

The LSTM's lackluster performance could be that it lacks the continuously plastic components that the Backpropamine network features.

Table 8: Comparison of the LSTM and Backpropamine in regards to performance in the T-Mazes. Using tables 6, 7.

Maze	**LSTM Reward**	**LSTM Pickups**	**LSTM Times Stuck**	**BP Reward**	**BP Pickups**	**BP Times Stuck**	**Total Episodes**
Tmaze	4	N/A	0	4	N/A	0	11
DoubleTMaze	6	N/A	0	12	N/A	0	21
TripleTMaze	0	N/A	0	11	N/A	1	35
QuadTMaze	0	N/A	0	33	N/A	15	70
TmazeHoming	0	4	0	2	3	0	9
DoubleTMazeHoming	0	4	0	0	8	0	17
TripleTMazeHoming	0	0	0	0	5	26	37
QuadTMazeHoming	0	0	0	16	17	33	65
Total:	10	8	0	86	35	75	

Chapter 5

Conclusions and Future Work

5.1 Overview and conclusions

This thesis aimed to explore how achievable structured exploration behaviour is in an Artificial Neural Network and test the ability of Neuromodulated Plasticity Rules to attain that. Furthermore we compared how Neuromodulation performs against a different model of Neural Networks, the LSTM. We developed a randomized and parameterized environment of T-Mazes that allows us to create human traces and test the performance of ANNs for different kinds of tasks.

Furthermore we successfully managed to collect a number of Traces from human Participants, and feed them to two different models of neural networks. Testing different combinations of parameters for the Neural Networks we found the best performers and put them to the test in the different task environments. We analyzed and compared their performance and have found that a network based on the Differentiable Plasticity with Backpropamine model can effectively learn from human actors how to explore a T-Maze of various depths and perform well in the homing tasks, exhibiting memorisation and retracting its steps performing much more consistently than its conventional counterpart. We also determined that the LSTM network is more stuck resistant in comparison to the neuromodulated plasticity network as long as both networks are properly trained.

On a side note, we have tested how a weighted and unweighted loss function affects the training and performance of the two models when using training data without any noise and have found minimal effect on both the LSTM and backpropamine network.

5.2 Future Work

Throughout this project neuromodulated plasticity has shown its ability to imitate human behaviour and continuously learn while applied to a task. Nevertheless to investigate the full potential of our approach we need to reconsider how the model is trained and ways to minimize the destructive cases when the agent is traversing through the maze. An application of this model could help solve other hard tasks for machines that are relatively easy for humans. Using both LSTM and Backpropamine models in a sequential approach could get us the best of both worlds, combining LSTM's stuck resistance with the backpropamine's ability in exploration.

Combining backpropamine with different models or architectures that we haven't explored during this project could also expand the abilities of the network. For example using a convolutional neural network can possibly scale this project to the 3D world. If we recall the 3D maze screensaver that was used in the Windows 95 era we can potentially emulate such an environment and outperform the left-hand rule that the implementation of the "player" used.

The controlled T-Maze environments we have created and used during this project can, in the future, be replaced with more complex environments or tasks with higher stakes. We can potentially pit the human participants and the trained agents against each other in a form of adversarial learning. Creating an framework where the neural network has a clear target that it needs to outperform as well as available traces to learn from.

BIBLIOGRAPHY

1. Hochreiter, Sepp & Schmidhuber, Jürgen. (1997). Long Short-term Memory. Neural computation. 9. 1735-80. 10.1162/neco.1997.9.8.1735.

2. Blynel, Jesper & Floreano, Dario. (2003). Exploring the T-Maze: Evolving Learning-Like Robot Behaviors using CTRNNs. 593-604. 10.1007/3-540-36605-9_54.

3. Soltoggio, Andrea. (2008). Neuromodulation Increases Decision Speed in Dynamic Environments. 139.

4. Vassiliades, V., & Christodoulou, C. (2016). Behavioral plasticity through the modulation of switch neurons. Neural networks : the official journal of the International Neural Network Society, 74, 35–51

5. Wunder, Michael & Littman, Michael & Babes-Vroman, Monica. (2010). Classes of Multiagent Q-learning Dynamics with ε-greedy Exploration. ICML 2010 - Proceedings, 27th International Conference on Machine Learning. 1167-1174.

6. Li, H., Zhang, Q., & Zhao, D. (2019). Deep reinforcement learning-based automatic exploration for navigation in unknown environment. IEEE transactions on neural networks and learning systems, 31(6), 2064-2076.

7. Burda, Y., Edwards, H., Storkey, A., & Klimov, O. (2018). Exploration by random network distillation. arXiv preprint arXiv:1810.12894.

8. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

9. O'Keefe J, Dostrovsky J. The hippocampus as a spatial map. Preliminary evidence from unit activity in the freely-moving rat. Brain Res. 1971;34(1):171-175.

10. Miconi, T., Stanley, K., & Clune, J. (2018, July). Differentiable plasticity: training plastic neural networks with backpropagation. In International Conference on Machine Learning (pp. 3559-3568). PMLR.
11. Schultz W, Dayan P, Montague PR. A neural substrate of prediction and reward. *Science*. 1997;275(5306):1593-1599.
12. Miconi, Thomas, et al. "Backpropamine: training self-modifying neural networks with differentiable neuromodulated plasticity." arXiv preprint arXiv:2002.10585 (2020).
13. Stevens, S. S. (1946). On the theory of scales of measurement. Science, 103, 677–680
14. Byrd, J., & Lipton, Z. (2019, May). What is the effect of importance weighting in deep learning?. In *International Conference on Machine Learning* (pp. 872-881)

Appendices

Appendix A tmaze.py

```
import random

# import everything else from gym_minigrid
from gym_minigrid.minigrid import *

from gym_minigrid.register import register

# Reward Values
DISTINCT_OBJECTS = 5

LOW = 0.2                   # Low reward
HIGH = 1                    # High Reward
PICKUP = 0.5                # Used in homing tasks, when the agent manages to
reach the first goal. Before returning home
FAIL_HOME = -0.4            # If the agent manages to pickup the reward but not
return home
CRASH_PENALTY = -0.4        # When the agent touches lava or crashes into a wall.
Might not be used
STEP_PENALTY = -0.05        # Penalty for every step taken

# Helper class for keeping track of reward positions and ends of corridors
class MazeEnd(Goal):
    def __init__(self, color='black', reward='low'):
        self.color = color
        self.turnAround = True
        if reward == 'high':
            self.reward = HIGH
        else:
            self.reward = 0
        super().__init__(color)

    def can_overlap(self):
        return True

    def render(self, img):
        # Give the floor a pale color
        color = COLORS[self.color]
        # fill_coords(img, point_in_rect(1, 1, 0.031, 1), color)
        fill_coords(img, point_in_rect(0.031, 1, 0.031, 1), color)

    def return_reward(self):
        return self.reward

    def __str__(self):
        return 'MazeEnd'

# The goal end of the maze
class MazeGoal(MazeEnd):
    def __init__(self, color='black'):
        super().__init__(color=color, reward='high')

    def __str__(self):
        return 'MazeGoal'

# The Start of the maze
class Start(MazeEnd):
    def __init__(self):
        self.turnAround = True
        super().__init__(color='green')

    def render(self, img):
        color = COLORS[self.color]
        fill_coords(img, point_in_rect(0.031, 1, 0.031, 1), color)

    def __str__(self):
        return 'Start'

class TMazeEnv(MiniGridEnv):
    # Actions that the agent is permitted to make
    class Actions(IntEnum):
        # Turn left, turn right, move forward
        left = 0
        right = 1
        forward = 2
        # Done completing task
        done = 6


    def __init__(self, high_reward_end=None, size=None, homing=False, corridors=3,
corridor_lengths=None, seed=None):
        self.reward_range = (-1, 1)  # unused
        self.actions = TMazeEnv.Actions
        self.high_reward_end = high_reward_end  # Which maze end has the high
reward.
        self.size = size
        self.homing = homing
        self.corridors = corridors
        self.corridor_lengths = corridor_lengths
        self.DISTINCT_OBJECTS = DISTINCT_OBJECTS

        if corridors == 3:
            self.max_steps = 10
        elif corridors == 7:
            self.max_steps = 20
        elif corridors == 15:
            self.max_steps = 35
        elif corridors == 31:
            self.max_steps = 70

        if self.homing:
            self.max_steps *= 2
        super().__init__(
            max_steps=self.max_steps,
            seed=seed,
            agent_view_size=1,
            see_through_walls=True
        )

        # Action enumeration for this environment
        self.actions = TMazeEnv.Actions
        # Actions are discrete integer values
        self.action_space = spaces.Discrete(len(self.actions))

    # Class which randomly generates maze corridor lengths
    # based on the number of the corridors
    def generate_corridors(self, corridors):
        def _check_overlap(corridor_lengths):
            if len(corridor_lengths) > 6:
                for i in range(7, len(corridor_lengths)):
                    parent_corridor = math.ceil(((i - 2) / 2 - 2) / 2)

                    # if the product of the length of a corridor and that of its
parent is negative
                    # then there is a danger of corridors or overlapping
                    if (corridor_lengths[i] * corridor_lengths[parent_corridor]) <
0:
                        if abs(corridor_lengths[parent_corridor]) <=
abs(corridor_lengths[i]) + 1:
                            stride = 2 if (i % 2) == 0 else -2
                            corridor_lengths[parent_corridor] =
(corridor_lengths[i] + stride) * -1

        corridor_lengths = []

        for i in range(corridors):
            corridor_lengths.append(super()._rand_int(3, 6))
            if (i % 2) != 0:
                corridor_lengths[i] *= -1

        _check_overlap(corridor_lengths)
        return corridor_lengths

    # Calculates how offset the start position needs to be from the edge of the
maze
    # measure the negative height which we add to the grid height
    # thus avoiding exceeding the limits of the grid
    def negative_grid_offset(self, corridor_lengths):
        max_negative_height = 0
        if len(corridor_lengths) > 3:
            for i in [3, 5]:
                children = [(2 * i + 1) * 2 + 2 - (i % 2), (2 * i + 2) * 2 + 2 -
(i % 2)]
                if children[1] < len(corridor_lengths):
                    for j in children:
                        if corridor_lengths[i] + corridor_lengths[j] <
max_negative_height:
                            max_negative_height = corridor_lengths[i] +
corridor_lengths[j]
                else:
                    if corridor_lengths[i] < max_negative_height:
                        max_negative_height = corridor_lengths[i]

        return max_negative_height

    # Calculates the size of the square grid that our maze will be placed in
    def maze_size(self, corridor_lengths):
        corridor_size = [0] * len(corridor_lengths)

        for i in reversed(range(len(corridor_lengths))):
            # find the subcorridors that are parallel to the parent corridor
            children = [(2 * i + 1) * 2 + 2 - (i % 2), (2 * i + 2) * 2 + 2 - (i %
2)]
            corridor_size[i] = abs(corridor_lengths[i])

            # if the subcorridors exist
            if children[1] < len(corridor_lengths):
                # measure the sum of the length of the corridors
                if corridor_size[children[0]] > corridor_size[children[1]]:
                    corridor_size[i] += abs(corridor_size[children[0]]) - 1
                else:
                    corridor_size[i] += abs(corridor_size[children[1]]) - 1

        # measure the negative height which we add to the grid height
        # thus avoiding exceeding the limits of the grid
        max_negative_height = self.negative_grid_offset(corridor_lengths)

        width = corridor_size[1] * 2 if corridor_size[1] > corridor_size[2] else
corridor_size[2] * 2
        height = corridor_size[0]

        if corridor_lengths[0] + max_negative_height < 0:
            height -= (corridor_lengths[0] + max_negative_height)

        size = 0

        if height > width:
            size = height + 2
        else:
            size = width + 2

        if (size % 2) == 0:
            size += 1

        return size

    def _gen_grid(self):
        # generates vertical corridors on the grid
        def _gen_vert_corridor(pos, length):
            stride = 1 if length > 0 else -1
            x = pos[0]
            y = pos[1]
            for i in range(0, length, stride):
                self.grid.set(x, y, None)
                y += stride
            return [x, y - stride]

        # generates horizontal corridors on the grid
        def _gen_horz_corridor(pos, length):
            stride = 1 if length > 0 else -1
            x = pos[0]
            y = pos[1]
            for i in range(0, length, stride):
                self.grid.set(x, y, None)
                x += stride
            return [x - stride, y]

        # recursively builds alternating vertical and horizontal corridors
        # in a tmaze manner
        # returns the ending points of the mazes, and every turn
        def recursive_builder(pos, seq, corridors, vert):
            maze_ends = []
            turning_points = []

            if vert:
                new_pos = _gen_vert_corridor(pos, corridors[seq])
            else:
                new_pos = _gen_horz_corridor(pos, corridors[seq])

            if seq < int(len(corridors) / 2):
                ends, turns = recursive_builder(new_pos, seq * 2 + 1, corridors,
not vert)
                maze_ends.extend(ends)
                turning_points.extend(turns)
                ends, turns = recursive_builder(new_pos, seq * 2 + 2, corridors,
not vert)
                maze_ends.extend(ends)
                turning_points.extend(turns)
                turning_points.append(new_pos)
            else:
                maze_ends.append(new_pos)

            return maze_ends, turning_points

        # generate the size of the corridors
        if self.corridor_lengths is None:
            self.corridor_lengths = self.generate_corridors(self.corridors)
            self.size = self.maze_size(self.corridor_lengths)
            self.width = self.size
            self.height = self.size
            # print("No corridors given")
            # print(str(self.corridor_lengths) + '\ngrid size=' + str(self.size))

        self.reward = 0
        self.grid = Grid(self.width, self.height)
        self.mission = "Find the Goal"

        # calculate the starting height of the agent
        start_height = 1
        max_negative_height = self.negative_grid_offset(self.corridor_lengths)

        if self.corridor_lengths[0] + max_negative_height < 0:
            start_height -= (self.corridor_lengths[0] + max_negative_height)

        self.agent_start_pos = (self.width // 2, start_height)
        self.agent_start_dir = 1

        # set all tiles to be blocked
        for i in range(self.width):
            for j in range(self.height):
                self.grid.vert_wall(i, j, 1)

        # create the paths using the lengths we generated earlier
        # self.turning_points = []
        self.maze_ends, self.turning_points = \
            recursive_builder(list(self.agent_start_pos), 0,
self.corridor_lengths, True)

        # Set the home square
        # Surrounding walls
        x = 0
        y = 0
        w = self.width
        h = self.height
        self.grid.horz_wall(x, y, w)
        self.grid.horz_wall(x, y + h - 1, w)
        self.grid.vert_wall(x, y, h)
        self.grid.vert_wall(x + w - 1, y, h)

        self.put_obj(Start(), self.agent_start_pos[0], self.agent_start_pos[1])
        for turn in self.turning_points:
            self.put_obj(Floor(color='blue'), turn[0], turn[1])

        # Set the maze ends with the correct rewards
        if self.high_reward_end is None:
            self.high_reward_end = self._rand_int(0, len(self.maze_ends))
        self.set_reward_pos(self.high_reward_end)

        # rotate maze according to seed
        rot = self._rand_int(0, 4)
        for i in range(rot):
            self.grid = self.grid.rotate_left()
            x = self.agent_start_pos[0]
            y = self.agent_start_pos[1]
            self.agent_start_pos = (y, self.grid.height - 1 - x)
            self.agent_start_dir -= 1
            if self.agent_start_dir < 0:
                self.agent_start_dir = 3

        # Place the agent
        if self.agent_start_pos is not None:
            self.agent_pos = self.agent_start_pos
            self.agent_dir = self.agent_start_dir
        else:
            self.place_agent()

    def set_reward_pos(self, i):
        self.put_obj(MazeGoal(), self.maze_ends[i][0], self.maze_ends[i][1])

        for j in range(len(self.maze_ends)):
            if j != i:
                self.put_obj(MazeEnd(), self.maze_ends[j][0], self.maze_ends[j]
[1])

    def reset(self):
        # Current position and direction of the agent
        self.agent_pos = None
        self.agent_dir = None

        # Generate a new random grid at the start of each episode
        # To keep the same grid for each episode, call env.seed() with
        # the same seed before calling env.reset()
        self._gen_grid()

        # These fields should be defined by _gen_grid
        assert self.agent_pos is not None
        assert self.agent_dir is not None

        # Check that the agent doesn't overlap with an object
        start_cell = self.grid.get(*self.agent_pos)
        assert start_cell is None or start_cell.can_overlap()

        # Item picked up, being carried, initially nothing
        self.carrying = None

        # Step count since episode start
        self.step_count = 0

        # Return first observation
        obs = self.gen_obs()
        return obs

    def step(self, action):
        # Some of the following code is copied from MiniGrid's step function
        self.step_count += 1
        reward = 0
        done = False

        # Get the position of and in front of the agent
        agent_pos = self.agent_pos
        fwd_pos = self.front_pos
        # Get the contents of the cell in front of the agent and where the agent
is standing
        agent_cell = self.grid.get(*agent_pos)
        fwd_cell = self.grid.get(*fwd_pos)

        # Rotate left
        if action == self.actions.left:
            if agent_cell is not None and agent_cell.type == 'floor':
                self.agent_dir -= 1
                if self.agent_dir < 0:
                    self.agent_dir += 4
                action = self.actions.forward
            else:   # remove invalid actions from step_count
                self.step_count -= 1


        # Rotate right
        if action == self.actions.right:
            if agent_cell is not None and agent_cell.type == 'floor':
                self.agent_dir = (self.agent_dir + 1) % 4
                action = self.actions.forward
            else:
                self.step_count -= 1

        # Get the position in front of the agent
        fwd_pos = self.front_pos
        # Get the contents of the cell in front of the agent
        fwd_cell = self.grid.get(*fwd_pos)

        # Move forward
        if action == self.actions.forward:
            if fwd_cell is None or fwd_cell.can_overlap():
                self.agent_pos = fwd_pos
                reward = STEP_PENALTY
            else:
                self.step_count -= 1

            # Check if we reached a Goal
            if fwd_cell is not None and type(fwd_cell) is MazeGoal:
                if self.homing:
                    reward = PICKUP
                    self.agent_dir = (self.agent_dir + 2) % 4   # rotate the agent

                    # remove the goal object from the floor
                    # replacing it with a grey floor tile
                    self.put_obj(MazeEnd(color='black'), self.agent_pos[0],
self.agent_pos[1])
                    self.put_obj(MazeGoal(color='red'), self.agent_start_pos[0],
self.agent_start_pos[1])
                    self.homing = False
                    reward = PICKUP
                else:
                    self.put_obj(MazeEnd(color='green'), self.agent_pos[0],
self.agent_pos[1])
                    self.agent_pos = fwd_pos
                    reward = HIGH
                    done = True
            # turn around the agent
            elif fwd_cell is not None and hasattr(fwd_cell, 'turnAround') == True:
                self.agent_dir = (self.agent_dir + 2) % 4  # rotate the agent

            # If the agent crashes the episode ends and a negative reward is given
            if fwd_cell is not None and fwd_cell.type == 'lava':
                done = True
                reward = CRASH_PENALTY

        # Checking for number of step taken to punish the agent when it fails to
return home
        if self.step_count >= self.max_steps:
            done = True
            reward = FAIL_HOME


        self.reward += reward
        obs = self.gen_obs()

        return obs, reward, done, {}

    # one hot take encoding of the observation of the agent
    # observation format =
    # [is agent in corridor, is agent at home, is agent at turning point, is agent
at maze end, is agent at goal]
    def OneHotEncoding(self):
        def wallOHE():
            # obj_type=Wall
            # [left, right, front]
            walls = [0] * 3

            right_vec = self.right_vec
            left_vec = np.array((right_vec[0] * -1, right_vec[1] * -1))

            agent_pos = self.agent_pos
            left_pos = agent_pos + left_vec
            right_pos = agent_pos + right_vec
            fwd_pos = self.front_pos
            # Get the contents of the cells left, right, and forward of the agent
            left_cell = self.grid.get(*left_pos)
            right_cell = self.grid.get(*right_pos)
            fwd_cell = self.grid.get(*fwd_pos)

            if type(left_cell) is Wall:
                walls[0] = 1
            if type(right_cell) is Wall:
                walls[1] = 1
            if type(fwd_cell) is Wall:
                walls[2] = 1

            return walls



        image = [0] * self.DISTINCT_OBJECTS

        # Get the observation of where the agent is standing
        agent_pos = self.agent_pos
        observed_cell = self.grid.get(*agent_pos)
        agent_obs = -1

        if observed_cell is None:
            agent_obs = 0
        elif isinstance(observed_cell, Start):
            agent_obs = 1
        elif isinstance(observed_cell, Floor):
            agent_obs = 2
        elif isinstance(observed_cell, MazeEnd):
            agent_obs = 3
        elif isinstance(observed_cell, MazeGoal):
            agent_obs = 4

        image[agent_obs] = 1

        # image.extend(wallOHE())
        # Comment out for debugging purposes
        # print(image)
        # print(np.array(image))
        #image = torch.FloatTensor(image)
        #image = torch.reshape(image, (1, 1, 5))
        return image


class DoubleTMaze(TMazeEnv):
    def __init__(self, homing=False, seed=None, high_reward_end=None):
        self.homing = homing
        self.corridors = 7

        super().__init__(corridors=self.corridors, homing=self.homing, seed=seed,
high_reward_end=high_reward_end)


class TripleTMaze(TMazeEnv):
    def __init__(self, homing=False, seed=None, high_reward_end=None):
        self.homing = homing
        self.corridors = 15
        super().__init__(corridors=self.corridors, homing=self.homing, seed=seed,
high_reward_end=high_reward_end)


class QuadTMaze(TMazeEnv):
    def __init__(self, homing=False, seed=None, high_reward_end=None):
        self.homing = homing
        self.corridors = 31

        super().__init__(corridors=self.corridors, homing=self.homing, seed=seed,
high_reward_end=high_reward_end)


class TMazeHoming(TMazeEnv):
    def __init__(self, seed=None, high_reward_end=None):
        super().__init__(homing=True, seed=seed, high_reward_end=high_reward_end)


class DoubleTMazeHoming(DoubleTMaze):
    def __init__(self, seed=None, high_reward_end=None):
        super().__init__(homing=True, seed=seed, high_reward_end=high_reward_end)


class TripleTMazeHoming(TripleTMaze):
    def __init__(self, seed=None, high_reward_end=None):
        super().__init__(homing=True, seed=seed, high_reward_end=high_reward_end)


class QuadTMazeHoming(QuadTMaze):
    def __init__(self, seed=None, high_reward_end=None):
        super().__init__(homing=True, seed=seed, high_reward_end=high_reward_end)
```

Appendix B diff_plasticity.py

```
import torch
import torch.nn as nn
from gym_minigrid.wrappers import *
import nn_utils

# environment imports
from t_maze.envs import *

debug = nn_utils.debug

# trace
traces_dir = "../collected_traces"
traces = nn_utils.retrieve_traces(traces_dir, shuffle=True)


# give inputs of all episodes trace
# test in maze
# calculate average
# next maze

# list of mazes and seeds
mazes = []
seeds = []

lstm = None
loss_function = None
optimizer = None
hidden = None
hidden_dim = 3


# train
def train(trace_file, zero_grad=False, zero_hidden=False):
    global hidden

    action_hit = 0
    action_total = 0
    total_loss = 0

    leftright_hit = 0
    leftright_total = 0

    line_count = 0

    with open(trace_file, "r") as fp:
        for line in fp:
            # trace format [x0, x1, x2, x3, x4] action reward
            line_count += 1
            if line.startswith('['):
                if zero_hidden:
                    hidden = torch.FloatTensor([0] * hidden_dim)
                    hidden = torch.reshape(hidden, (1, 1, hidden_dim))
                    hidden = (hidden, hidden)

                # retrieve observation and the desired action
                # from trace file
                image, target_action, lab_reward = nn_utils.parse_line(line)
                nn_input = nn_utils.prepare_lstm_input(image)

                # give the observation to the lstm
                # out should contain the nn's proposed action
                # Forward pass
                out, hidden = lstm(nn_input, hidden)
                hidden[0].detach_()
                hidden[1].detach_()

                # Convert raw scores to probabilities
                softmax_output = torch.softmax(out, dim=2)

                # Reshape action
                action_tensor = torch.LongTensor([target_action])

                # Update parameters
                # optimizer.zero_grad()
                loss = loss_function(out[:, -1].clone(), action_tensor)
                loss.backward()

                if zero_grad:
                    lstm.zero_grad()
                    optimizer.zero_grad()

                _, nn_action = torch.max(out[:, -1], 1)

                action_hit = nn_utils.count_hit(nn_action, target_action,
action_hit)
                action_total += 1

                # measure hit percentage for left/right actions
                if target_action == 0 or target_action == 1:
                    leftright_hit = nn_utils.count_hit(nn_action, target_action,
leftright_hit)
                    leftright_total += 1

                total_loss += loss

                # Print pass info
                if debug:
                    nn_utils.print_pass(line, nn_input, softmax_output,
target_action, nn_action, loss)
            elif line.startswith('MiniGrid-'):  # new episode
                # zero the parameter gradients
                mazes.append(line.strip())
                line = fp.readline()
                seeds.append(int(line.strip()))
                if debug:
                    print('\nTrain:')
                    print(mazes[len(mazes) - 1])
                    print(seeds[len(seeds) - 1])
                    print('########')

    optimizer.step()
    lstm.zero_grad()
    optimizer.zero_grad()

    return action_hit / action_total, leftright_hit / leftright_total,
total_loss.item(), line_count


def test(trace_file):
    global hidden
    mazes = []
    seeds = []
    total_rewards = []

    with open(trace_file, "r") as fp:
        for line in fp:
            if line.startswith('MiniGrid'):
                maze = line.strip()
                seed = int(fp.readline())

                mazes.append(maze)
                seeds.append(seed)

                total_reward = 0

                # number_of_episodes = nn_utils.number_of_episodes(maze, seed)
                # print('Number of Episodes:' + str(number_of_episodes))
                env = gym.make(maze, seed=seed)
                env = FlatObsWrapper(env)

                # set initial observation
                ohe = env.OneHotEncoding()
                nn_input = nn_utils.prepare_lstm_input(ohe)

                while True:
                    # Create a window to view the environment
                    if debug:
                        env.render('human')

                    # Retrieve the agent's action
                    out, hidden = lstm(nn_input, hidden)

                    # transform out probability into a discrete action
                    action = out[:, -1].tolist()[0]
                    action = int(action.index(max(action)))
                    # Give action to environment[
                    _, reward, done, _ = env.step(action)

                    # prepare tensor observation
                    ohe = env.OneHotEncoding()
                    nn_input = nn_utils.prepare_lstm_input(ohe)

                    # sum up reward
                    total_reward += reward

                    if debug:
                        print(ohe)
                        print('probabilities: ' + str(out[:, -1].tolist()[0]))
                        print('action: ' + str(action))

                    if done:
                        if debug:
                            if not env.window.closed:
                                env.window.close()
                        break

                total_rewards.append(total_reward)

    return mazes, seeds, total_rewards


def lvq(trace_file):
    global hidden

    action_hit = 0
    action_total = 0

    with open(trace_file, "r") as fp:
        for line in fp:
            if debug:
                print('Line: ' + line.strip())
            # trace format [x0, x1, x2, x3, x4] action reward
            if line.startswith('['):
                # zero the parameter gradients
                # lstm.zero_grad()
                # retrieve observation and the desired action
                # from trace file
                image, action, _ = nn_utils.parse_line(line)
                nn_input = nn_utils.prepare_lstm_input(image)

                # give the observation to the lstm
                # out should contain the nn's proposed action
                # Forward pass
                out, hidden = lstm(nn_input, hidden)
                hidden[0].detach_()
                hidden[1].detach_()

                _, prediction = torch.max(out[:, -1], 1)
                # if lstm doesn't guess right propagate loss
                if action.item() != prediction.item():
                    loss = loss_function(out[:, -1].clone(), action.clone())
                    loss.backward()
                    optimizer.step()
                    optimizer.zero_grad()

                    if debug:
                        print('image: ' + str(image))
                        print('out: ' + str(out[:, -1]))
                        print('lstm action: ' + str(prediction.item()))
                        print('loss: ' + str(loss.item()))
                        print('########')
                else:
                    action_hit += 1
                action_total += 1

    return action_hit / action_total


def main(learning_rate=None, weight_decay=None, save=None):
    global lstm, loss_function, optimizer, hidden, hidden_dim
    # input dimension
    # 5 - obs
    # 1 - reward
    # output dimension
    # 1 - output step
    input_dim = 6
    hidden_dim = 3
    batch_size = 1
    seq_len = 1
    n_layers = 1

    lr = learning_rate
    wd = weight_decay     # Weight Decay - L2

    # create lstm nn
    lstm = nn.LSTM(input_dim, hidden_dim, n_layers, batch_first=True)

    # hidden represents the reward
    # passed in tuples
    # 0 initial reward
    hidden = torch.FloatTensor([0] * hidden_dim)
    hidden = torch.reshape(hidden, (1, 1, hidden_dim))
    hidden = (hidden, hidden)


    # declare loss function
    # or criterion
    # weights = torch.FloatTensor([1.0, 1.0, 1.0])
    weights = torch.FloatTensor([1 / 6, 1 / 6, 1 / 53])
    loss_function = torch.nn.CrossEntropyLoss(weight=weights)
    optimizer = torch.optim.Adam(lstm.parameters(), lr=lr, weight_decay=wd)
    lstm.zero_grad()
    optimizer.zero_grad()

    train_test_split = 0.8
    train_traces = int(len(traces) * train_test_split) if len(traces) > 4 else
len(traces)
    use_lvq = nn_utils.use_lvq
    find_worst_contenders = nn_utils.use_worst_contenders

    loss_list = []
    hit_list = []
    lr_hit_list = []
    lc_list = []

    print('Number of Traces:' + str(len(traces)))
    print('Traces: ' + str(traces))
    print('Training Traces: ' + str(train_traces))
    print('# ######## #')

    for i in range(train_traces):
        print('Train: ' + traces[i])

        # hit_perc, lr_hit_perc, total_loss = train(traces[i], zero_grad = False,
zero_hidden = False)
        # hit_perc, lr_hit_perc, total_loss = train(traces[i], zero_grad=True,
zero_hidden=False)
        hit_perc, lr_hit_perc, total_loss, line_count = train(traces[i],
zero_grad=False, zero_hidden=True)
        # hit_perc, lr_hit_perc, total_loss = train(traces[i], zero_grad=True,
zero_hidden=True)

        print('hit percentage: ' + str(hit_perc))
        print('left/right hit percentage: ' + str(lr_hit_perc))
        print('total loss: ' + str(total_loss))

        loss_list.append(total_loss)
        hit_list.append(hit_perc)
        lr_hit_list.append(lr_hit_perc)
        lc_list.append(line_count)

    print('##########')

    if use_lvq:
        for i in range(train_traces):
            print('LVQ: ' + traces[i])
            hit_perc = lvq(traces[i])
            print('hit percentage: ' + str(hit_perc))

    if find_worst_contenders:
        nn_utils.find_worst_contenders(loss_list, hit_list, lr_hit_list, traces)

    # create plots
    plot_name = 'LSTM | wd=' + str(wd)
    nn_utils.create_plot(plot_name, lr, hidden_dim, loss_list, hit_list,
lr_hit_list, traces, normalize_data=True, display=False, save=save)

    for i in range(train_traces, len(traces)):
        print('Test: ' + traces[i])
        mazes, seeds, total_rewards = test(traces[i])
        nn_utils.tabulate_print(mazes, seeds, total_rewards)

    return loss_list, hit_list, lr_hit_list, lc_list


if __name__ == '__main__':
    loss_list, hit_list, lr_hit_list, lc_list = main(0.15, 0.001)
```

Appendix C lstm.py

```
import torch
import torch.nn as nn
from gym_minigrid.wrappers import *
import nn_utils

# environment imports
from t_maze.envs import *

debug = nn_utils.debug

# trace
traces_dir = "../collected_traces"
traces = nn_utils.retrieve_traces(traces_dir, shuffle=True)

# give inputs of all episodes trace
# test in maze
# calculate average
# next maze

# list of mazes and seeds
mazes = []
seeds = []

lstm = None
loss_function = None
optimizer = None
hidden = None
hidden_dim = 3


# train
def train(trace_file, zero_grad=False, zero_hidden=False):
    global hidden

    action_hit = 0
    action_total = 0
    total_loss = 0

    leftright_hit = 0
    leftright_total = 0

    line_count = 0

    with open(trace_file, "r") as fp:
        for line in fp:
            # trace format [x0, x1, x2, x3, x4] action reward
            line_count += 1
            if line.startswith('['):
                if zero_hidden:
                    hidden = torch.FloatTensor([0] * hidden_dim)
                    hidden = torch.reshape(hidden, (1, 1, hidden_dim))
                    hidden = (hidden, hidden)

                # retrieve observation and the desired action
                # from trace file
                image, target_action, lab_reward = nn_utils.parse_line(line)
                nn_input = nn_utils.prepare_lstm_input(image)

                # give the observation to the lstm
                # out should contain the nn's proposed action
                # Forward pass
                out, hidden = lstm(nn_input, hidden)
                hidden[0].detach_()
                hidden[1].detach_()

                # Convert raw scores to probabilities
                softmax_output = torch.softmax(out, dim=2)

                # Reshape action
                action_tensor = torch.LongTensor([target_action])

                # Update parameters
                # optimizer.zero_grad()
                loss = loss_function(out[:, -1].clone(), action_tensor)
                loss.backward()

                if zero_grad:
                    lstm.zero_grad()
                    optimizer.zero_grad()

                _, nn_action = torch.max(out[:, -1], 1)

                action_hit = nn_utils.count_hit(nn_action, target_action,
action_hit)
                action_total += 1

                # measure hit percentage for left/right actions
                if target_action == 0 or target_action == 1:
                    leftright_hit = nn_utils.count_hit(nn_action, target_action,
leftright_hit)
                    leftright_total += 1

                total_loss += loss

                # Print pass info
                if debug:
                    nn_utils.print_pass(line, nn_input, softmax_output,
target_action, nn_action, loss)
            elif line.startswith('MiniGrid-'):  # new episode
                # zero the parameter gradients
                mazes.append(line.strip())
                line = fp.readline()
                seeds.append(int(line.strip()))
                if debug:
                    print('\nTrain:')
                    print(mazes[len(mazes) - 1])
                    print(seeds[len(seeds) - 1])
                    print('########')

    optimizer.step()
    lstm.zero_grad()
    optimizer.zero_grad()

    return action_hit / action_total, leftright_hit / leftright_total,
total_loss.item(), line_count


def test(trace_file):
    global hidden
    mazes = []
    seeds = []
    total_rewards = []

    with open(trace_file, "r") as fp:
        for line in fp:
            if line.startswith('MiniGrid'):
                maze = line.strip()
                seed = int(fp.readline())

                mazes.append(maze)
                seeds.append(seed)

                total_reward = 0

                # number_of_episodes = nn_utils.number_of_episodes(maze, seed)
                # print('Number of Episodes:' + str(number_of_episodes))
                env = gym.make(maze, seed=seed)
                env = FlatObsWrapper(env)

                # set initial observation
                ohe = env.OneHotEncoding()
                nn_input = nn_utils.prepare_lstm_input(ohe)

                while True:
                    # Create a window to view the environment
                    if debug:
                        env.render('human')

                    # Retrieve the agent's action
                    out, hidden = lstm(nn_input, hidden)

                    # transform out probability into a discrete action
                    action = out[:, -1].tolist()[0]
                    action = int(action.index(max(action)))

                    # Give action to environment[
                    _, reward, done, _ = env.step(action)

                    # prepare tensor observation
                    ohe = env.OneHotEncoding()
                    nn_input = nn_utils.prepare_lstm_input(ohe)

                    # sum up reward
                    total_reward += reward

                    if debug:
                        print(ohe)
                        print('probabilities: ' + str(out[:, -1].tolist()[0]))
                        print('action: ' + str(action))

                    if done:
                        if debug:
                            if not env.window.closed:
                                env.window.close()
                        break

                total_rewards.append(total_reward)

    return mazes, seeds, total_rewards


def lvq(trace_file):
    global hidden

    action_hit = 0
    action_total = 0

    with open(trace_file, "r") as fp:
        for line in fp:
            if debug:
                print('Line: ' + line.strip())
            # trace format [x0, x1, x2, x3, x4] action reward
            if line.startswith('['):
                # zero the parameter gradients
                # lstm.zero_grad()
                # retrieve observation and the desired action
                # from trace file
                image, action, _ = nn_utils.parse_line(line)
                nn_input = nn_utils.prepare_lstm_input(image)

                # give the observation to the lstm
                # out should contain the nn's proposed action
                # Forward pass
                out, hidden = lstm(nn_input, hidden)
                hidden[0].detach_()
                hidden[1].detach_()

                _, prediction = torch.max(out[:, -1], 1)
                # if lstm doesn't guess right propagate loss
                if action.item() != prediction.item():
                    loss = loss_function(out[:, -1].clone(), action.clone())
                    loss.backward()
                    optimizer.step()
                    optimizer.zero_grad()

                    if debug:
                        print('image: ' + str(image))
                        print('out: ' + str(out[:, -1]))
                        print('lstm action: ' + str(prediction.item()))
                        print('loss: ' + str(loss.item()))
                        print('########')
                else:
                    action_hit += 1
                action_total += 1

    return action_hit / action_total


def main(learning_rate=None, weight_decay=None, save=None):
    global lstm, loss_function, optimizer, hidden, hidden_dim
    # input dimension
    # 5 - obs
    # 1 - reward
    # output dimension
    # 1 - output step
    input_dim = 6
    hidden_dim = 3
    batch_size = 1
    seq_len = 1
    n_layers = 1

    lr = learning_rate
    wd = weight_decay     # Weight Decay - L2

    # create lstm nn
    lstm = nn.LSTM(input_dim, hidden_dim, n_layers, batch_first=True)

    # hidden represents the reward
    # passed in tuples
    # 0 initial reward
    hidden = torch.FloatTensor([0] * hidden_dim)
    hidden = torch.reshape(hidden, (1, 1, hidden_dim))
    hidden = (hidden, hidden)


    # declare loss function
    # or criterion
    # weights = torch.FloatTensor([1.0, 1.0, 1.0])
    weights = torch.FloatTensor([1 / 6, 1 / 6, 1 / 53])
    loss_function = torch.nn.CrossEntropyLoss(weight=weights)
    optimizer = torch.optim.Adam(lstm.parameters(), lr=lr, weight_decay=wd)
    lstm.zero_grad()
    optimizer.zero_grad()

    train_test_split = 0.8
    train_traces = int(len(traces) * train_test_split) if len(traces) > 4 else
len(traces)
    use_lvq = nn_utils.use_lvq
    find_worst_contenders = nn_utils.use_worst_contenders

    loss_list = []
    hit_list = []
    lr_hit_list = []
    lc_list = []

    print('Number of Traces:' + str(len(traces)))
    print('Traces: ' + str(traces))
    print('Training Traces: ' + str(train_traces))
    print('# ######## #')

    for i in range(train_traces):
        print('Train: ' + traces[i])

        # hit_perc, lr_hit_perc, total_loss = train(traces[i], zero_grad = False,
zero_hidden = False)
        # hit_perc, lr_hit_perc, total_loss = train(traces[i], zero_grad=True,
zero_hidden=False)
        hit_perc, lr_hit_perc, total_loss, line_count = train(traces[i],
zero_grad=False, zero_hidden=True)
        # hit_perc, lr_hit_perc, total_loss = train(traces[i], zero_grad=True,
zero_hidden=True)

        print('hit percentage: ' + str(hit_perc))
        print('left/right hit percentage: ' + str(lr_hit_perc))
        print('total loss: ' + str(total_loss))

        loss_list.append(total_loss)
        hit_list.append(hit_perc)
        lr_hit_list.append(lr_hit_perc)
        lc_list.append(line_count)

    print('##########')

    if use_lvq:
        for i in range(train_traces):
            print('LVQ: ' + traces[i])
            hit_perc = lvq(traces[i])
            print('hit percentage: ' + str(hit_perc))

    if find_worst_contenders:
        nn_utils.find_worst_contenders(loss_list, hit_list, lr_hit_list, traces)

    # create plots
    plot_name = 'LSTM | wd=' + str(wd)
    nn_utils.create_plot(plot_name, lr, hidden_dim, loss_list, hit_list,
lr_hit_list, traces, normalize_data=True, display=False, save=save)

    for i in range(train_traces, len(traces)):
        print('Test: ' + traces[i])
        mazes, seeds, total_rewards = test(traces[i])
        nn_utils.tabulate_print(mazes, seeds, total_rewards)

    return loss_list, hit_list, lr_hit_list, lc_list


if __name__ == '__main__':
    loss_list, hit_list, lr_hit_list, lc_list = main(0.15, 0.001)
```

Appendix D nn_utils.py

```
import ast
import glob
import random
import sys

import torch
import numpy
from matplotlib import pyplot as plt
from torch.autograd import Variable

DISCRETE_ACTIONS = 3

debug = False
traces_dir = "../collected_traces"
results_folder = 'results/best_results/'

use_lvq = False
use_worst_contenders = False

environments = ['MiniGrid-TMaze-v0',
                    'MiniGrid-DoubleTMaze-v0',
                    'MiniGrid-TripleTMaze-v0',
                    'MiniGrid-QuadTMaze-v0',
                    'MiniGrid-TMazeHoming-v0',
                    'MiniGrid-DoubleTMazeHoming-v0',
                    'MiniGrid-TripleTMazeHoming-v0',
                    'MiniGrid-QuadTMazeHoming-v0']

def number_of_episodes(maze, seed):
    maze_ends = [2, 4, 8, 16,
                 2, 4, 8, 16]

    index = environments.index(maze)

    # initialize task random function
    task_rand = random.Random()
    task_rand.seed(seed)

    episodes = -1

    # calculate how many episodes the task will be played
    if maze_ends[index] == 2:
        episodes = task_rand.randint(4, 8)
    elif maze_ends[index] == 4:
        episodes = task_rand.randint(8, 12)
    elif maze_ends[index] == 8:
        episodes = task_rand.randint(16, 20)
    elif maze_ends[index] == 16:
        episodes = task_rand.randint(32, 36)

    return episodes
```

```
def encode_one_hot(value, max):
    encoding = [0] * max
    encoding[value] = 1

    return encoding


def retrieve_traces(directory, shuffle=False):
    traces = glob.glob(directory + "/*.txt")
    if shuffle:
        random.shuffle(traces)
    return traces


def parse_line(line):
    line = line.strip()
    split = line.split(']')
    image = ast.literal_eval(split[0] + ']')

    split = split[1].split()
    action = int(split[0])
    reward = float(split[1])

    action = torch.tensor([action])

    return image, action, reward


# Input Format
# observation - image
# action
# reward
# bias
def prepare_dp_input(image, action, reward):
    bias = 1.0

    inputs = image
    inputs.append(bias)

    # Convert input to Numpy array
    inputs = numpy.array(inputs)

    return torch.FloatTensor(inputs)


# Input Format
# observation - image
# action
# reward
# bias
def prepare_lstm_input(image):
    bias = 1.0
    image.append(bias)

    tensor_len = len(image)

    # Reshape image and action into tensors
    input = torch.FloatTensor(image)
    input = torch.reshape(input, (1, 1, tensor_len))

    return input


def print_pass(line, nn_input, softmax_output, target_action, prediction, loss):
    print('Line:\t\t' + line.strip())
    print('NN input:\t' + str(nn_input))
    print('NN output:\t' + str(softmax_output))
    print('\tNN action:\t\t' + str(prediction))
    print('\tTarget action:\t' + str(target_action.item()))

    print('Loss: ' + str(loss.item()))
    print('########')


def count_hit(output, target, counter):
    if torch.is_tensor(output):
        output = output.item()
    if torch.is_tensor(target):
        target = target.item()

    return counter + 1 if output == target else counter


def create_plot(name, lr, hidden_layer_size, loss_list, hit_list, lr_hit_list,
traces=None, normalize_data=False, display=False, save=None):
    def file_len(fname):
        with open(fname) as f:
            for i, l in enumerate(f):
                pass
        return i + 1

    normalized_loss = [0] * len(loss_list)

    if traces is not None and normalize_data:
        name += ' | Normalized Loss'
        sum_len = 0
        for i in range(len(loss_list)):
            sum_len += file_len(traces[i])

        avg_len = sum_len / len(loss_list)
        for i in range(len(loss_list)):
            normalized_loss[i] = loss_list[i] * (avg_len / file_len(traces[i]))

        loss_list = normalized_loss
    # Create plot
    fig, ax = plt.subplots()
    fig_title = name + ' | lr = ' + str(lr) + " | hidden_layer_size = " +
str(hidden_layer_size)
    fig.suptitle(fig_title)
    ax.plot(loss_list, label="loss", marker='o', color='blue')
    ax.tick_params(axis='y', labelcolor='blue')
    ax2 = ax.twinx()
    ax2.plot(hit_list, label="total hit percentage", marker='o', color='red')
    # ax2.plot(normalized_loss, label='Normalized Loss', color='tab:purple')
    plt.plot(lr_hit_list, label="left/right hit percentage", marker='o',
color='green')
    ax.set_ylim(bottom=0)
    ax2.set_ylim([0.0, 1.0])

    ax.set(xlabel="Epochs", ylabel="Loss")

    plt.ylabel("hit_percentage")

    ax.legend()
    ax2.legend()

    if save is not None:
        plt.savefig(results_folder + save + fig_title + '.png')

    if display:
        plt.show()


def find_worst_contenders(loss_list, hit_perc, lr_hit_perc, traces):
    loss = 0
    worst_loss = ''
    lr = sys.float_info.max
    worst_lr = ''
    hit = sys.float_info.max
    worst_hit = ''

    for i in range(len(loss_list)):
        if loss_list[i] > loss:
            loss = loss_list[i]
            worst_loss = traces[i]
        if lr_hit_perc[i] < lr:
            lr = lr_hit_perc[i]
            worst_lr = traces[i]
        if hit_perc[i] < hit:
            hit = hit_perc[i]
            worst_hit = traces[i]

    print('######\n')

    print('worst_loss: ' + worst_loss)
    print('worst_hit: ' + worst_hit)
    print('worst_lr: ' + worst_lr)

    print('######\n')

def tabulate_print(mazes, seeds, total_rewards):
    average_reward = [0] * len(environments)
    reward_counter = [0] * len(environments)

    for i in range(len(mazes)):
        for j in range(len(environments)):
            if mazes[i] == environments[j]:
                average_reward[j] += total_rewards[i]
                reward_counter[j] += 1

    with open(results_folder + 'average.txt', 'w') as f:
        for i in range(len(average_reward)):
            average_reward[i] = average_reward[i] / reward_counter[i]

            print(environments[i] + '\t' + str(average_reward[i]), file=f)

if __name__ == '__main__':
    traces = retrieve_traces(traces_dir)

    classes = [0] * 3

    for trace_file in traces:
        with open(trace_file, "r") as fp:
            for line in fp:
                if line.startswith('['):
                    _, target_action, _ = parse_line(line)

                    classes[target_action] += 1

    print(classes)
```

Appendix E manual_control.py

```
"""
This code was taken from gym-minigrid. It is need for manually controlling the
agent in the t-maze environment.
Some changes on the file were necessary so I considered it useful to exist here as
well.

Manual control: python manual_control.py --env MiniGrid-TMaze-v0
"""

import argparse

# environment imports
from t_maze.envs import *

from trace import *
from gym_minigrid.wrappers import *
from window import Window


# Environment Parameters
args = None
env = None
window = None

trace_file = None

reward_episode = False       # has the goal been collected
                             # if it has allow one more episode just to see the
reward collected

def redraw(img):
    if not args.agent_view:
        img = env.render('rgb_array', tile_size=args.tile_size)

    window.show_img(img)


def reset():
    init_episode(trace_file, args.env, args.seed)


    # one more step, allows the user to see what reward he has picked
    if reward_episode:
        reward_episode = False
        window.close()
        finalize_episode(trace_file)
    else:
        # retrieve the one hot encoding observation of the current tile
        ohe = env.OneHotEncoding()

        # send agent step to environment
        obs, reward, done, info = env.step(action)

        # update ui info
        window.set_caption(reward)
        window.update_steps()

        if reward != 0:
            log_step(trace_file, str(ohe), action.value, reward)

        if done:
            reward_episode = True

        redraw(obs)


def key_handler(event):
    # print('pressed', event.key)

    if event.key == 'escape':
        window.close()
        return

    if event.key == 'backspace':
        trace_file.write("reset\n")
        reset()
        return

    if event.key == 'left':
        step(env.actions.left)
        return
    if event.key == 'right':
        step(env.actions.right)
        return
    if event.key == 'up':
        step(env.actions.forward)
        return

    # Spacebar
    # if event.key == ' ':
    #     step(env.actions.toggle)
    #     return
    # if event.key == 'pageup':
    #     step(env.actions.pickup)
    #     return
    # if event.key == 'pagedown':
    #     step(env.actions.drop)
    #     return
    #
    # if event.key == 'enter':
    #     step(env.actions.done)
    #     return
```

```
def load_env(env_details, file, task_no, max_task, episode_no, max_episode):
    global args, env, window, trace_file
    trace_file = file

    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--env",
        help="gym environment to load",
        default='MiniGrid-TMaze-v0'
    )
    parser.add_argument(
        "--seed",
        type=int,
        help="random seed to generate the environment with",
        default=1
    )
    parser.add_argument(
        "--end",
        type=int,
        help="which corridor end the reward is placed",
        default=None
    )

    parser.add_argument(
        "--tile_size",
        type=int,
        help="which corridor end the reward is placed",
        default=32
    )
    parser.add_argument(
        '--agent_view',
        default=False,
        help="draw the agent sees (partially observable view)",
        action='store_true'
    )

    args = parser.parse_args(env_details)
    # print(args)
    if 'TMaze' in args.env:
        env = gym.make(args.env, seed=args.seed, high_reward_end=args.end)
        env.seed(args.seed)
    else:
        env = gym.make(args.env)
        env.seed(args.seed)

    if args.agent_view:
        # env = RGBImgObsWrapper(env)
        env = RGBImgPartialObsWrapper(env, 140)
        env = ImgObsWrapper(env)
    else:
        env = RGBImgObsWrapper(env, 70)

    window = Window('gym_minigrid - ' + args.env, env, task_no, max_task,
episode_no, max_episode)
    window.reg_key_handler(key_handler)

    global reward_episode
    reward_episode = False

    reset()

    # Blocking event loop
    window.show(block=True)
    obs = env.reset()
    redraw(obs)


def step(action):
    global reward_episode


    # one more step, allows the user to see what reward he has picked
    if reward_episode:
        reward_episode = False
        window.close()
        finalize_episode(trace_file)
    else:
        # retrieve the one hot encoding observation of the current tile
        ohe = env.OneHotEncoding()

        # send agent step to environment
        obs, reward, done, info = env.step(action)

        # update ui info
        window.set_caption(reward)
        window.update_steps()

        if reward != 0:
            log_step(trace_file, str(ohe), action.value, reward)

        if done:
            reward_episode = True

        redraw(obs)


def key_handler(event):
    # print('pressed', event.key)

    if event.key == 'escape':
        window.close()
        return

    if event.key == 'backspace':
        trace_file.write("reset\n")
        reset()
        return

    if event.key == 'left':
        step(env.actions.left)
        return
    if event.key == 'right':
        step(env.actions.right)
        return
    if event.key == 'up':
        step(env.actions.forward)
        return

    # Spacebar
    # if event.key == ' ':
    #     step(env.actions.toggle)
    #     return
    # if event.key == 'pageup':
    #     step(env.actions.pickup)
    #     return
    # if event.key == 'pagedown':
    #     step(env.actions.drop)
    #     return
    #
    # if event.key == 'enter':
    #     step(env.actions.done)
    #     return


def load_env(env_details, file, task_no, max_task, episode_no, max_episode):
    global args, env, window, trace_file
    trace_file = file

    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--env",
        help="gym environment to load",
        default='MiniGrid-TMaze-v0'
    )
    parser.add_argument(
        "--seed",
        type=int,
        help="random seed to generate the environment with",
        default=1
    )
    parser.add_argument(
        "--end",
        type=int,
        help="which corridor end the reward is placed",
        default=None
    )

    parser.add_argument(
        "--tile_size",
        type=int,
        help="which corridor end the reward is placed",
        default=32
    )
    parser.add_argument(
        '--agent_view',
        default=False,
        help="draw the agent sees (partially observable view)",
        action='store_true'
    )

    args = parser.parse_args(env_details)
    # print(args)
    if 'TMaze' in args.env:
        env = gym.make(args.env, seed=args.seed, high_reward_end=args.end)
        env.seed(args.seed)
    else:
        env = gym.make(args.env)
        env.seed(args.seed)

    if args.agent_view:
        # env = RGBImgObsWrapper(env)
        env = RGBImgPartialObsWrapper(env, 140)
        env = ImgObsWrapper(env)
    else:
        env = RGBImgObsWrapper(env, 70)

    window = Window('gym_minigrid - ' + args.env, env, task_no, max_task,
episode_no, max_episode)
    window.reg_key_handler(key_handler)

    global reward_episode
    reward_episode = False

    reset()

    # Blocking event loop
    window.show(block=True)
```

Appendix F epoch.py

```
import random
import manual_control
from trace import *

environments = ['MiniGrid-TMaze-v0',
                'MiniGrid-DoubleTMaze-v0',
                'MiniGrid-TripleTMaze-v0',
                'MiniGrid-QuadTMaze-v0',
                'MiniGrid-TMazeHoming-v0',
                'MiniGrid-DoubleTMazeHoming-v0',
                'MiniGrid-TripleTMazeHoming-v0',
                'MiniGrid-QuadTMazeHoming-v0']

task_order = [0, 1, 2, 3,
              4, 5, 6, 7]
maze_ends = [2, 4, 8, 16,
             2, 4, 8, 16]
agent_view = [False, False, False, False,
              False, False, False, False]


# prepares the arguments to be passed on to the maze
def prepare_args(env, seed, agent_view, reward_end):
    if agent_view:
        args = ["--env", env, "--seed", seed, "--agent_view", "--end", reward_end]
    else:
        args = ["--env", env, "--seed", seed, "--end", reward_end]

    return args


def main():
    # initialize trace file and random function
    trace_file = initialize_trace()
    main_rand = random.Random()

    for i in range(len(task_order)):
        # create seed for task
        seed = str(main_rand.randint(0, 20000))

        # initialize task random function
        task_rand = random.Random()
        task_rand.seed(seed)

        episodes = -1

        # calculate how many episodes the task will be played
        if maze_ends[i] == 2:
            episodes = task_rand.randint(4, 8)
        elif maze_ends[i] == 4:
            episodes = task_rand.randint(8, 12)
        elif maze_ends[i] == 8:
            episodes = task_rand.randint(16, 20)
        elif maze_ends[i] == 16:
            episodes = task_rand.randint(32, 36)
        # print('episodes=' + str(episodes))

        # retrieve task environment
        env = environments[task_order[i]]

        # calculate when the reward end will change
        end_change = task_rand.randint(1, episodes - 1)
        reward_end = str(task_rand.randint(0, maze_ends[i]-1))
        # print('end_change=' + str(end_change))

        # prepare the arguments for the task
        args = prepare_args(env, seed, agent_view[i], reward_end)

        # start the task
        for j in range(end_change):
            manual_control.load_env(args, trace_file, j, episodes, i,
len(task_order))

        # generate a new reward end
        second_reward_end = reward_end
        while second_reward_end == reward_end:
            second_reward_end = str(task_rand.randint(0, maze_ends[i]-1))

        # prepare the arguments for the task
        args = prepare_args(env, seed, agent_view[i], second_reward_end)

        # start the task with a changed reward location
        for j in range(end_change, episodes):
            manual_control.load_env(args, trace_file, j, episodes, i,
len(task_order))

    # finalize trace file
    finalize_trace(trace_file)


if __name__ == '__main__':
    main()
```

Appendix G trace.py

```
import os
import time

path = "trace"

# Initialize trace at start of epoch
def initialize_trace():
    if not os.path.exists(path):
        os.makedirs(path)

    timestamp = time.strftime("%Y%m%d-%H%M.txt")
    filepath = os.path.join(path, timestamp)
    file = open(filepath, "w")

    return file


# Initialize trace at start of episode
def init_episode(file, env, seed):
    file.write(env + '\n')
    file.write(str(seed) + '\n')


# Finalizes episode
def finalize_episode(file):
    file.write("done\n")


# Logs action, the observation and the reward of a single timestep
def log_step(file, obs, action, reward):
    file.write(str(obs) + '\t' + str(action) + '\t' + str(reward) + '\n')


# Close the trace
def finalize_trace(file):
    file.write('trace complete\n')
    file.close()
```