The Imitation Game

In this project, we attempt to train and agent (rather than implementing a specific decision agent) to imitate the agent's we've build previously. From each of Project 1 and Project 2, we take Agent 1 and Agent 3 respectively, and build an ML agent to mimic it in the following way:

- Take the input data to be state descriptions of the current knowledge of the gridworld.
- Take the output data to be the corresponding actions the agent takes in the input states.
- The ML agent should consist of a neural network that maps from the input space to the output space (or, as necessary, modified, i.e., for softmax/probability regression).
- Train the model on data generated by Agent 1 and Agent 3, over many episodes and many gridworlds.

After sufficient training, the ML agent should be able to simulate the appropriate gridworld and use the trained network to make action selections given the current state. Repeatedly simulating the original agent and the ML agent on identical gridworlds will allow a comparison of performance between the two.

1) **How should the state space (current information) and action space (action selected) be represented for the model? How does it capture the relevant information, in a relevant way, for your model space? One thing to consider here is local vs global information.**

We have built a classification model to solve this project to mimic both the Agents. As we know that a neural network-based classification model must have a state space and action space i.e., we expect to pass our current information into our model and receive a prediction regarding the best possible action. So, for this we have considered different information for both the agents.

a.) To mimic an agent from project 1, we have selected the Agent 1 for this task. The Agent 1 has the power to sense whether its neighboring cells are blocked or unblocked. This information about the neighboring cells then gets updated in the agent's knowledge base. This knowledge base is then reflected in the discovered grid according to which our agent plans its path. So, for this agent's state space we have taken a 5x5 window of our discovered grid for every step that the agent takes. This window is always formed such that the agent's current cell is in the center (if the agent is in a corner cell, then we have taken the outer board cells as blocked cells as they were serving the same purpose). For all the cells in this window we have basically divided our grid world into 5 categories.

   i.) Current position of the agent (0) – This is the current position of the agent.
   ii.) Blocked cells (-1) – This is if the cell is blocked.
   iii.) Visited cells (2) – This is if the cell has been visited by the agent.
   iv.) Discovered cells (1) – This is if the cell is unvisited but sensed by the agent.
   v.) Undiscovered cells (3) – This is if the cell is both unvisited and unsensed by the agent.

   We have taken the action space as the direction our agent moves based on all the information that it has up to that point. As this agent is only allowed to move in 4

directions that is up, down, right, left; we have taken these four directions as our action space.

Afterwards we have flattened these 5x5 windows and put the output direction next to it to make our dataset csv.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	3	1	-1	-1	-1	1	1	-1	-1	1	1	1	right
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	0	3	1	-1	-1	-1	1	1	-1	1	1	1	1	right
2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	2	0	3	1	-1	-1	3	1	1	1	1	1	1	1	right
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	2	0	3	1	-1	3	-1	1	1	1	1	1	1	1	right
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	2	0	3	1	3	-1	3	1	1	1	1	1	1	1	right

Dataset.head() for agent 1

b.) To mimic an agent from project 2, we have selected the Agent 3 for this task. The Agent 3 has the power to sense the total number of blocked and empty cells in its vicinity, but it doesn't exactly know which of its neighboring cells is which until it actually bumps into one of the blocks. Using this knowledge this agent tries to infer which cells could be definitely blocked and which could be definitely unblocked. Then it updates these into its knowledge base. This knowledge base is then reflected in the discovered grid according to which our agent plans its path. So, for this agent's state space we have taken a 5x5 window of our discovered grid for every step that the agent takes as well. This window is always formed such that the agent's current cell is in the center (if the agent is in a corner cell, then we have taken the outer board cells as blocked cells as they were serving the same purpose). For all the cells in this window we have basically divided our grid world into 5 categories.

i.) Current position of the agent (0) – This is the current position of the agent.
ii.) Blocked cells (-1) – This is if the cell is blocked.
iii.) Visited cells (2) – This is if the cell has been visited by the agent.
iv.) Discovered cells (1) – This is if the cell is unvisited but sensed by the agent.
v.) Undiscovered cells (3) – This is if the cell is both unvisited and unsensed by the agent.

For this agent we have also taken the number of neighboring cells that the agent has sensed as blocked as input.

We have taken the action space as the direction our agent moves based on all the information that it has up to that point. As this agent is only allowed to move in 4 directions that is up, down, right, left; we have taken these four directions as our action space.

Afterwards we have flattened these 5x5 windows and put the number of sensed blocked cells and the output direction next to it to make our dataset csv.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	1	-1	-1	3	3	1	-1	-1	1	1	1	1	down
1	-1	-1	-1	-1	-1	-1	-1	2	-1	1	-1	-1	0	3	1	-1	-1	3	3	1	-1	-1	1	1	1	1	right
2	-1	-1	-1	-1	-1	-1	2	-1	3	1	-1	2	0	3	1	-1	3	3	3	1	-1	1	1	1	1	1	right
3	-1	-1	-1	-1	-1	2	-1	3	1	1	2	2	0	1	1	3	3	3	1	1	1	1	1	1	1	3	right
4	-1	-1	-1	-1	-1	-1	3	-1	1	1	2	2	0	1	1	3	3	-1	1	1	1	1	1	1	1	2	right

Dataset.head() for agent 3

We have chosen these fields as our state space and action space as we felt that this is all the information that our original agents had to make the decision of where to go next. As can be seen we have used a local information approach. We also tried the global information approach where we gave the full discovered grid world to our model as input to predict the direction of where to next but that didn't really give any appreciable results as we were just giving too much information to our model. We feel that by doing so we increased the number of parameters for our model to train by such a big number that our model couldn't really comprehend what to do with all that information. So, it was giving us really bad results. Therefore, we switched to local information which in turn gave us amazing results.

2) How are you defining your loss function when training your model?

Training process requires defining a loss function and an optimization algorithm. Importantly, the choice of loss function is directly related to the activation function used in the outer layer of our neural network. These two design elements are connected.

Few of the common loss functions include:

Loss Functions	***Problem Types***
BCELoss	*Binary cross-entropy loss for binary classification*
CrossEntropyLoss	*Categorical cross-entropy loss for multi-class classification*
MSELoss	*Mean Squared Loss for regression*

Table 1.2 Different Loss Functions for problem types

Now, we know from our state space and action space that our problem is of type *multi-class classification*, wherein we need to train our model to predict one of the four correct actions (movement in up, down, right, or left direction) given state space at time t. This clarifies that we need ***CrossEntropyLoss*** function and a *SoftMax* activation function in the outer layer.

Cross-Entropy Loss (or Log Loss):

Each predicted probability is compared to actual class output value (0 or 1) and a score is calculated that penalizes the probability based on the distance from the expected value. The penalty is logarithmic, offering a small score for small differences (0.1 or 0.2) and enormous score for a large difference (0.9 0r 1.0).

Cross-Entropy Loss is minimized, where smaller values represent a better model than larger values.

3) **In training, how many episodes on how many different grid worlds were necessary to get good performance of your model on the training data?**

In training, we have taken 10-15 (compared models trained on both 10 and 15 epochs and then chosen the optimal one) episodes of 1000 different grid worlds for both agent 1 and 3 to get a good performance. We have generated 1000 different grid worlds of size 50x50 and then for each grid world we had stored the discovered grid of the original agent for each step in a csv file. As our agent takes many steps to solve a grid, we have gotten a dataset of more than 190,000 rows for each agent.

4) **How did you avoid overfitting? Since you want the ML agent to mimic the original agent, should you avoid overfitting?**

We have avoided overfitting by taking a huge dataset of more than 190,000 entries for both the agents so that the model is able to detect the signal better. We also closely kept a track of the training and the test accuracy. Since they always came pretty close to each other for all our models, we know for sure that our models are not overfitted. Furthermore, we stopped our training early as soon as our training accuracy got a bit constant and wasn't increasing that much.

Even though we want our ML agents to mimic the original agents, we should avoid overfitting as we're ultimately going to generate new mazes and try to solve these mazes using our models. These mazes wouldn't have been seen by our agents during training, so we want our agents to be prepared for this, we want our agent to be prepared the real world, for the world that it hasn't yet seen before. If we overfit in an attempt to mimic our original agents, our model won't be able to solve unseen new mazes with the same efficiency. Our ML models are trained on the data generated by the original agents, but the thing to keep in mind here is that this is just a small fraction of the possible data that can be generated by our original agents. So overfitting our models on this small fraction of data doesn't seem like a good idea.

5) **How did you explore the architecture space, and test the different possibilities to find the best architecture?**

We trained both our agent's models on 2 types of architectures:

a. Feedforward Neural Network
b. Convolutional Neural Network

We trained a total of 12 models for each agent, consisting of both the architectures experimenting with number of layers, nodes per layer, activation functions, window sizes, strides, filters, etc, a summary of which is presented in the table below.

<u>Agent 1</u>**:**

a. Feedforward Neural Network:

model 1: (Base Model) name: My5b5_Ag1_9867_model layers: 25, 16, 4 learning_rate: 0.0025 epochs: 10

batch_size: 128 Accuracy: 98.67 Inference: Base Model
model 2: (Adding hidden layer) name: My5b5_Ag1_9863_model layers: 25, 16, 8, 4 learning_rate: 0.0025 epochs: 10 batch_size: 128 Accuracy: 98.63 Inference: Adding a dense layer has a negative effect on the accuracy, since it increases the model complexity and tries to overfit.
model 3: (Increasing epochs) name: My5b5_Ag1_9878_model layers: 25, 16, 8, 4 learning_rate: 0.0025 epochs: 15 batch_size: 128 Accuracy: 98.78 Inference: Increasing the number of epochs has a positive effect on the accuracy, since it increases the time and data for the model to learn better.
model 4: (Decreasing learning rate) name: My5b5_Ag1_9854_model layers: 25, 16, 4 learning_rate: 0.0005 epochs: 15 batch_size: 128 Accuracy: 98.54 Inference: Decreasing the learning rate has a negative effect on the accuracy, since it takes too long to get the optimal SGD.
model 5: (Increasing learning rate) name: My5b5_Ag1_9875_model layers: 25, 16, 4 learning_rate: 0.005 epochs: 15 batch_size: 128 Accuracy: 98.75 Inference: Increasing the learning rate has a positive effect on the accuracy, since it takes bigger steps and is able to reach the optimal SGD sooner.
model 6: (Increasing batch size) name: My5b5_Ag1_9879_model layers: 25, 16, 8, 4 learning_rate: 0.005 epochs: 15 batch_size: 256 Accuracy: 98.79 Inference: Increasing the batch size doesn't have any effect on the accuracy in this case.
model 7: (Decreasing batch size) name: My5b5_Ag1_9885_model layers: 25, 16, 4

learning_rate: 0.005 epochs: 15 batch_size: 64 Accuracy: 98.85 Inference: Decreasing the batch size has a positive effect on the accuracy in this case.

b. Convolutional Neural Network:

model 8: (Base Convolution model) name: My_conv9889_model layers: Conv2D(16, (2,2), valid padding, input_dim = (5,5,1)), Flatten, Dense(8), Dense(4) learning_rate: 0.005 epochs: 15 batch_size: 64 Accuracy: 98.89 Inference: Base Model
model 9: (Increasing output dim of conv layer) name: My_conv9913_model layers: Conv2D(32, (2,2), valid padding, input_dim = (5,5,1)), Flatten, Dense(16), Dense(4) learning_rate: 0.005 epochs: 15 batch_size: 64 Accuracy: 99.13 Inference: Increasing the output dim of conv layer has a positive effect on the accuracy as it gives more features for the model to learn.
model 10: (Adding a dense layer) name: My_conv9908_model layers: Conv2D(32, (2,2), valid padding, input_dim = (5,5,1)), Flatten, Dense(16), Dense(8),Dense(4) learning_rate: 0.005 epochs: 15 batch_size: 64 Accuracy: 99.08 Inference: Adding a dense layer has a negative effect on the accuracy, since it increases the model complexity and tries to overfit.
model 11: (Decreasing kernel size) name: My_conv9899_model layers: Conv2D(32, (1,1), valid padding, input_dim = (5,5,1)), Flatten, Dense(16), Dense(4) learning_rate: 0.005 epochs: 15 batch_size: 64 Accuracy: 98.99 Inference: Decreasing the kernel size has a negative effect on the accuracy, since it is able to learn less features of the data.
model 12: (Increasing conv layers) name: My_conv9917_model

layers: Conv2D(64, (2,2), valid padding, input_dim = (5,5,1)),Conv(32, (2,2), valid padding, input_dim = (8,8,1)) Flatten, Dense(16), Dense(4) learning_rate: 0.005 epochs: 15 batch_size: 64 Accuracy: 99.17 Inference: Increasing the conv layers have a positive effect on the accuracy, since it is able to learn more features of the data.

Agent 3:

a. Feedforward Neural Network:

model 1: (Base Model) name: My5b5_Ag3_9830_model layers: 25, 16, 4 learning_rate: 0.0025 epochs: 10 batch_size: 128 Accuracy: 98.30 Inference: Base Model
model 2: (Adding hidden layer) name: My5b5_Ag3_9843_model layers: 25, 16, 8, 4 learning_rate: 0.0025 epochs: 10 batch_size: 128 Accuracy: 98.43 Inference: Adding a dense layer has a positive effect on the accuracy, since it increases the faetures for the model to learn.
model 3: (Increasing epochs) name: My5b5_Ag3_9829_model layers: 25, 16, 8, 4 learning_rate: 0.0025 epochs: 15 batch_size: 128 Accuracy: 98.29 Inference: Increasing the number of epochs has a negative effect on the accuracy, since it tries to overfit.
model 4: (Decreasing learning rate) name: My5b5_Ag3_9763_model layers: 25, 16, 8, 4 learning_rate: 0.0005 epochs: 10 batch_size: 128 Accuracy: 97.63 Inference: Decreasing the learning rate has a negative effect on the accuracy, since it takes too long to get the optimal SGD.
model 5: (Increasing learning rate) name: My5b5_Ag3_9814_model

layers: 25, 16, 8, 4 learning_rate: 0.005 epochs: 10 batch_size: 128 Accuracy: 98.14 Inference: Increasing the learning rate has a negative effect on the accuracy, since it takes bigger steps and therefore jumps over the optimal SGD sooner.
model 6: (Increasing batch size) name: My5b5_Ag3_9801_model layers: 25, 16, 8, 4 learning_rate: 0.0025 epochs: 10 batch_size: 256 Accuracy: 98.01 Inference: Increasing the batch size has a negative effect on the accuracy in this case.
model 7: (Decreasing batch size) name: My5b5_Ag3_9835_model layers: 25, 16, 4 learning_rate: 0.0025 epochs: 10 batch_size: 64 Accuracy: 98.35 Inference: Decreasing the batch size has a negative effect on the accuracy in this case.

b. Convolution Neural Network:

model 8: (Base Convolution model) name: My5b5_conv9872_model layers: Conv2D(32, (2,2), valid padding, input_dim = (13,2,1)), Flatten, Dense(16), Dense(4) learning_rate: 0.0025 epochs: 10 batch_size: 128 Accuracy: 98.72 Inference: Base Model
model 9: (Increasing output dim of conv layer) name: My5b5_conv9867_model layers: Conv2D(64, (2,2), valid padding, input_dim = (13,2,1)), Flatten, Dense(16), Dense(4) learning_rate: 0.0025 epochs: 10 batch_size: 128 Accuracy: 98.67 Inference: Increasing the output dim of the conv layer has a negative effect on the accuracy, since it increases the model complexity and tries to overfit.
model 10: (Adding a dense layer) name: My5b5_conv9875_model layers: Conv2D(32, (2,2), valid padding, input_dim = (13,2,1)), Flatten, Dense(16), Dense(8), Dense(4)

learning_rate: 0.0025 epochs: 10 batch_size: 128 Accuracy: 98.75 Inference: Adding a dense layer has a positive impact on the model as the model get more features to learn from.
model 11: (Decreasing kernel size) name: My5b5_conv9859_model layers: Conv2D(32, (1,1), valid padding, input_dim = (13,2,1)), Flatten, Dense(16), Dense(8), Dense(4) learning_rate: 0.0025 epochs: 10 batch_size: 128 Accuracy: 98.59 Inference: Decreasing kernel size has a negative impact on the model, since it increases the model complexity and tries to overfit.
model 12: (Increasing conv layers) name: My5b5_conv9884_model layers: Conv2D(64, (2,2), valid padding, input_dim = (13,2,1)),Conv(32, (2,2), valid padding), Flatten, Dense(16), Dense(4) learning_rate: 0.0025 epochs: 10 batch_size: 128 Accuracy: 98.84 Inference: Increasing the conv layers have a positive effect on the accuracy, since it is able to learn more features of the data.

As we can see from this table CNNs have performed much better as compared to ANNs. In particular CNNs with 2 convolutional layers have performed the best for both the agents.

6) **Do you think increasing the size or complexity of your model would offer any improvements? Why or why not?**

For question 5, we experimented with a lot of architectures to train different models altering number of layers, nodes per layer, activation functions, window sizes, strides, filters, etc in attempt to find the optimal models.

We changed the size and complexity of our models as well, while some had a positive effect on the accuracy of the model, others had a negative effect, this is so because everything depends on the data. Nothing absolute can be said about these things. Every data is different and therefore the optimal technique to learn that data is also different, we just have to find the best technique that suits our data the most. A summary of this can be seen in the above table. According to the table the most optimal models have come out to be:

<u>Agent 1</u>:

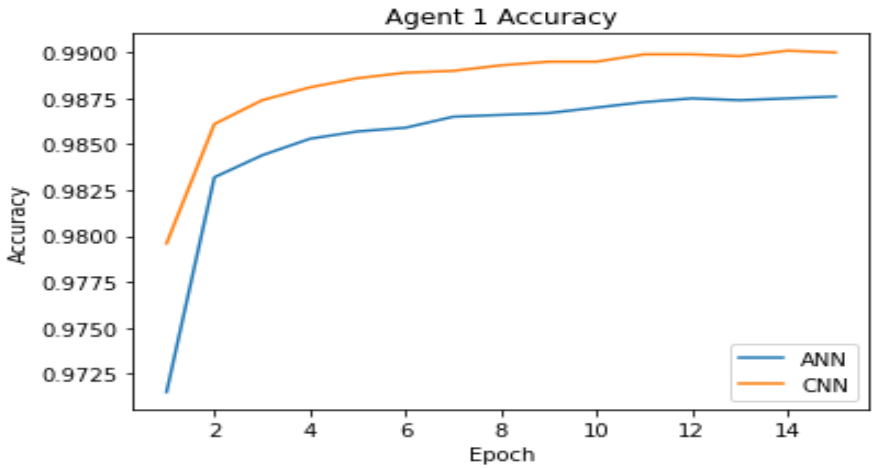
a. Feedforward Neural Network

model 7:
name: My5b5_Ag1_9885_model

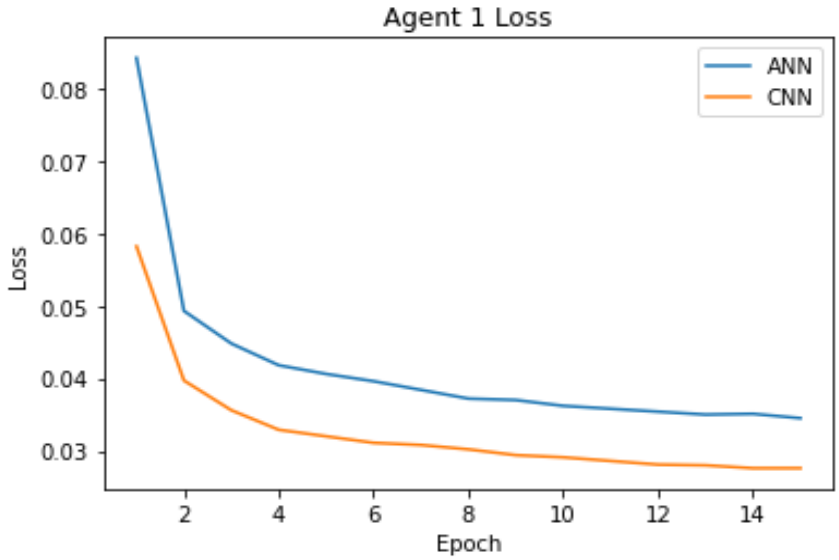
layers: 25, 16, 4
learning_rate: 0.005
epochs: 15
batch_size: 64
Accuracy: 98.85

b. Convolutional Neural Network

model 12:
name: My_conv9917_model
layers: Conv2D(64, (2,2), valid padding, input_dim = (5,5,1)),Conv(32, (2,2), valid padding, input_dim = (8,8,1)) Flatten, Dense(16), Dense(4)
learning_rate: 0.005
epochs: 15
batch_size: 64
Accuracy: 99.17



Agent 1 optimal models for ANN vs CNN accuracy plot



Agent 1 optimal models for ANN vs CNN loss plot

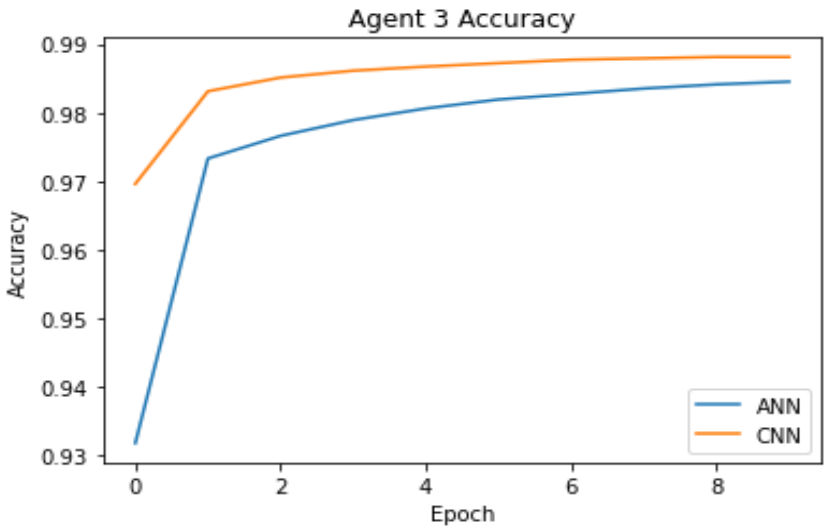
Agent 3:

a. Feedforward Neural Network

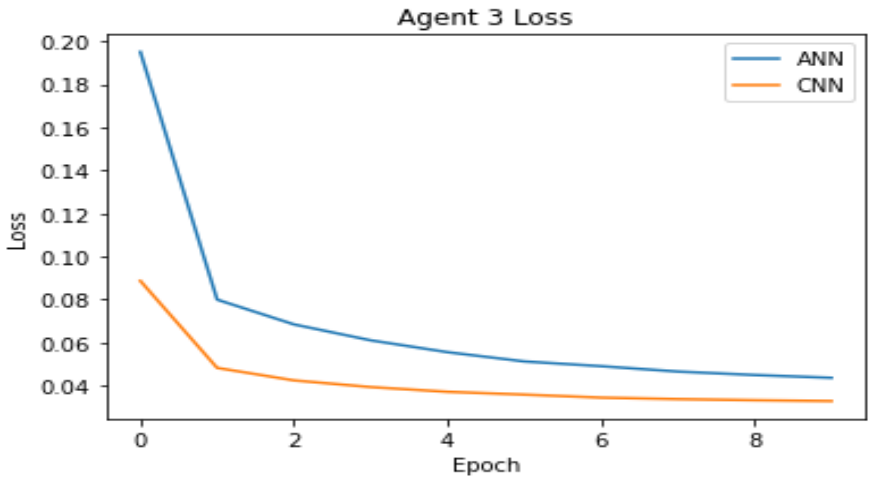
model 2:
name: My5b5_Ag3_9843_model
layers: 25, 16, 8, 4
learning_rate: 0.0025
epochs: 10
batch_size: 128
Accuracy: 98.43

b. Convolutional Neural Network

model 12:
name: My5b5_conv9884_model
layers: Conv2D(64, (2,2), valid padding, input_dim = (13,2,1)),Conv(32, (2,2), valid padding), Flatten, Dense(16), Dense(4)
learning_rate: 0.0025
epochs: 10
batch_size: 128
Accuracy: 98.84



Agent 3 optimal models for ANN vs CNN accuracy plot

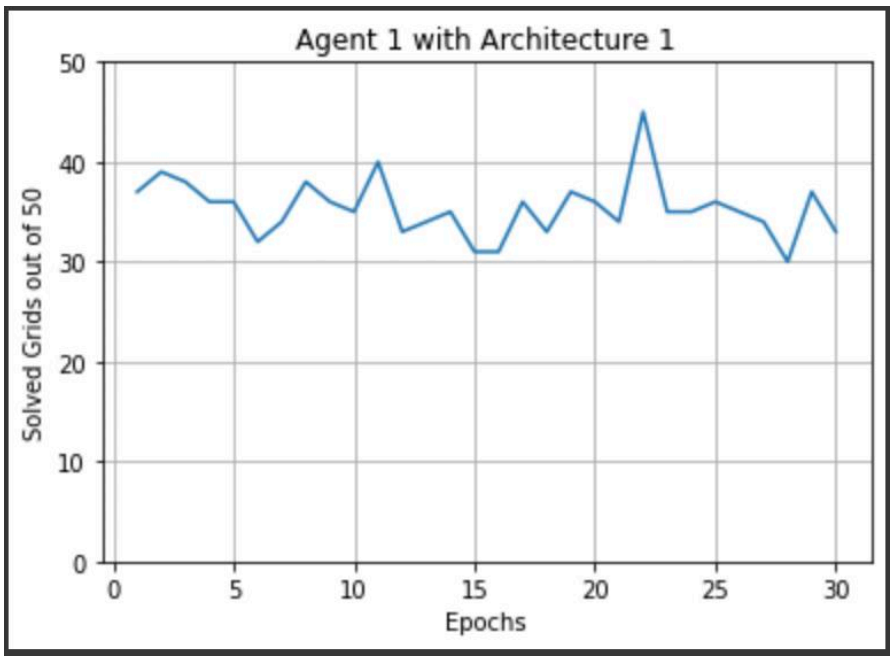


Agent 3 optimal models for ANN vs CNN loss plot

7) **Does good performance on test data correlate with good performance in practice? Simulate the performance of your ML agent on new grid worlds to evaluate this.**

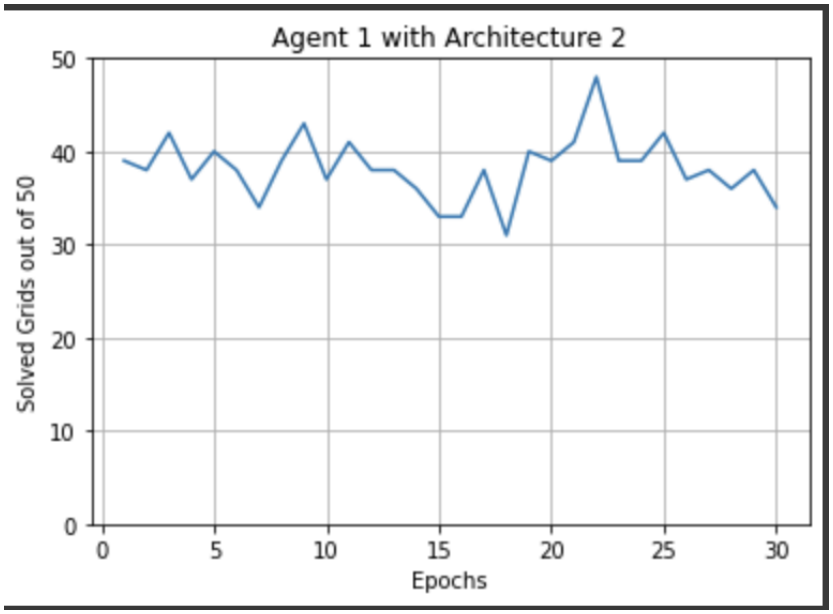
We get very good performance while training and testing the model. However, it does not reciprocate very well when testing on new gridworlds. We run 50 solvable mazes per epoch for 30 epochs and collect the data for performance evaluation.

Below are the plots for Agent 1's performance with Architecture 1



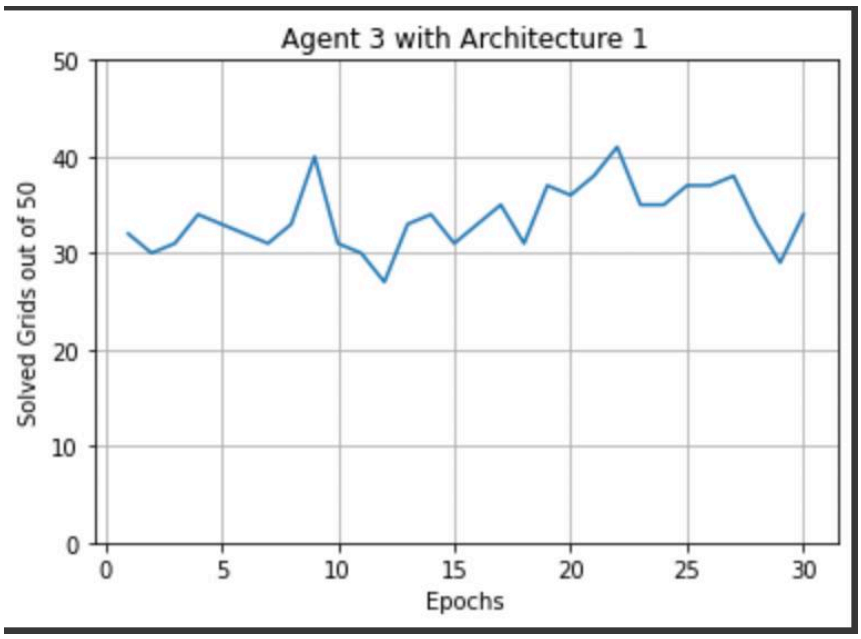
Dense Layers: Agent 1's plot for no. of grids solved out of 50 grids in 30 epochs.

Architecture 1 of Agent 1 is able to solve majority of the grids, with average of 70% grid solvability.



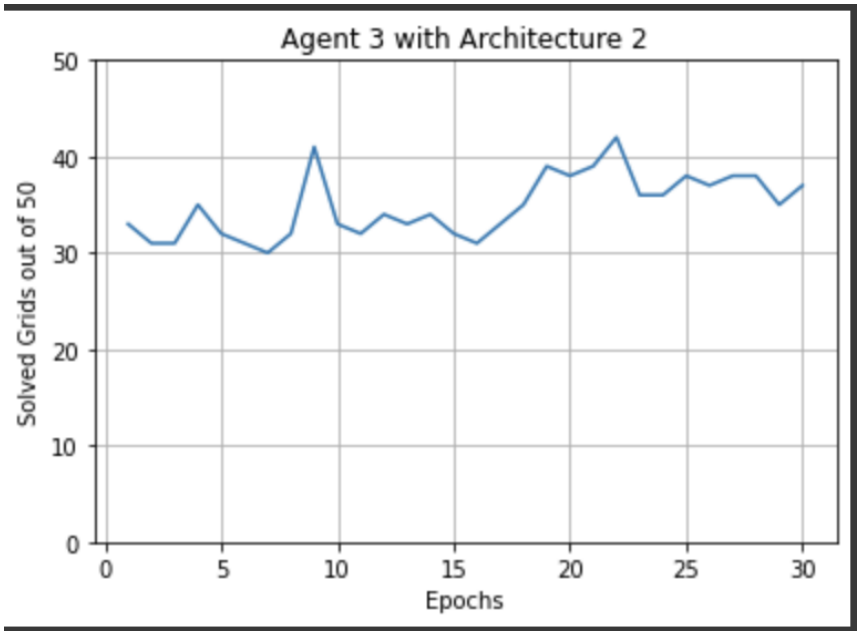
CNN: Agent 1's plot for no. of grids solved out of 50 grids in 30 epochs.

Architecture 2 of Agent 1 performs much better with respect to Architecture as it has 2 convolutional layers added which helps it to learn more features about the data. Solves around 77% percent of the grids



Dense Layers: Agent 3's plot for no. of grids solved out of 50 grids in 30 epochs.

Architecture 1 of agent 3 does not perform too good as compared to its training phase. Also, there are more solvable grids which could not be solved by ML agent.



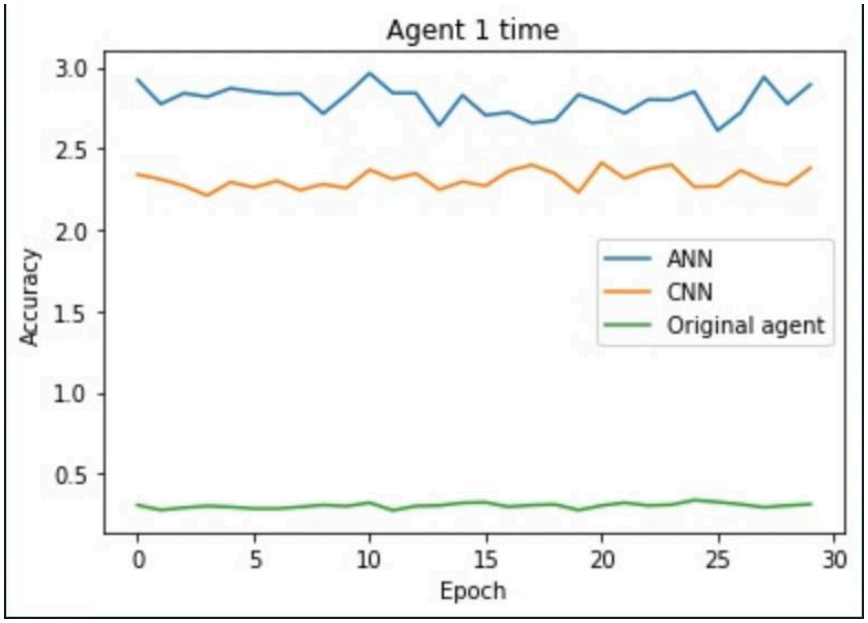
CNN: Agent 1's plot for no. of grids solved out of 50 grids in 30 epochs.

CNN Architecture of Agent 3 again performs better as compared to its dense layers, which is expected as it was in the case of Agent 1.

8) **For your best model structure, for each architecture, plot a) performance on test data as a function of training rounds, and b) average performance in practice on new grid worlds. How do your ML agents stack up against the original agents? Do either ML agents offer an advantage in terms of training time?**

Agent 1 – Trajectory Comparison

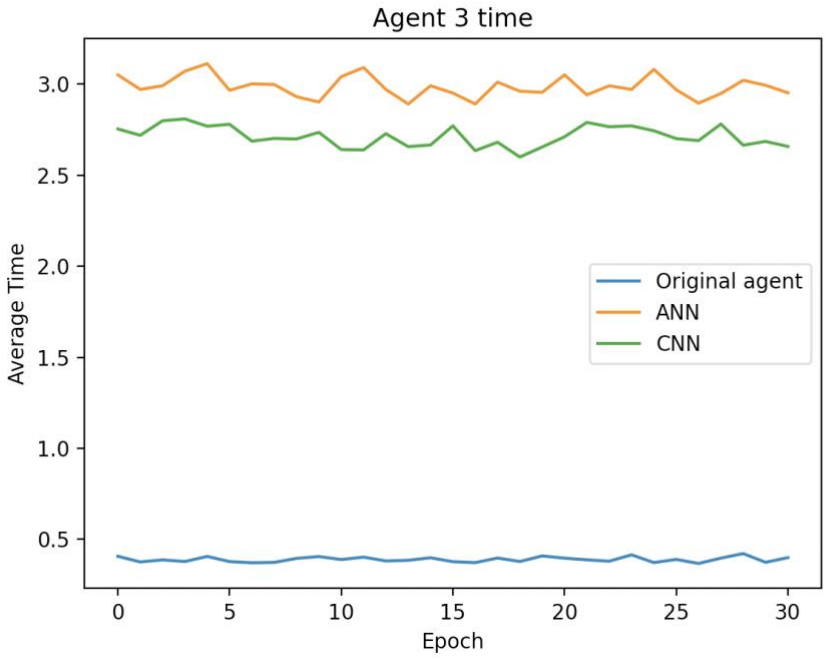
Average Trajectory of CNN seems to be better on an average than the Dense Network and original agent



Agent 1 – Time Comparison

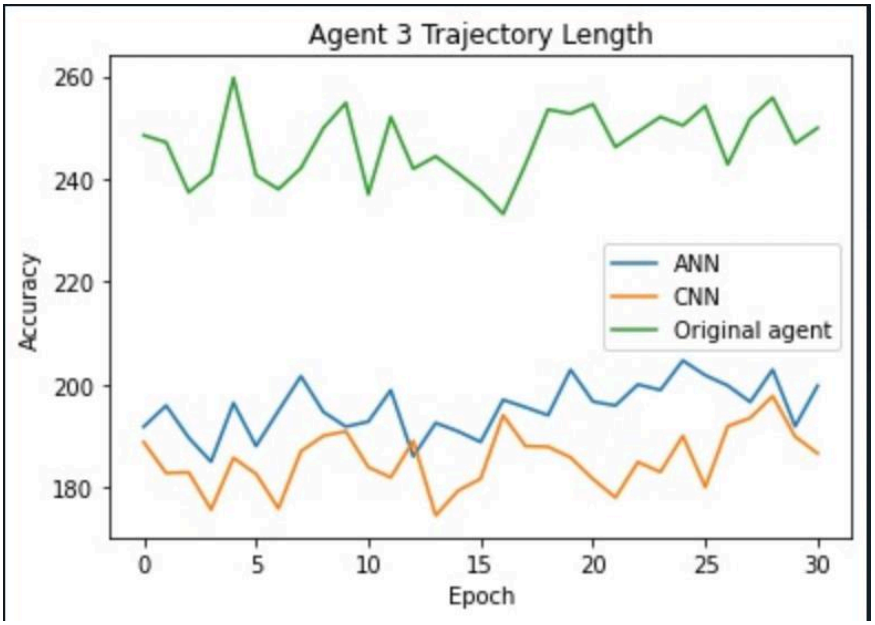
Original agents take least time to solve all the grids, followed by CNN and then Dense Network

---



Agent 3-Time comparison

Original agents take least time to solve all the grids, followed by CNN and then Dense Network



Agent 3 – Trajectory Comparison

Average Trajectory of CNN seems to be better than the Dense Network and original agent

In some cases when ML model goes into continuous loop of going back towards block (these cases would have eventually fallen under unsolvable grids), after certain number of repetitions, instead of taking the argmax, we take the second highest probability of the directions label which breaks this movement of going backward and forward in few cases.

Individual Contribution:

1. Parth Goel: Developed and explored the model architectures.
2. Prashant Kanth: Generated the data using the original agents.
3. Utkarsh Jha: Ran the tests to compare the original and ML agents