CS 520: Project 4 - The Imitation Game

Jenil Ashwin Jain - jj822
Parth Hasmukh Jain - pj269
Samyuktha Gopalsamy - sg1652

Contents

Abstract

This project aims to train two agents to imitate the agents we have previously built from projects 1 and 2. We have chosen the 4-neighbor agent from Project 1 and the Example Inference agent from Project 2 and built two ML agents that mimic its actions. We experiment with two kinds of neural network architectures (one with only Full Dense Layers and the other with at least one Convolutional Neural Layer) for each agent. The ML agents make appropriate selections in the grid world, given the current state. We evaluate and compare the ML agents against the original agents.

Problem Definition

We build two ML agents to mimic the actions taken by the agents chosen from Projects 1 (4-neighbor agent) and 2(Example Inference agent) in the following way:

- The state descriptions of the current knowledge of the grid world is given as input
- The corresponding actions taken by the agent is given as the input
- The ML agent consists of a neural network that maps from the input space to the output space
- The model is trained on the data generated by the agents over many different grid worlds and episodes
- The trained network is used to make action selections given the current state

Data Generation

As the agent traverses the grid world to reach the target, a 9X9 local snippet of the knowledge grid around the current position is taken at each step as the agents' input data. Next, the current position in the grid world for each step is also taken as the input. Finally, the corresponding action taken at each state is stored. The agents from Projects 1 and 2 are solved on over 6500 episodes of 50 x 50 grid worlds.

So, to sum up, as the agent traverses the grid world, the current instance of the knowledge grid, the current position, and the corresponding action taken by the agent are stored. Then collected data is dumped into a pickle file for storage and further processing.

Number of different gridworld/episodes: 6500 (3500: Training, 3000: Testing)
Size of grid world: 50 x 50

4-Neighbor agent (Project 1)

The input data for the agent from project 1 consists of the flattened 9x9 snippet of the knowledge grid around the current position at a given instance. The knowledge grid world represents the updated local information of the presence of blocked and unblocked cells in the entire grid. The output data consists of the corresponding actions the agent takes on the input states.

Example Inference agent (Project 2)

The input data for the agent from project 2 consists of the flattened knowledge grid at a given instance, the corresponding C_x(sensed information). The knowledge grid world represents the updated global information of the presence of blocked and unblocked cells in the grid. In contrast, the clue values like C_x represent the local information about the utmost eight neighbors.The output data consists of the corresponding actions the agent takes on the input states.

Data Wrangling

The knowledge grid collected is initially represented as a dim x dim matrix (2d array). It is transformed to a flattened array (1d array) and then transformed to a tensor. The tensor of the knowledge grid is given as input to the neural network model.

The actions taken by the agent are initially collected as {left, right, up and down} based on the direction in which the agent moves from the current cell position to the neighbor. The categorical direction variables are encoded to binary vectors using the Keras to_categorical() method.

Evaluation of Model and Architectures

To find the optimal structure, we experimented with the number of layers, nodes per layer, activation functions, window sizes, strides, filters, etc. The final model architecture has the following characteristics:

Model characteristic	Value	Description
Loss function	Categorical Cross-Entropy	Computes the cross-entropy loss between actual action taken and predicted action.
Activation function for hidden layer	Rectified Linear activation function or ReLU	f(x)=max(0,x) where x is the input to a neuron
Activation function for the output layer	Softmax	Used as the activation function in the output layer of the neural network model to

		predict the multinomial probability distribution
Window size	(3, 3)	Sliding window of size 3 x 3
Number of filters	32	The convolutional layer learns from 32 filters in parallel for a given input.
Optimizer	Adam	Adam optimization is a stochastic gradient descent method that is based on the adaptive estimation of first-order and second-order moments.
Learning rate	0.01	Hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated.
Batch size	64	Defines the number of samples that will be propagated through the network

Q1 State space and action space representation

Project 1:

- In the case of Project 1, we define the state space as the current state of the environment, the current position of the agent in the environment.

- The current state of the environment represents a dim * dim array where ‘0’ indicates the cell is blocked and ‘1’ indicates the cell is unblocked.

- In our project, we consider taking local information with respect to the environment as input. We are taking a 9X9 local snippet of the grid around which the current position is.

- The current position of the Agent is passed as a one-hot encoded vector of size dim as an input.

- The actions taken by the agent are initially collected as {left, right, up and down} based on the direction in which the agent moves from the current cell position to the neighbor.

- The categorical direction variables are encoded to vectors using the Keras to_categorical() method.

- We are mapping each direction to an integer value between 0-3. Action up, down, left, right are represented as 0,1,2 and 3, respectively.

Project 2:

- For Project 2, the state space additionally includes the sensed information of the cells at any stage.

- The sensed knowledge in the state space is represented as a dim*dim array where each cell displays the number of blocked cells among the neighbors. We represent the sensed information as follows:
  - ‘-1’ - The cell is not yet discovered.
  - ‘0-8’ - The number of blocked cells, i.e., C_x.

- The current position of the Agent is passed as a one-hot encoded vector of size dim as an input.

- The neural network takes the state space as an input and outputs the probabilities for actions in the action space.

- The action space consists of the corresponding actions the agent takes on the input states.

Q2 Loss function

We define categorical cross-entropy loss when training our model using tf.keras.metrics.categorical_crossentropy. Categorical cross-entropy is a loss function that is used in multi-class classification tasks. These are tasks where an example can only belong to one out of many possible categories, and the model must decide which one.

It computes the cross-entropy loss between actual actions taken by the agent and predicted actions. We use this cross-entropy loss function as there are more than two action labels that are to be classified are left, right, up, down. The action labels are provided to the model in an one_hot encoded representation. The categorical cross-entropy loss function calculates the loss of an example by computing the following sum:

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

Q3 Number of episodes in Training the model

We have generated data for **3500** iterations for training the model.

Training the model for 3500 iterations resulted in exploring the model for 7 lakh data points. We also tried increasing the number of iterations to 4000 but the results were constant.

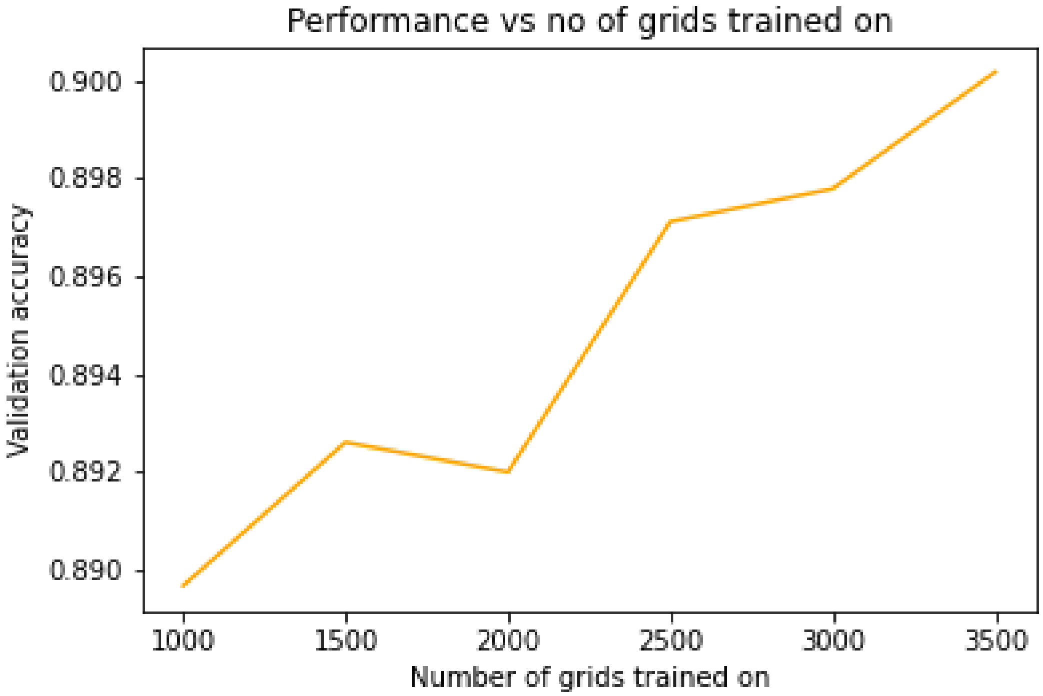


Figure 1 : Performance vs number of grids trained on

Q4 Overfitting

Overfitting in machine learning occurs when the neural network model fits the training data very well but fails to generalize on unseen test data. This results in a good performance on train data with no correlation of good performance on real data.

The following strategies were used to avoid overfitting in our projects:

- We were getting optimum results while using 2 dense hidden layers and 1 dense output layer when we further increased the number of the hidden dense layer we were not1 getting any improvements in accuracy so we decided to stick with 2 hidden layers
- Generated more training data over many episodes and different grid worlds to ensure that the training data included varied patterns
- Performed data wrangling and removed unnecessary features in project 2 input data
- Observed the agent’s performance on validation along with unseen test data. Stopped training the model when validation accuracy did not improve further for a certain number of epochs.

Q5 Architecture space exploration

Summary of Hyperparameters for all the architectures:

Activation function for hidden dense layer	-	Rectified Linear Unit (ReLU)
Activation function for output layer	-	Softmax
Learning Rate	-	0.01
Batch Size	-	64
Epochs	-	10
Loss Function	-	Categorical Cross Entropy
Optimizer	-	ADAM optimizer

Architecture 1 for Agent 1: Model with only dense layers

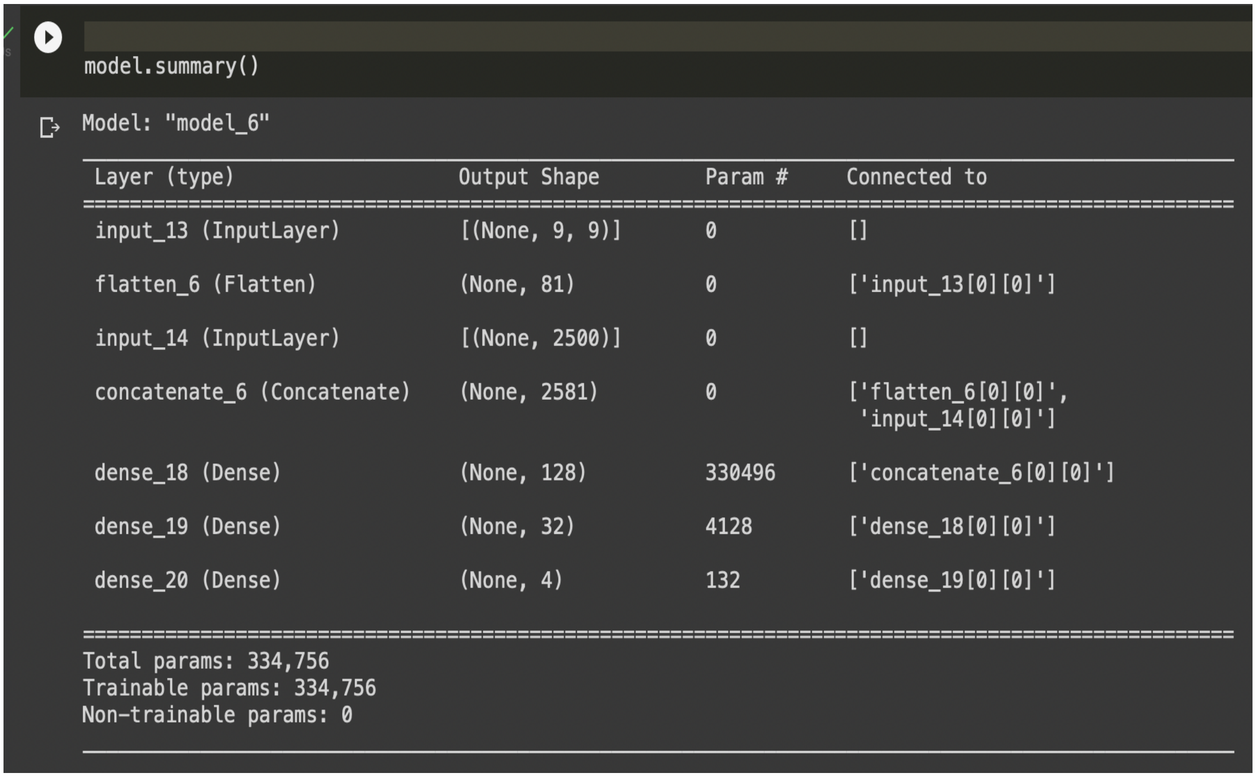
A dense layer is one that is a neural network layer that is connected deeply, which means each neuron in the dense layer receives input from all neurons of its previous layer.

We explored the architecture by keeping an eye on the validation accuracy over different architectures. We tried testing the model with 1 dense layer at first. The model did not achieve the expected results so we tried increasing the dense layers to 2,3 and 4. We observed that after 3 layers the performance did not increase much.

Construction of Architecture 1:

- First, an input layer L1 which takes the input as the knowledge grid.
- A flattened layer after the knowledge grid is added.
- We add an input layer L2 which takes the current position of the agent in the grid world in the one-hot encoded form.
- Then the three input layers are concatenated.
- Next, we add 3 Fully Dense Layers with the last layer being the output layer.

Initially, we tested the model with 2 Fully dense layers with 32 nodes in each dense layer but we were getting an accuracy of 83%. So then we increased the number of nodes in each dense layer, so we used 128 nodes for the first layer and 32 for the first layer then we got an accuracy of 90.2%. After the 2 dense layers, we add a final dense output layer with 4 nodes which gives us the output in the form of actions mapped as numbers.



```
model.summary()

Model: "model_6"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_13 (InputLayer)          [(None, 9, 9)]       0           []

 flatten_6 (Flatten)            (None, 81)           0           ['input_13[0][0]']

 input_14 (InputLayer)          [(None, 2500)]       0           []

 concatenate_6 (Concatenate)    (None, 2581)         0           ['flatten_6[0][0]',
                                                                  'input_14[0][0]']

 dense_18 (Dense)               (None, 128)          330496      ['concatenate_6[0][0]']

 dense_19 (Dense)               (None, 32)           4128        ['dense_18[0][0]']

 dense_20 (Dense)               (None, 4)            132         ['dense_19[0][0]']

==================================================================================================
Total params: 334,756
Trainable params: 334,756
Non-trainable params: 0
__________________________________________________________________________________________________
```

Figure 2: Model Summary For Agent 1 with dense layer

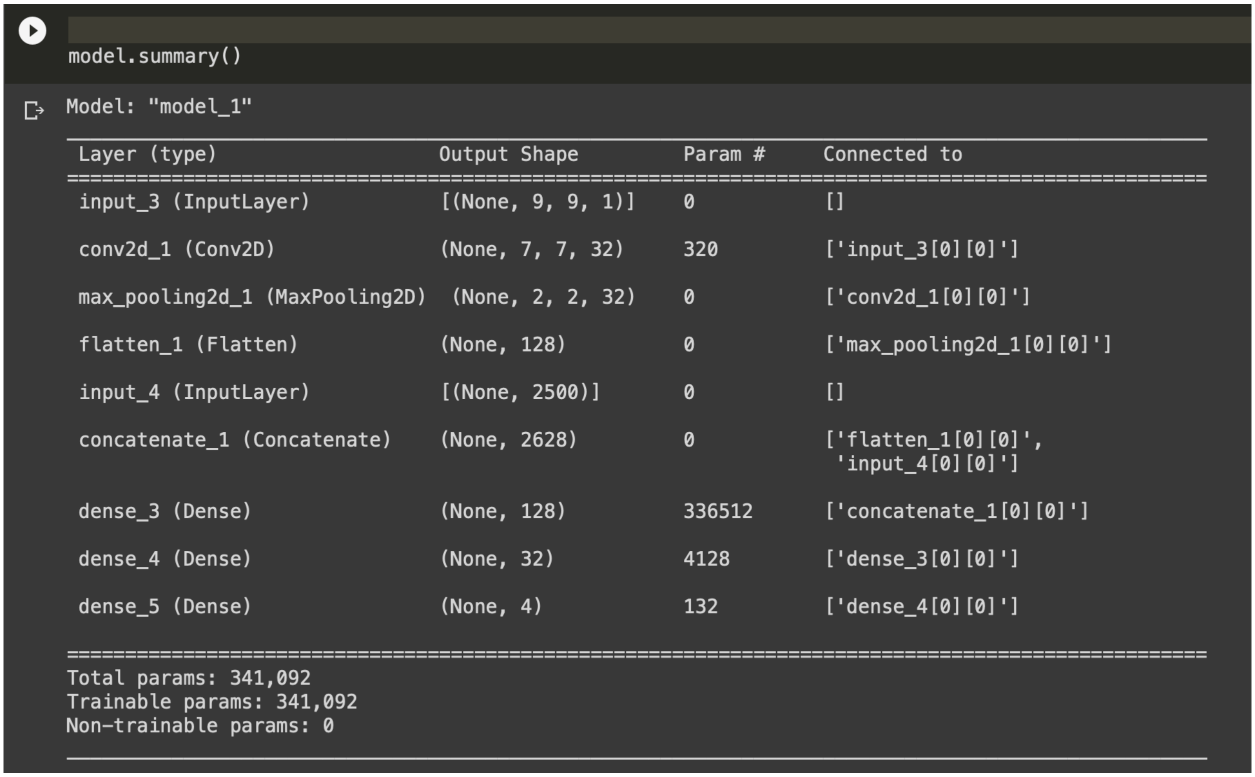
Architecture 2 for Agent 1: At Least One Convolutional Neural Layer

Convolutional layers are the major building blocks used in convolutional neural networks. Convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input.

In this along with a convoluted layer, we added max pooling for the convoluted layer. Max pooling is a pooling operation that calculates the maximum value for patches of a feature map, and uses it to create a downsampled (pooled) feature map. We experimented with applying max pooling and not applying max pooling, the model with max-pooling gave a better accuracy than the model without max pooling.

Construction of the Architecture:

- First, an input layer L1 which takes the input as the knowledge grid.
- Reshape the input layer to a 4D tensor as the Conv2D expects the input to be 3 dimensional.
- A max-pooling layer after the Conv2D layer is added.
- A flatten layer takes the output of this layer as input.
- We add an input layer L2 which takes the current position of the agent in the grid world in the one-hot encoded form.
- Then the three input layers are concatenated.
- Next, we add 3 Fully Dense Layers with the last layer being the output layer.



```
model.summary()
```

```
Model: "model_1"
_________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
=================================================================================================
 input_3 (InputLayer)           [(None, 9, 9, 1)]    0           []

 conv2d_1 (Conv2D)              (None, 7, 7, 32)     320         ['input_3[0][0]']

 max_pooling2d_1 (MaxPooling2D)  (None, 2, 2, 32)    0           ['conv2d_1[0][0]']

 flatten_1 (Flatten)            (None, 128)          0           ['max_pooling2d_1[0][0]']

 input_4 (InputLayer)           [(None, 2500)]       0           []

 concatenate_1 (Concatenate)    (None, 2628)         0           ['flatten_1[0][0]',
                                                                  'input_4[0][0]']

 dense_3 (Dense)                (None, 128)          336512      ['concatenate_1[0][0]']

 dense_4 (Dense)                (None, 32)           4128        ['dense_3[0][0]']

 dense_5 (Dense)                (None, 4)            132         ['dense_4[0][0]']

=================================================================================================
Total params: 341,092
Trainable params: 341,092
Non-trainable params: 0
_________________________________________________________________________________________________
```

Figure 3: Model Summary For Agent 1 with Convolutional layer

Architecture 1 for Agent 3: Model with only dense layers

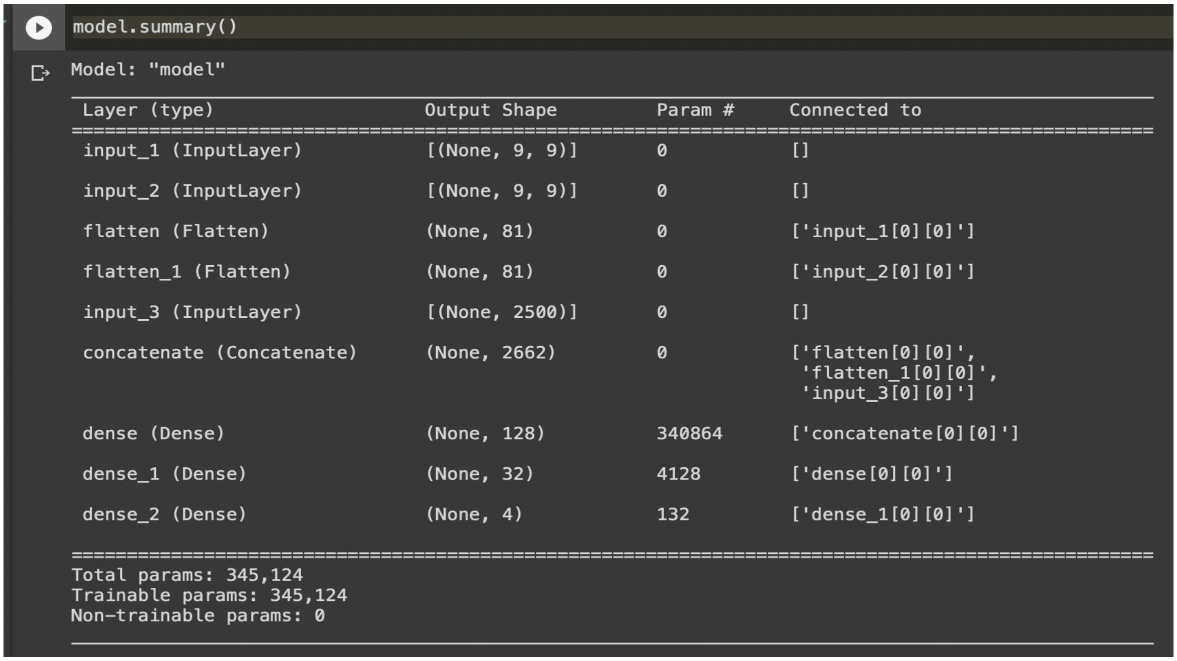
A dense layer is one that is a neural network layer that is connected deeply, which means each neuron in the dense layer receives input from all neurons of its previous layer.

We explored the architecture by keeping an eye on the validation accuracy over different architectures. We tried testing the model with 1 dense layer at first. The model did not achieve the expected results so we tried increasing the dense layers to 2,3 and 4. We observed that after 3 layers, the performance did not increase much.

Construction of the Architecture:

- First, an input layer L1 which takes the input as the knowledge grid.
- Next input layer L2 which takes the input of the sensed information Cx.
- The input layers L1 and L2 are flattened.
- We add an input layer L3 which takes the current position of the agent in the grid world in the one-hot encoded form.
- Then the three input layers are concatenated.
- Next, we add 3 Fully Dense Layers with the last layer being the output layer.

Initially, we tested the model with 2 Fully dense layers with 32 nodes in each dense layer but we did not get the desired accuracy. So then we increased the number of nodes in each dense layer, so we used 128 nodes for the first layer and 32 for the first layer then we got a desired accuracy . After the 2 dense layers, we add a final dense output layer with 4 nodes which gives us the output in the form of actions mapped as numbers.



```
model.summary()
```

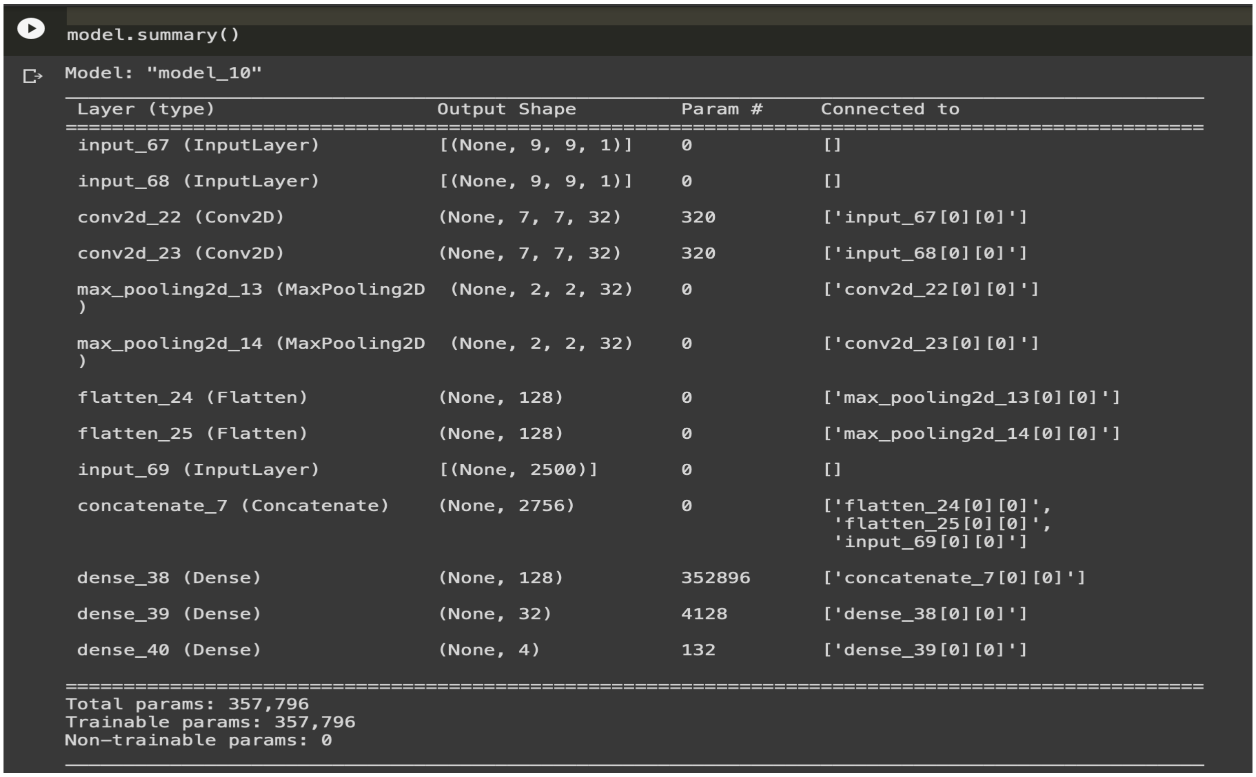
```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 9, 9)]       0           []

 input_2 (InputLayer)           [(None, 9, 9)]       0           []

 flatten (Flatten)              (None, 81)           0           ['input_1[0][0]']

 flatten_1 (Flatten)            (None, 81)           0           ['input_2[0][0]']

 input_3 (InputLayer)           [(None, 2500)]       0           []

 concatenate (Concatenate)      (None, 2662)         0           ['flatten[0][0]',
                                                                  'flatten_1[0][0]',
                                                                  'input_3[0][0]']

 dense (Dense)                  (None, 128)          340864      ['concatenate[0][0]']

 dense_1 (Dense)                (None, 32)           4128        ['dense[0][0]']

 dense_2 (Dense)                (None, 4)            132         ['dense_1[0][0]']

==================================================================================================
Total params: 345,124
Trainable params: 345,124
Non-trainable params: 0
__________________________________________________________________________________________________
```

Figure 4: Model Summary For Agent 3 with Fully dense layer

Architecture 2 for Agent 3: At Least One Convolutional Neural Layer

Construction of the Architecture:

- First, an input layer L1 which takes the input as the knowledge grid.
- Reshape the input layer to a 4D tensor as the Conv2D expects the input to be in 3 dimensional.
- A max-pooling layer after the Conv2D layer is added.
- A flatten layer takes the output of this layer as input.
- We add an input layer L2 which takes the current position of the agent in the grid world in the one-hot encoded form.
- Then the three input layers are concatenated.
- Next, we 3 Fully Dense Layers with the last layer being the output layer.



```
model.summary()
```

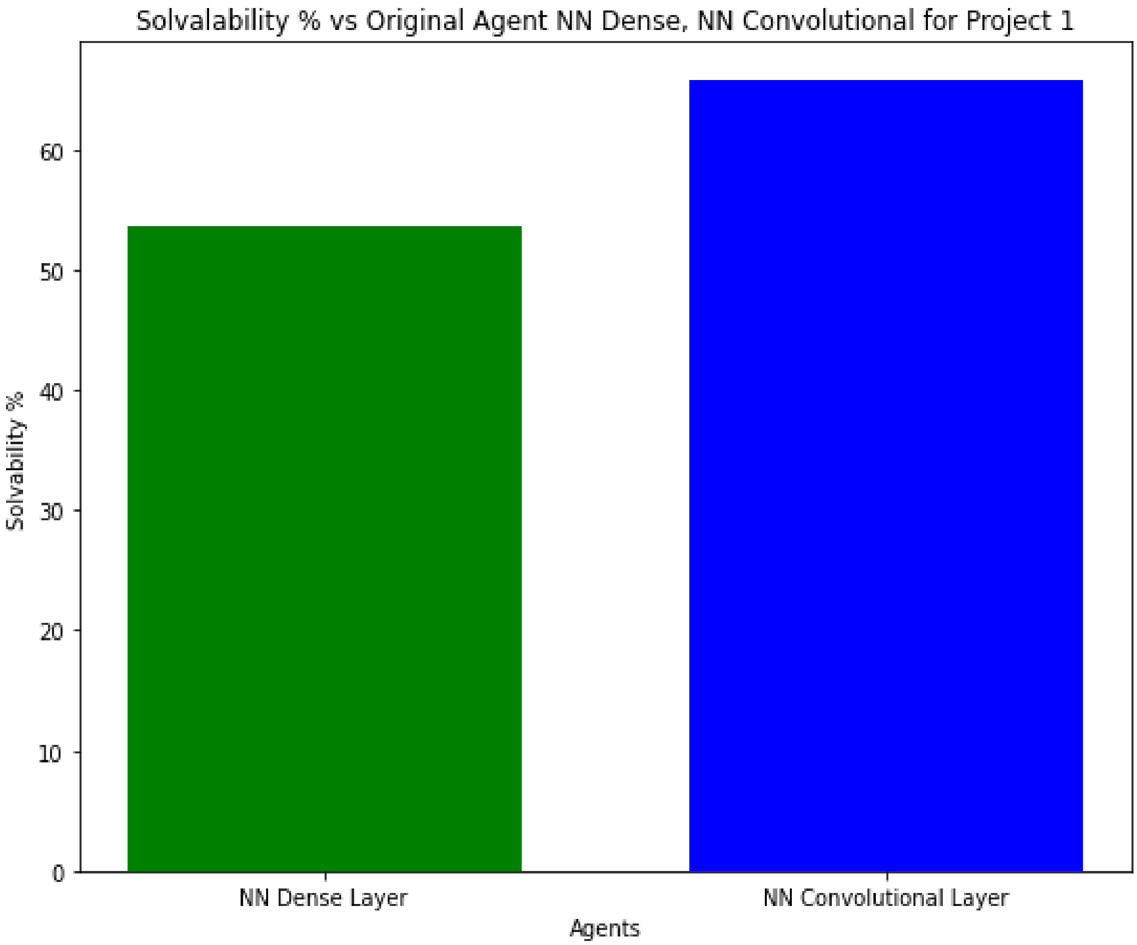
```
Model: "model_10"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_67 (InputLayer)          [(None, 9, 9, 1)]    0           []

 input_68 (InputLayer)          [(None, 9, 9, 1)]    0           []

 conv2d_22 (Conv2D)             (None, 7, 7, 32)     320         ['input_67[0][0]']

 conv2d_23 (Conv2D)             (None, 7, 7, 32)     320         ['input_68[0][0]']

 max_pooling2d_13 (MaxPooling2D  (None, 2, 2, 32)    0           ['conv2d_22[0][0]']
 )

 max_pooling2d_14 (MaxPooling2D  (None, 2, 2, 32)    0           ['conv2d_23[0][0]']
 )

 flatten_24 (Flatten)           (None, 128)          0           ['max_pooling2d_13[0][0]']

 flatten_25 (Flatten)           (None, 128)          0           ['max_pooling2d_14[0][0]']

 input_69 (InputLayer)          [(None, 2500)]       0           []

 concatenate_7 (Concatenate)    (None, 2756)         0           ['flatten_24[0][0]',
                                                                  'flatten_25[0][0]',
                                                                  'input_69[0][0]']

 dense_38 (Dense)               (None, 128)          352896      ['concatenate_7[0][0]']

 dense_39 (Dense)               (None, 32)           4128        ['dense_38[0][0]']

 dense_40 (Dense)               (None, 4)            132         ['dense_39[0][0]']

==================================================================================================
Total params: 357,796
Trainable params: 357,796
Non-trainable params: 0
__________________________________________________________________________________________________
```

Figure 5: Model Summary For Agent 3 with Convolutional layers

Q6 Model complexity

We tried experimenting with the different features of the model. We tried increasing the filters, the number of layers but the accuracy of the model saturated and the model resulted in overfitting, so we used models with the above model summary.

Q7 Model generalization



The original agent's solvability is 100% for all the solvable grids. The NN agent with fully dense layers has a solvability of 53.6%. And the NN agent with a convolutional layer has a solvability of 65.7%. The test data was generated for 3000 solvable grid worlds. The reason for such a solvability rate is that the ML agent gets stuck between the blocked cells where the original agent finds the path by backtracking.

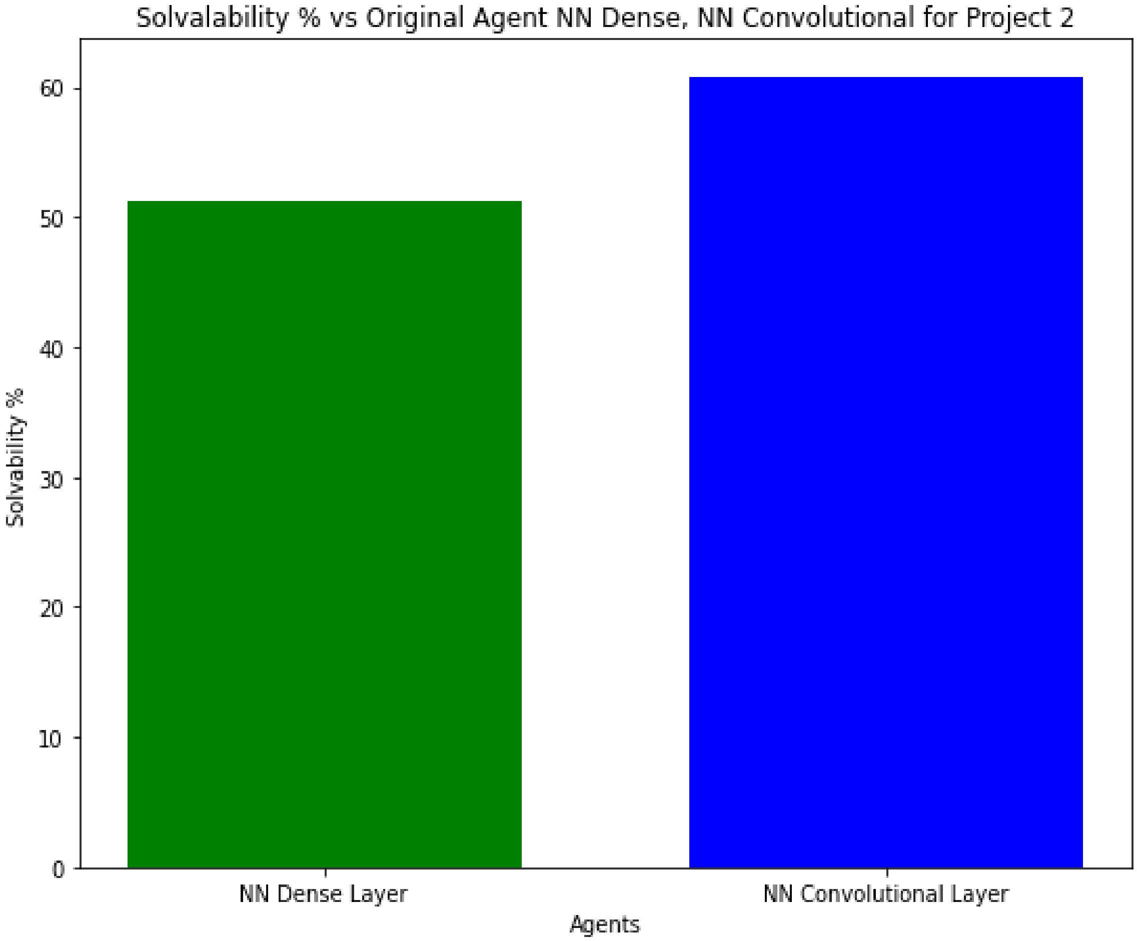
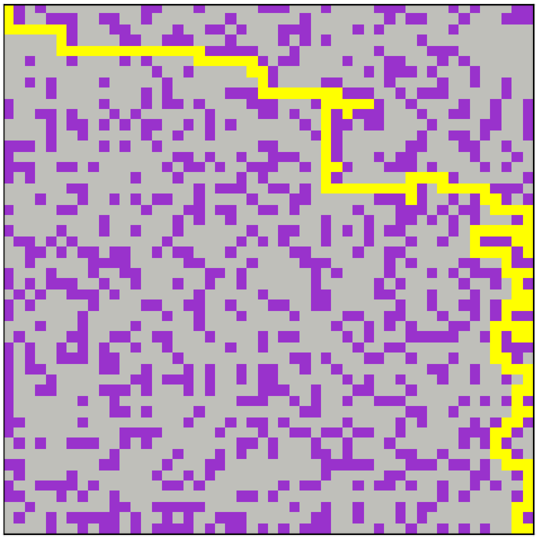


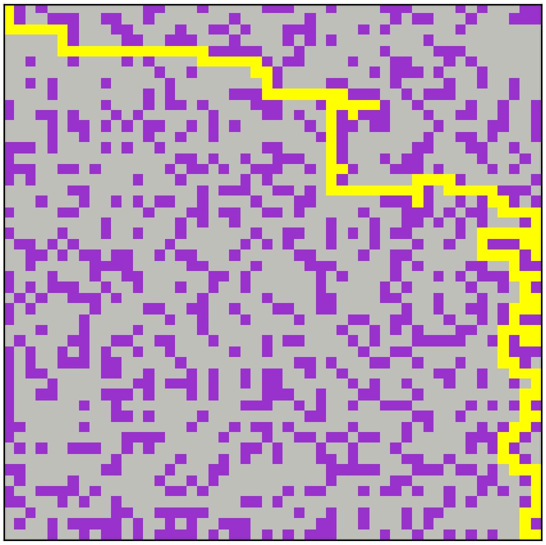
Figure: Solvability of grid world vs agents

The original agent's solvability is 100% for all the solvable grids. The NN agent with a fully dense layer has a solvability of around 51%. The NN agent with a convolutional layer has a solvability of around 60%. The test data was generated for 3000 solvable grid worlds. The reason for such a solvability rate is that the ML agent gets stuck between the blocked cells where the original agent finds the path by backtracking.

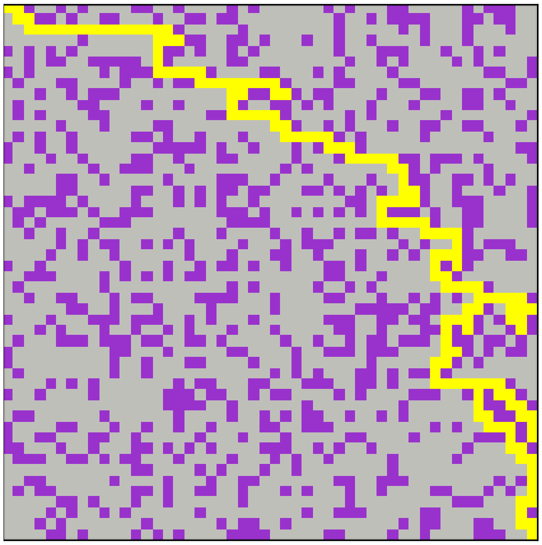
Final paths traversed by the agents



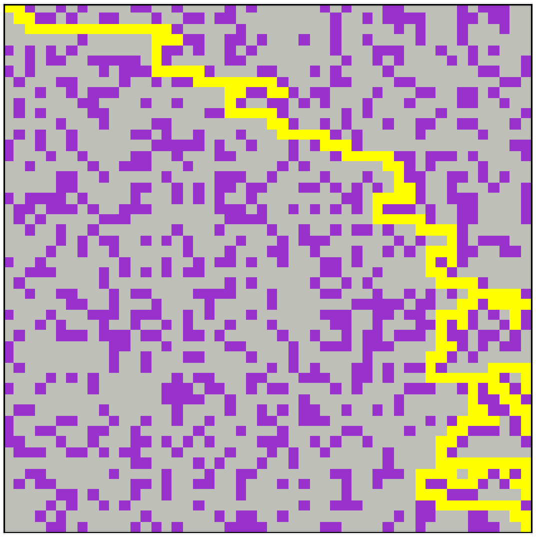
Repeated Astar path for Project 1



NN Agent path for Project 1



Repeated Astar path for Project 2



NN Agent path for Project 2

As we can see in both the projects the path followed by the NN agent and the Astar is relatively similar.

Q8 Performance plots

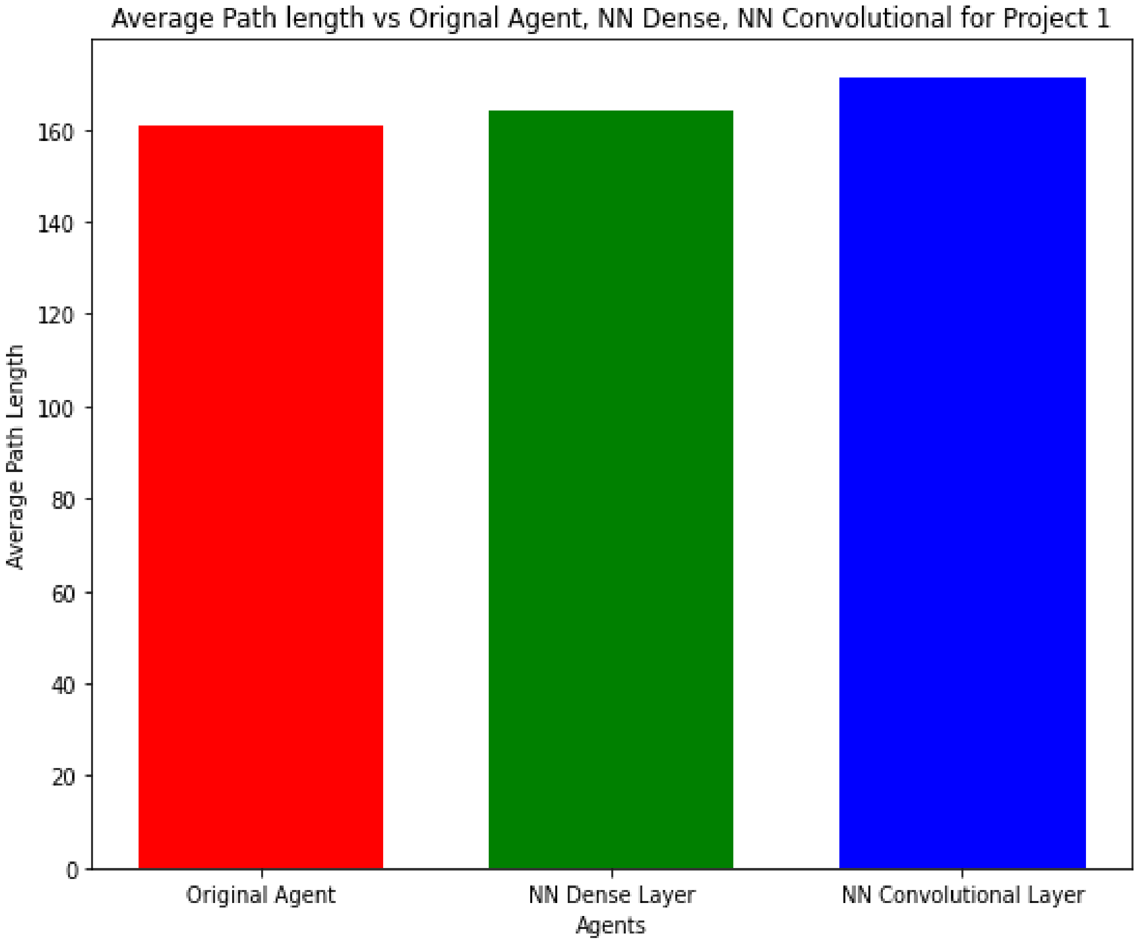


Figure : Average Path length vs Agents (Project 1)

The above figure was generated by running the Original Agent, ML Agent with dense layers and ML agent with Convolutional layer on grids of dimension 50x50. Here, we observe that the Original Agent was able to solve the grid in 161 steps, the ML agent with fully dense layers takes 164 steps and the ML agent with the Convolutional layer takes 167 steps. As the ML agents have learned from the original agent the path lengths are comparable.

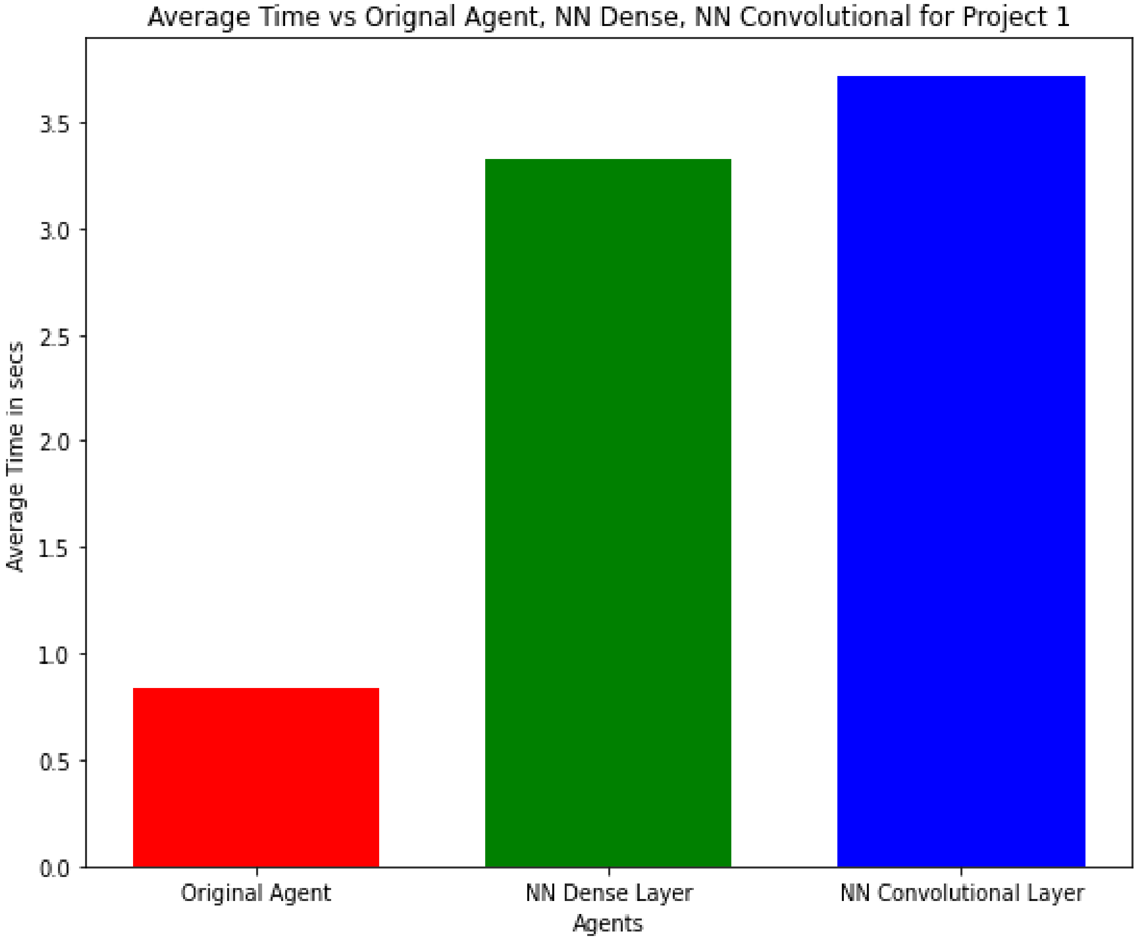


Figure : Average TIme in secs vs Agents (Project 1)

The above figure was generated by running the Original Agent, ML Agent with dense layers and ML agent with Convolutional layer on grids of dimension 50x50. Here, we observe that the Original Agent was able to solve the grid in 0.7 seconds, the ML agent with fully dense layers takes 3.35 seconds and the ML agent with the Convolutional layer takes 3.64 seconds.

The predictions are taken at each step, that is why the ML agent takes so much more time than the original agent. So, in general, the predictions do not take time but when a large number of predictions equal to the path length are called it takes a large amount of time.

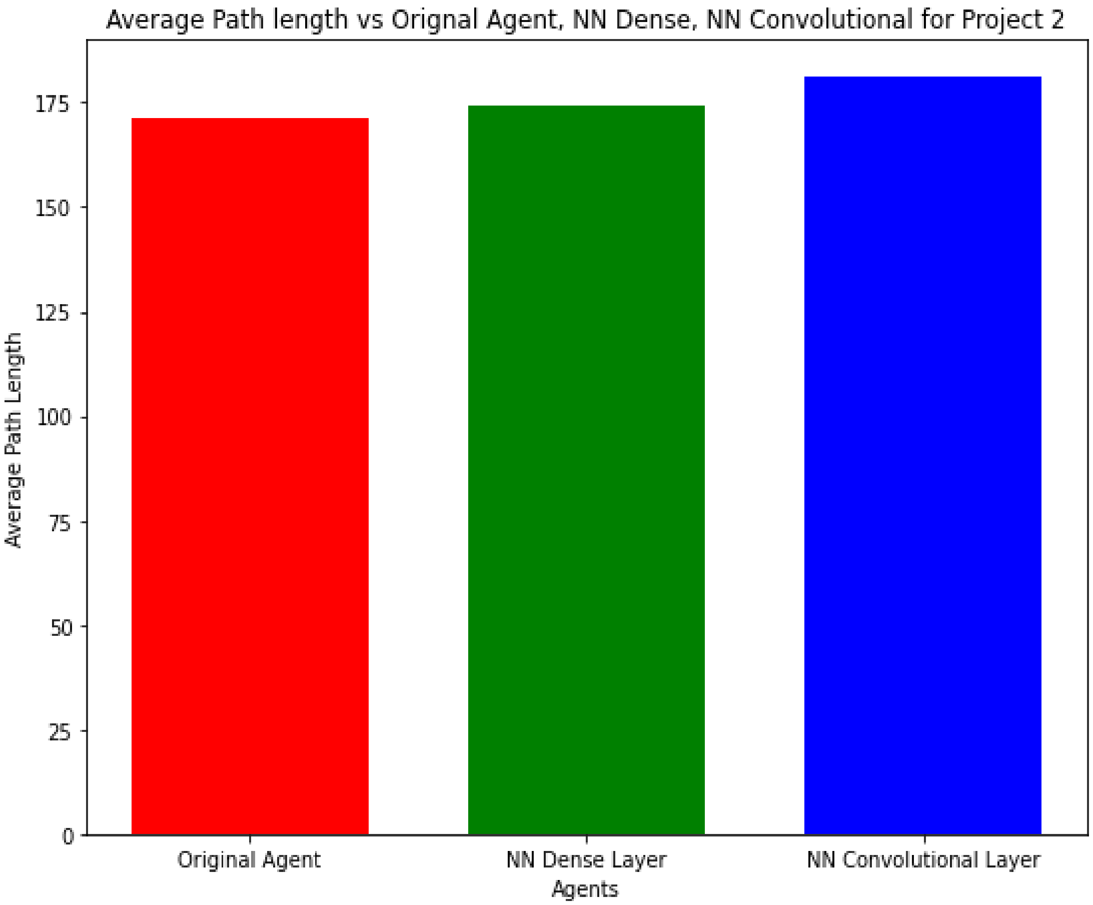


Figure : Average Path length vs Agents (Project 2)

The above figure was generated by running the Original Agent, ML Agent with dense layers and ML agent with Convolutional layer on grids of dimension 50x50. Here, we observe that the Original Agent was able to solve the grid in 173 steps, the ML agent with fully dense layers takes 176 steps and the ML agent with the Convolutional layer takes 183 steps. As the ML agents have learned from the original agent, the path lengths are comparable.

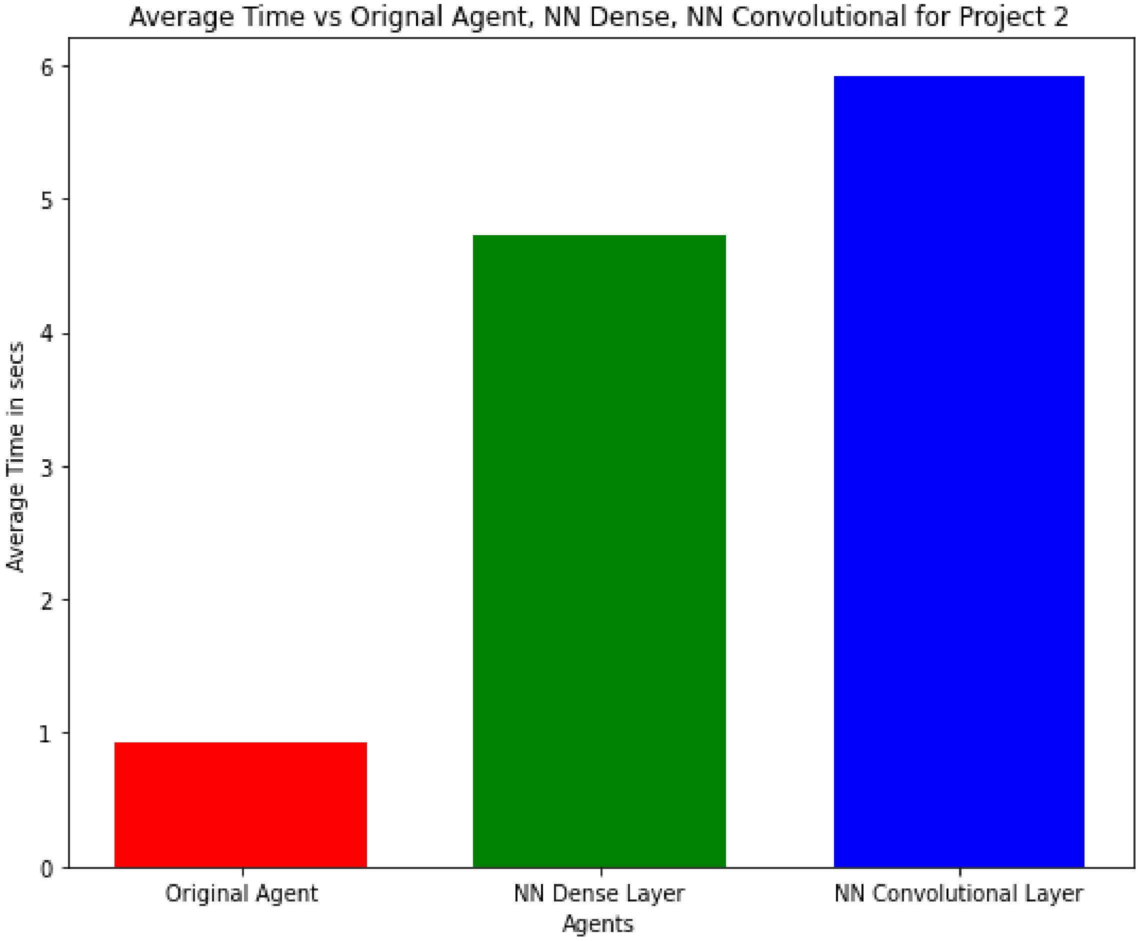


Figure : Average TIme in secs vs Agents (Project 2)

The above figure was generated by running the Original Agent, ML Agent with dense layers and ML agent with Convolutional layer on grids of dimension 50x50. Here, we observe that the Original Agent was able to solve the grid in 0.86 seconds, the ML agent with fully dense layers takes 4.71 seconds, and the ML agent with the Convolutional layer takes 5.9 seconds.

The predictions are taken at each step, that is why the ML agent takes so much more time than the original agent. So, in general, the predictions do not take time, but when a large number of predictions equal to the path length are called it takes a large amount of time.

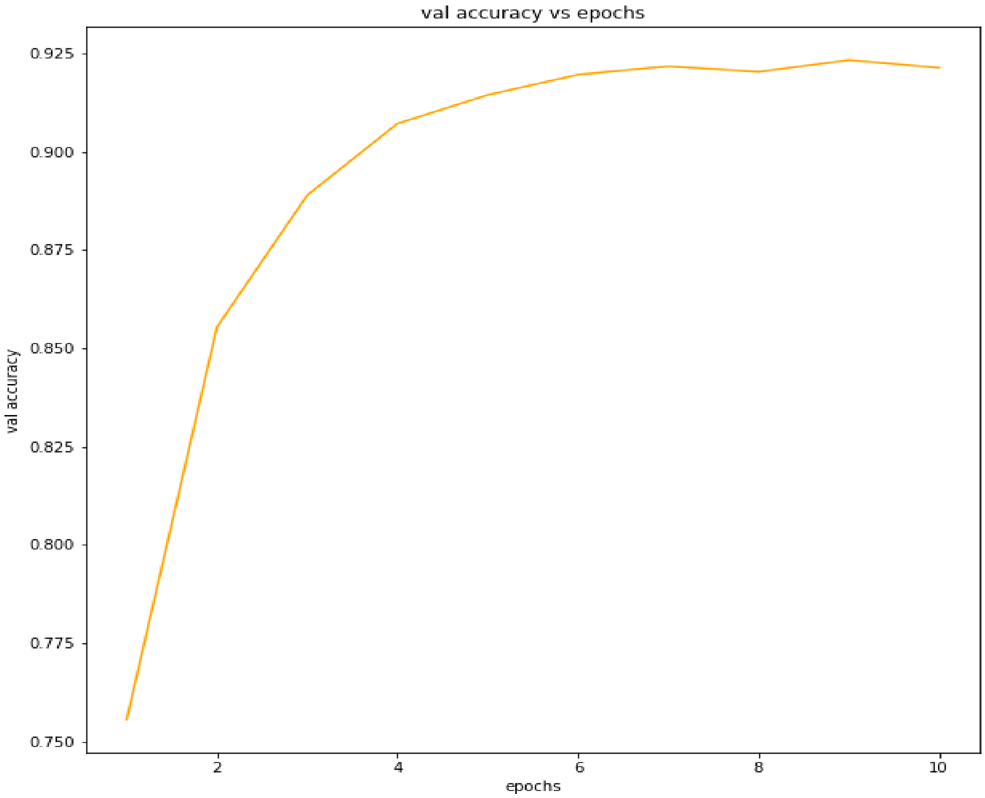


Figure : Validation accuracy vs number of epochs for NN with dense layers (Project 1)

The above figure is generated for ML Agent with dense layers for 10 epochs. The accuracy value ranges from 76.12% to 91.37% over the 10 epochs. We can also observe from the graph that the accuracy plateaus after 10 epochs.

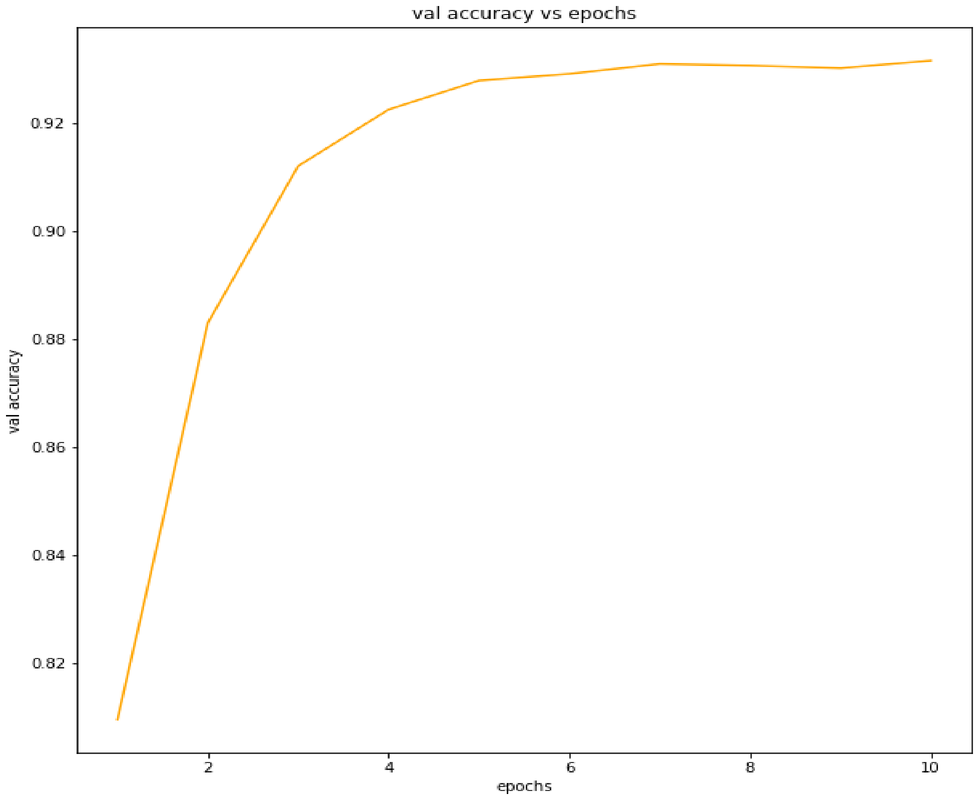


Figure: Validation accuracy vs number of epochs for NN with convolutional layers (Project 1)

The above figure is generated for ML Agent with dense layers for 10 epochs. The accuracy value ranges from 81% to 92% over the 10 epochs. We can also observe from the graph that the accuracy plateaus after 10 epochs.

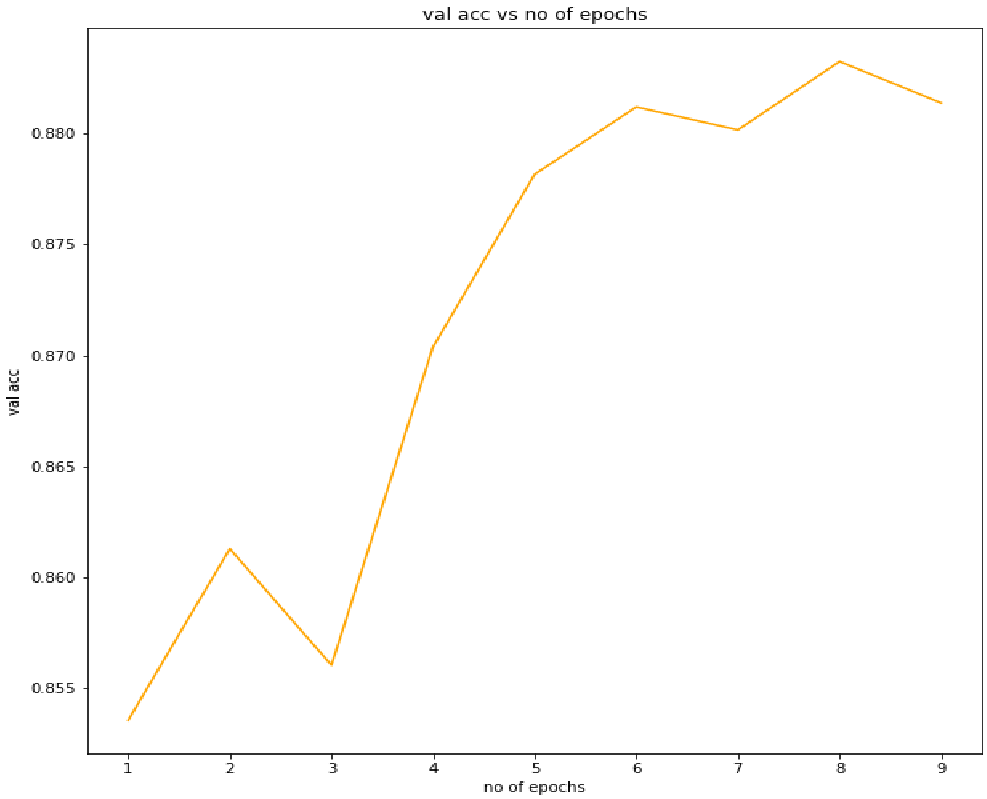


Figure: Validation accuracy vs number of epochs for NN with dense layers (Project 2)

The above figure is generated for ML Agent with dense layers for 10 epochs. The accuracy value ranges from 85% to 91% over the 10 epochs. We can also observe from the graph that the accuracy plateaus after 10 epochs.

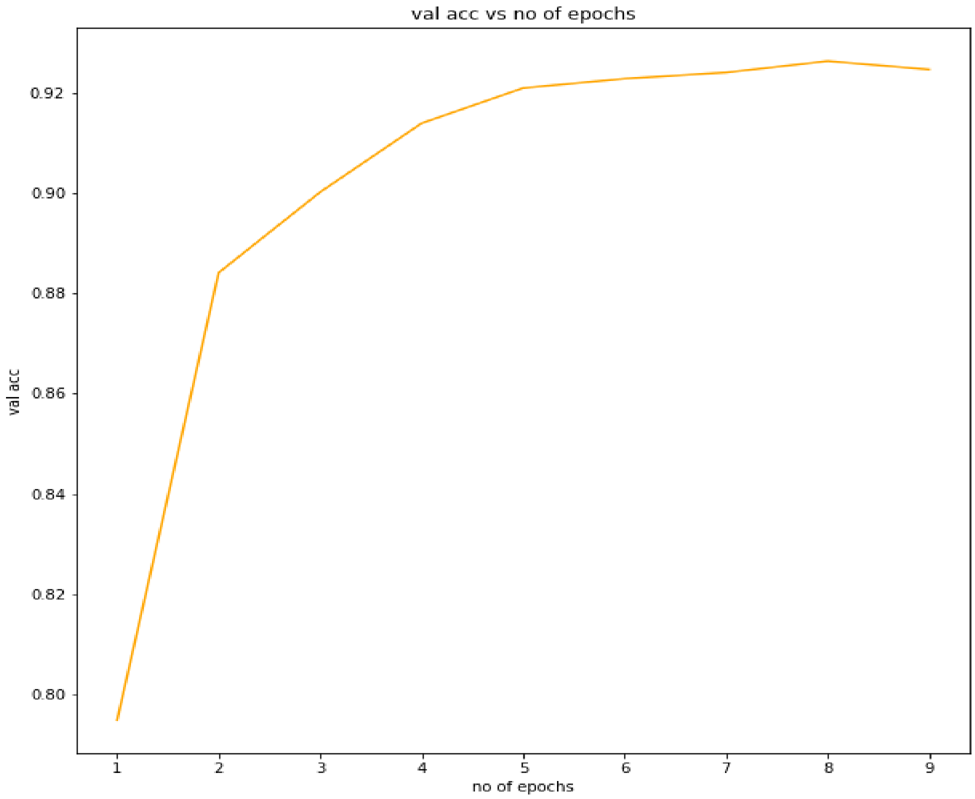
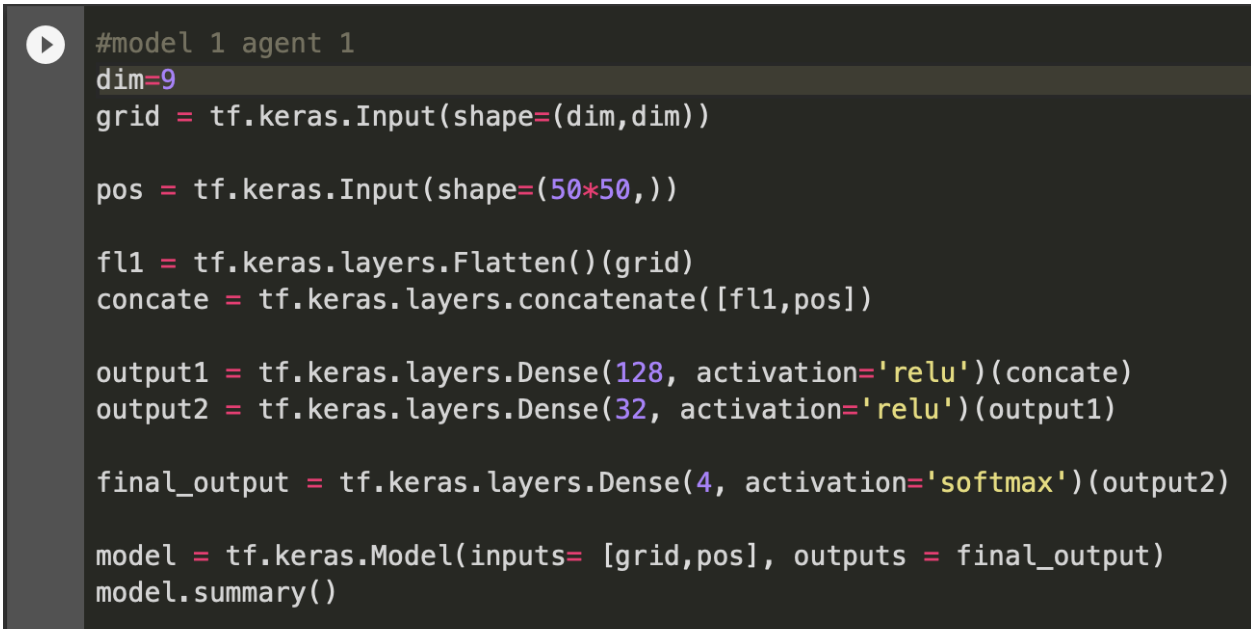


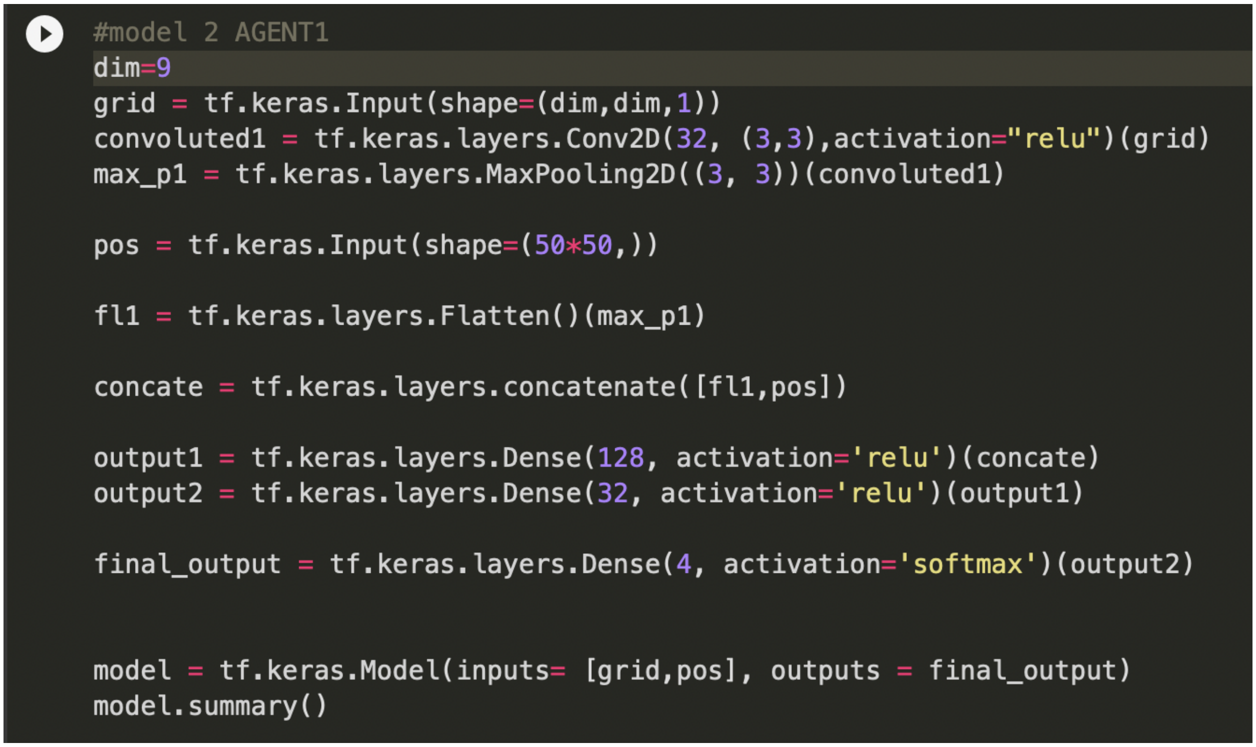
Figure: Validation accuracy vs. number of epochs for NN with convolutional layers (Project 2)

The above figure is generated for ML Agent with a Convolutional layer for 10 epochs. The accuracy value ranges from 79% to 92% over the 10 epochs. We can also observe from the graph that the accuracy plateaus after 10 epochs.

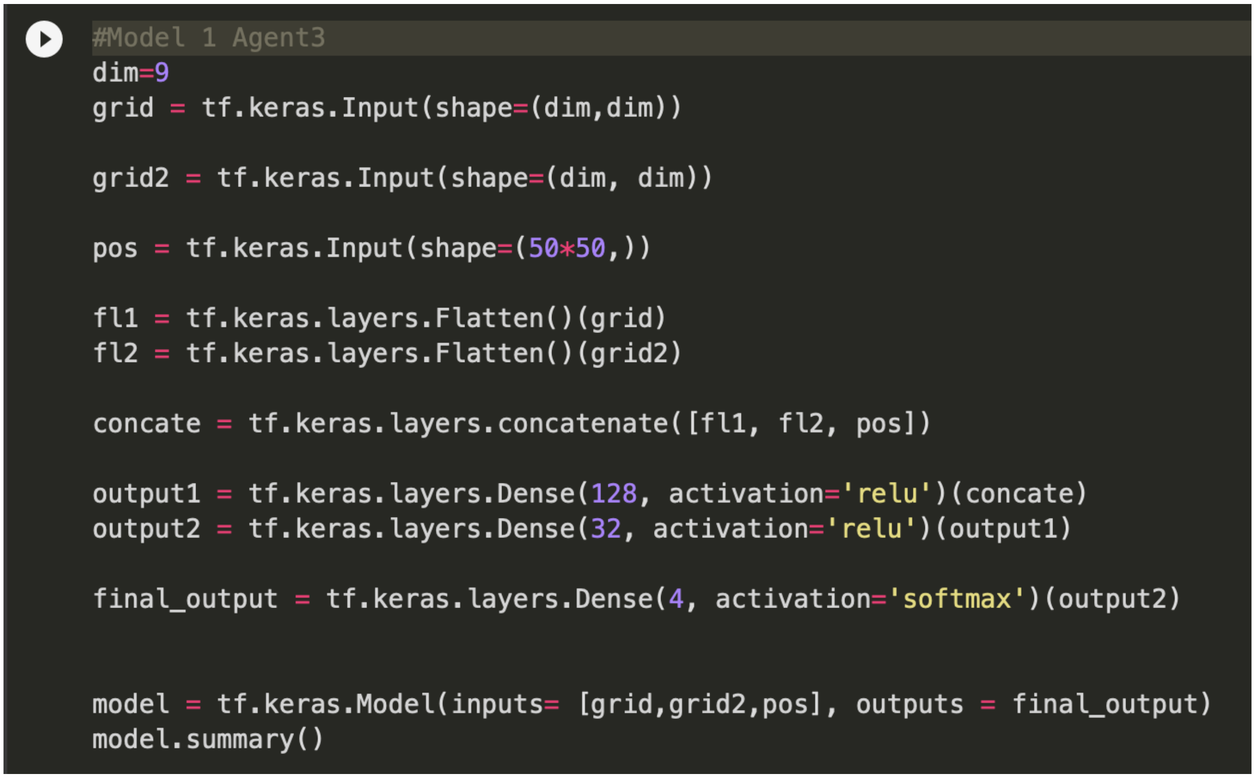
Appendix



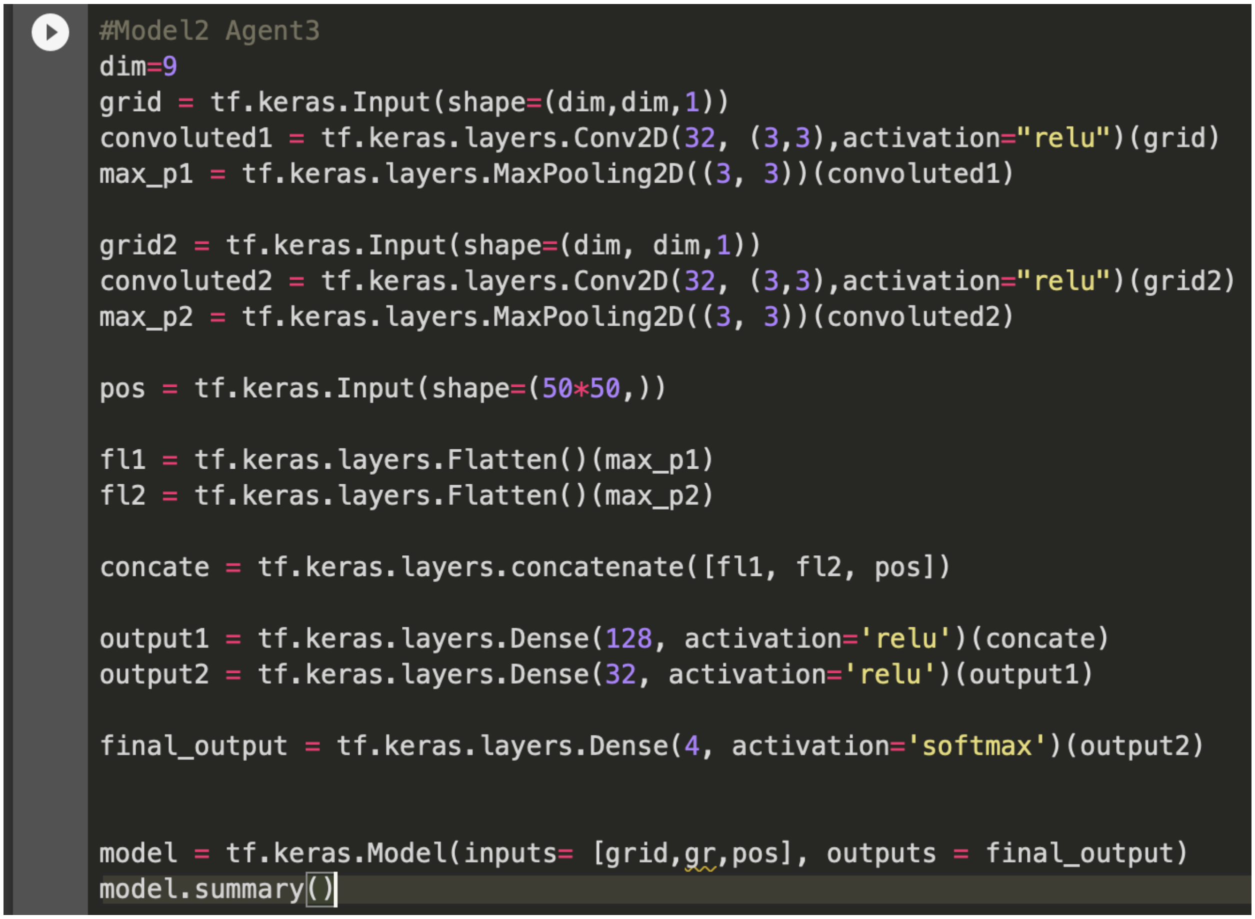
```
#model 1 agent 1
dim=9
grid = tf.keras.Input(shape=(dim,dim))

pos = tf.keras.Input(shape=(50*50,))

fl1 = tf.keras.layers.Flatten()(grid)
concate = tf.keras.layers.concatenate([fl1,pos])

output1 = tf.keras.layers.Dense(128, activation='relu')(concate)
output2 = tf.keras.layers.Dense(32, activation='relu')(output1)

final_output = tf.keras.layers.Dense(4, activation='softmax')(output2)

model = tf.keras.Model(inputs= [grid,pos], outputs = final_output)
model.summary()
```



```
#model 2 AGENT1
dim=9
grid = tf.keras.Input(shape=(dim,dim,1))
convoluted1 = tf.keras.layers.Conv2D(32, (3,3),activation="relu")(grid)
max_p1 = tf.keras.layers.MaxPooling2D((3, 3))(convoluted1)

pos = tf.keras.Input(shape=(50*50,))

fl1 = tf.keras.layers.Flatten()(max_p1)

concate = tf.keras.layers.concatenate([fl1,pos])

output1 = tf.keras.layers.Dense(128, activation='relu')(concate)
output2 = tf.keras.layers.Dense(32, activation='relu')(output1)

final_output = tf.keras.layers.Dense(4, activation='softmax')(output2)


model = tf.keras.Model(inputs= [grid,pos], outputs = final_output)
model.summary()
```



```
#Model 1 Agent3
dim=9
grid = tf.keras.Input(shape=(dim,dim))

grid2 = tf.keras.Input(shape=(dim, dim))

pos = tf.keras.Input(shape=(50*50,))

fl1 = tf.keras.layers.Flatten()(grid)
fl2 = tf.keras.layers.Flatten()(grid2)

concate = tf.keras.layers.concatenate([fl1, fl2, pos])

output1 = tf.keras.layers.Dense(128, activation='relu')(concate)
output2 = tf.keras.layers.Dense(32, activation='relu')(output1)

final_output = tf.keras.layers.Dense(4, activation='softmax')(output2)


model = tf.keras.Model(inputs= [grid,grid2,pos], outputs = final_output)
model.summary()
```



```
#Model2 Agent3
dim=9
grid = tf.keras.Input(shape=(dim,dim,1))
convoluted1 = tf.keras.layers.Conv2D(32, (3,3),activation="relu")(grid)
max_p1 = tf.keras.layers.MaxPooling2D((3, 3))(convoluted1)

grid2 = tf.keras.Input(shape=(dim, dim,1))
convoluted2 = tf.keras.layers.Conv2D(32, (3,3),activation="relu")(grid2)
max_p2 = tf.keras.layers.MaxPooling2D((3, 3))(convoluted2)

pos = tf.keras.Input(shape=(50*50,))

fl1 = tf.keras.layers.Flatten()(max_p1)
fl2 = tf.keras.layers.Flatten()(max_p2)

concate = tf.keras.layers.concatenate([fl1, fl2, pos])

output1 = tf.keras.layers.Dense(128, activation='relu')(concate)
output2 = tf.keras.layers.Dense(32, activation='relu')(output1)

final_output = tf.keras.layers.Dense(4, activation='softmax')(output2)


model = tf.keras.Model(inputs= [grid,gr,pos], outputs = final_output)
model.summary()
```

Bonus Question 3

To accomplish this, we must create an agent capable of outperforming all others. The agent must determine which action to take for each step (move left/right/up/down). For this purpose, we will train the agent, which will learn a policy (Q) which tells what is the best next move to make every step the agent takes results in a penalty or a reward (after they reach the goal). When we train the policy, these penalties and rewards are used as input. The learning algorithm modifies the action-value function Q for each state visited during training. The highest value denotes the best course of action. The reward or penalty received when the action is taken is used to update the values. A model learns at every stage of its journey, not just when it arrives at the goal.
After performing different actions, we will get different rewards according to different situations. Specifically, We can assign this reward and penalty system.

- Hit a boundary: -10
- To the goal: 50
- Hit a block: -30
- The rest: -0.1

We can use the above reward and penalty model to train our model. We are going to assign a -0.1 penalty as we want our model to reach the goal, so if it does not reach, we incur a slight penalty.

For this, we will define the Q table. In which state description of each cell is defined along with the action taken at each cell is defined. And as the agent trains, the Q table will be updated at each step.

We can use the below equation to define the policy Q:

$$Q(state, action) \leftarrow (1-\alpha)Q(state, action) + \alpha\Big(reward + \gamma \max_{a} Q(next\ state, all\ actions)\Big)$$

Where we can explore α(learning data),γ(discount rate) to get the optimum result

α :(the learning rate) should decrease as you continue to gain a larger and larger knowledge base.

γ- determines how much importance we want to give to future rewards. A high value for the discount factor (close to **1**) captures the effective long-term award, whereas a discount factor of **0** makes our agent consider only immediate reward, hence making it greedy,

Algorithm:

```
We can initialize the Q-table by all zeros.
    Start exploring actions:
For each state, select anyone among all possible actions for the current state(s).
    Travel to the next state (S') as a result of that action (a).
For all possible actions from the state (S')
    Select the one with the highest Q-value.
Update Q-table values using the equation.
Set the next state as the current state.
If the goal state is reached, then end and repeat the process.
```