COMP 532

Machine Learning & Bio-inspired Optimisation

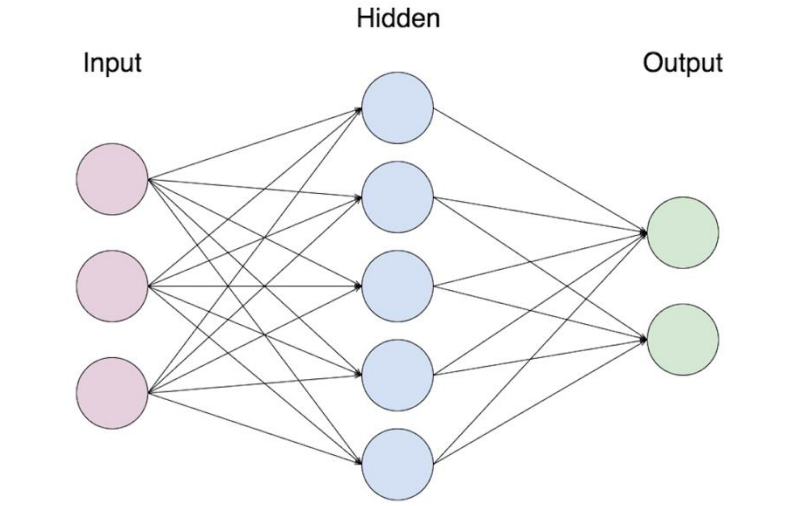
ASSIGNMENT - 2

Deshpande, Saurabh Nitin

Deep Reinforcement Learning Agent with Open AI Gym and Keras

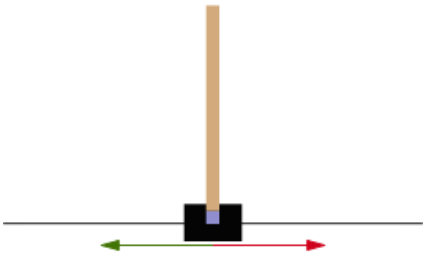
Aim:
This assignment aims to train a self-learning deep reinforcement agent for a simple video game. **Reinforcement Learning** helps us to train an agent such that it learns from previous mistakes to improve its future decisions and hence its overall performance. The agent learns from its environment by interacting with it to maximise its cumulative reward. It does this by trial and error, receiving rewards for good actions and punishment for bad decisions. For every action that an agent takes, it receives feedback from the environment. From the feedback, we can retrieve the reward (previously defined) and the next state of the environment. **Deep Reinforcement Learning** is a mix of deep learning and reinforcement learning.

Neural Network:
A neural network is an algorithm that learns from pairs of examples (input and output data), detects patterns in the data and predicts output when fed with unseen data input. The picture below shows a single hidden layer; however, we will be experimenting with different numbers of hidden layers. Since there are two buttons (0 and 1) for the game, we will have 2 nodes in the out. **Keras** is a high-level neural networks python API that makes it simple to implement a basic neural network.



3 inputs, 1 hidden layer, 2 outputs

The Cartpole Problem:
Cartpole, also known as an **Inverted Pendulum**, is a pendulum with its centre of gravity above its pivot point. It is unstable but can be controlled by moving the pivot point under the centre of mass. The goal is to keep the cartpole balanced by applying appropriate forces to a pivot point.



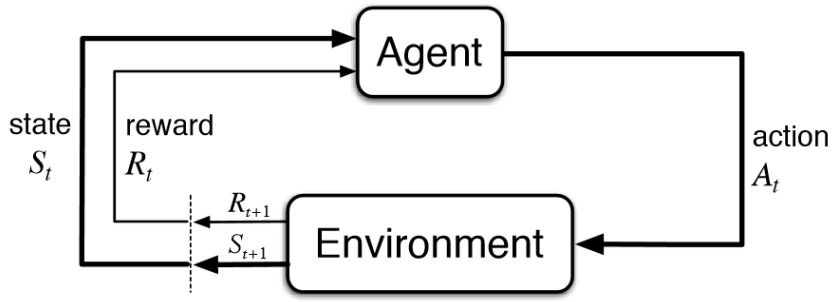
A cartpole schematic

We have chosen to apply deep reinforcement learning techniques on the CartPole-v1 game based on top of OpenAI Gym (a collection of environments to develop and test Reinforcement Learning algorithms). In this game, we have a pole standing inside a cart and the goal is to balance the pole by moving the cart from one side to another to keep the pole balanced in an upright position. The CartPole-v1 game has a maximum episode step of 500 and a reward threshold of 475. We consider the agent successful in this environment if we can balance the pole for 500 frames and a failure when the pole is either 15° away from being fully vertical or the cart slides more than 2.4 units from the centre.

How do we reward the agent? For every frame that the agent moves and the pole is balanced (i.e., not more than 15° away from being vertically upright and not more than 2.4 units from the centre), the agent gets a score of +1. The aim is to get up to +500 points.

Deep Q Network Architecture:
In the Deep Q Network algorithm, a neural network is used to perform the best action based on the environment (state). We have a function called the Q function used to estimate a potential reward based on a state. Q (state, action) helps us to calculate the expected future value based on the state and action.

To achieve this optimal behaviour, we can apply deep reinforcement learning and we will be using the Deep Q-Network model architecture. For our implementation, we used **Keras** and **Gym** libraries. We chose to apply this to the CartPole game. The CartPole game is a simple environment in the OpenAI gym collection. It is built on the Markov Chain model like below:



Markov Chain – 'SARSA'

We start with an initial environment.

Importing the environment

```
# initialize network class
def __init__(self):

    # prepare the OpenAI Cartpole-v1 Environment
    self.env = gym.make(AGENT_ENV)
```

At this point, it does not have any associated rewards, but it has a state (S_t). For every iteration through the environment, the agent takes a current state (S_t) and predicts the best action (A_t) before performing the action in the environment. The environment in turn gives a reward ($R_t + 1$) for the action, the next state ($S_t + 1$) and information telling us if the new state is terminal. This iteration continues until it is terminated.
The line of code below helps us to retrieve this information:

```
nextState, reward, terminal, _ = env.step(action)
```

We know that our goal is to balance the pole on the moving cart. For this particular problem, instead of pixel information or image sequences, the state gives us information about the angle of the pole and the position of the cart. The agent actions in moving the cart are mapped to values 0 or 1 depending on whether the agent wants to move the cart to the left or right. (i.e., 0 for left, 1 for right).

```
Action:
    Type:   Discrete(2)
    Num     Action
    0       Push cart to the left
    1       Push cart to the right
```

When we pass the action (0 or 1), the game environment returns the results for that step. The **terminal** flag tells us whether or not the game has ended (i.e., if a state is in the final state). The **next_state** variable carries the information (possible state values) below:

```
Observation:
    Type:   Box(4)
    Num     Observation               Min             Max
    0       Cart Position             -4.8            4.8
    1       Cart Velocity             -Inf            Inf
    2       Pole Angle                -24°            24°
    3       Pole Velocity At Tip      -Inf            Inf
```

To train the agent, we need the old state, the action, the next state and the reward.

Run

To run this simulation, first, we initialise the CartPole-v1 environment then we go through different episodes of the gameplay. For each episode of the game, we have to reset the environment i.e., return to a state of zero before starting (set its initial state). Because the limit of frames is 500, we have limited the maximum number of frames to iterate through to 500. We render (display) the CartPole game on the screen. Next, we select a random action space, either 0 (left) or 1 (right). Based on the action space, the agent performs an action and after every action, we receive the next state, the associated reward and also a flag that tells us if the action has been completed or not – (state, action, reward, nextState, terminal).

If we play the game at this point, we can see that the agent tries to balance the pole but did not get a really good score. But since it's a game with random moves, it is a great starting point.

The Neural Network

The Neural Network algorithm learns from various examples, detects the patterns in the dataset and predicts the output of an unseen input data. We have used **Keras** to implement a neural network.

First, we create an empty neural network. Loss and optimizer are basic parameters that define the characteristics of our Neural Network. For our neural network to learn and predict from environment data, we initialise our model and then feed it with the environment data. When we call the **fit( )** function, the model trains on our input and output pair. When the model is trained on those data, it can then predict an output based on unseen input data.

```
model.fit(state, target)
```

The line of code above helps us to train the network.

```
model.predict(state)
```

The line of code above helps us predict reward from a current state using the model previous trained from our initial input data.

In our experiment, we have used a 3-layer network, a 2-layer network and tweaked its structure and parameters testing different values.

- **3 layers:** 512, 256, 64 neurons
- **2 layers:** 64, 24 neurons

We will discuss this shortly.

The Deep Q Network – DQN

The Deep Q Network is a Reinforcement Learning technique aimed at choosing the best action for given circumstances (observation). Each possible action for each possible observation has its Q value where Q stands for the quality of a given move.

In a typical game, what we are usually after is that we maximize our total score. This is our reward. Coming back to our CartPole example, if for example the pole is tilted at an angle to the right. The expected future reward of moving it to the cart to the right will be higher than if we decide to move the cart to the left. The longer we can keep the pole balances the higher our score.

We need to find a way to represent this idea as a formula that we can optimize. The loss value indicates how far our prediction is from the actual target. The prediction of our model, for example, could be showing us that it sees more value from moving the cart to the left when it can gain more reward by moving the cart to the right. We aim to reduce this loss (the gap between the prediction and the target values).

The loss function is defined as seen below:

$$loss = (r + \gamma max Q'(s, a') - Q(s, a))^2$$

First, we carry out an action and then observe the reward r and the resulting new state, s. From this result, we then calculate the maximum target, Q and reduce it by a discount so that the future reward is worth less than the immediate reward. Finally, we add the current reward to the discounted future reward to get our target value. To get the loss, we will subtract our current prediction from the target. If we square this, we can punish large loss values more and treat positive values the same way as the negative values.

Formal notation:

$$Q^{new}(s_t, a_t) \leftarrow (1 - \alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

We can implement this formula using **Keras**.

```
target = reward + gamma * (np.amax(model.predict(nextState))
```

The line of code above helps us to retrieve our target using the formula above. If a learning rate is not defined when the model is initialized, the model will internally define one by itself. The gap between our target and the predicted value is decreased by the learning rate. As we keep repeating the update process, the Q-value converges to the true Q-value. Thus, the loss reduces, and the score increases.

How do we end up with accurate Q values?
This is where deep neural networks and linear algebra comes in. For each state experienced by our agent, we are going to remember it:

```
remember(state, action, reward, nextState, terminal)
```

And then perform and experience replay.

```
replayBuffer()
```

Key features of the deep Q Network are the remember and replay functions.

Remember

In the deep Q Network, the neural network is designed to forget the previous experiences because it overwrites them with new experiences. Through the biologically inspired experience replay process we can uniformly sample experiences from the memory and update the Q-value of each entry. This helps to minimize correlation between subsequent actions. Because of this, we create a list implementation of the memory, to store previous experiences and observations. This is needed to retrain the model with previous experiences.
In our remember function, we pick the state, action, reward and next state and add them as an entry inside our memory list. The memory list will look like below:

```
memory = [(state, action, reward, nextState, terminal)]
```

See remember function code below:

```
def remember(self, state, action, reward, nextState, terminal):
    self.memory.append((state, action, reward, nextState, terminal))
    if len(self.memory) > self.trainStart:
        if self.epsilonMax > self.epsilonMin:
            self.epsilonMax *= self.epsilonDecay
```

For the agent to perform well in the long term, we need to take into consideration the future rewards and not just immediate rewards. To do this, we add the discount rate (gamma) to the current state reward. This is to force the agent to learn to maximize the discounted future reward based on the given state (i.e., update Q-value with the cumulative discounted future rewards).

The memory in our implementation is designed to store 2000 samples (moving average memory). This is a deque implementation such that when we add a new instance to the memory, the first sample in the memory gets pushed away. We are storing the state, action, reward, and completion.

Epsilon – is the probability of taking a random action. During training, this epsilon decreases as we update it with the epsilon decay.

Replay

This function trains the neural network with the experiences we have in the memory. We sample some experiences from the memory. We can refer to this as minibatch with a specific batch size. Should the batch size be less than the size of our memory list, the minibatch will contain every instance in the memory.

How can the network converge given that it's predicting its own input? Because we are testing a simple implementation and we are using deep Q N, it converges.

```
miniBatch = random.sample(memory, min(len(memory), batchSize))
```

We keep playing random games until we reach the minimum required memory to keep training the model.

We initialize state and next_state as NumPy arrays.
We also create three lists of action, reward, terminal.

We loop through the memory to sample all the data in our miniBatch and we run a prediction on the whole batch to get our targets and next targets. Then we apply the reward function of the Deep Q Network.

How do we set our Hyperparameters?

These parameters are passed to the reinforcement learning agent.

- **episodes** - number of games we want the agent to play
- **gamma** - decay or discount rate, to calculate the future discounted reward
- **epsilon** - exploration rate, this is the rate in which an agent randomly decides its action rather than prediction.
- **epsilonDecay** - we want to decrease the number of explorations as it gets good at playing games.
- **epsilonMin** - we want the agent to explore at least this amount.
- **learningRate** - determines how much neural net learns in each iteration (if used).
- **batchSize** - determines how much memory DQN will use to learn.

The act function returns the act. Depending on the size of our epsilon our action will be random or predicted from the model.

The **train( )** function helps us to train our deep Q network model using reinforcement learning step by step. We render every step using env.render( ). As long as the terminal flag is False, we keep training the model. The result from every step is stored in the memory list, we then use this for training on every step. For the training example in the image below, the algorithm reached convergence at the 90^{th} episode with a score of 500.

```
episode:  71/1000, score: 227, e: 0.19
episode:  72/1000, score: 269, e: 0.15
episode:  73/1000, score: 248, e: 0.11
episode:  74/1000, score: 275, e: 0.087
episode:  75/1000, score: 226, e: 0.069
episode:  76/1000, score: 285, e: 0.052
episode:  77/1000, score: 200, e: 0.043
episode:  78/1000, score: 273, e: 0.032
episode:  79/1000, score: 342, e: 0.023
episode:  80/1000, score: 265, e: 0.018
episode:  81/1000, score: 265, e: 0.014
episode:  82/1000, score: 251, e: 0.011
episode:  83/1000, score: 263, e: 0.0081
episode:  84/1000, score: 489, e: 0.005
episode:  85/1000, score: 317, e: 0.0036
episode:  86/1000, score: 343, e: 0.0026
episode:  87/1000, score: 249, e: 0.002
episode:  88/1000, score: 293, e: 0.0015
episode:  89/1000, score: 482, e: 0.001
episode:  90/1000, score: 500, e: 0.001
```

Sample training results

```
episode:   6/1000, score: 425
episode:   7/1000, score: 401
episode:   8/1000, score: 276
episode:   9/1000, score: 453
episode:  10/1000, score: 303
episode:  11/1000, score: 257
episode:  12/1000, score: 252
episode:  13/1000, score: 297
episode:  14/1000, score: 286
episode:  15/1000, score: 426
episode:  16/1000, score: 274
episode:  17/1000, score: 304
episode:  18/1000, score: 500
```

Sample testing results

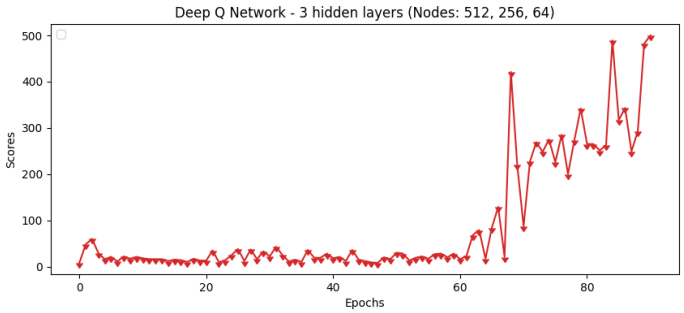
Experiments

In our code, we are running 1000 episodes to train the game. When the model reaches the 500 score, we save it so we can then use it to test. Once we start training, we can observe that our epsilon continues to decrease with each episode. After the 500 score, it quits. Sometimes, training is faster. Sometimes it is slower (i.e., the model converges faster sometimes and sometimes it takes a long time to get to convergence). For this reason, we train the model multiple times until our test predicts good values.

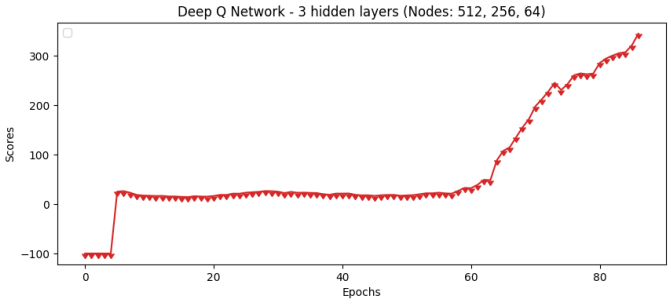
```
Max. Episode Steps: 500
Reward Threshold: 475.0
Model: "CartPole_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 4)]               0
_________________________________________________________________
dense (Dense)                (None, 512)               2560
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_2 (Dense)              (None, 64)                16448
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 130
=================================================================
Total params: 150,466
Trainable params: 150,466
Non-trainable params: 0
_________________________________________________________________
```

Model Summary for a Network with 3 hidden layers (Nodes: 512, 256, 64)

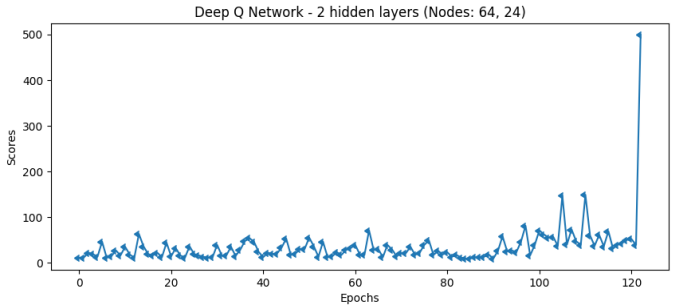
Experiment 1 - (3 Hidden Layers vs 2 Hidden Layers)



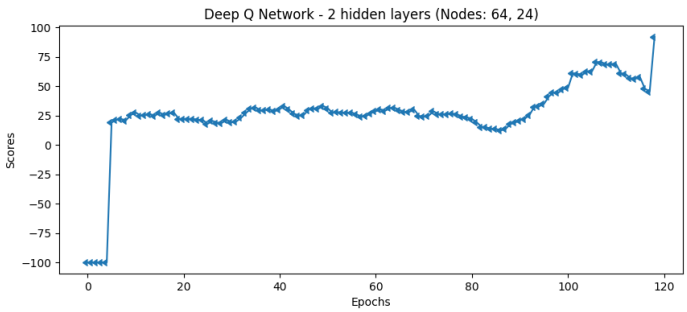
Plot of Scores against Epochs (512, 256, 64)



Plot of Scores (Moving Average) against Epochs (512, 256, 64)



Plot of Scores against Epochs (64, 24)



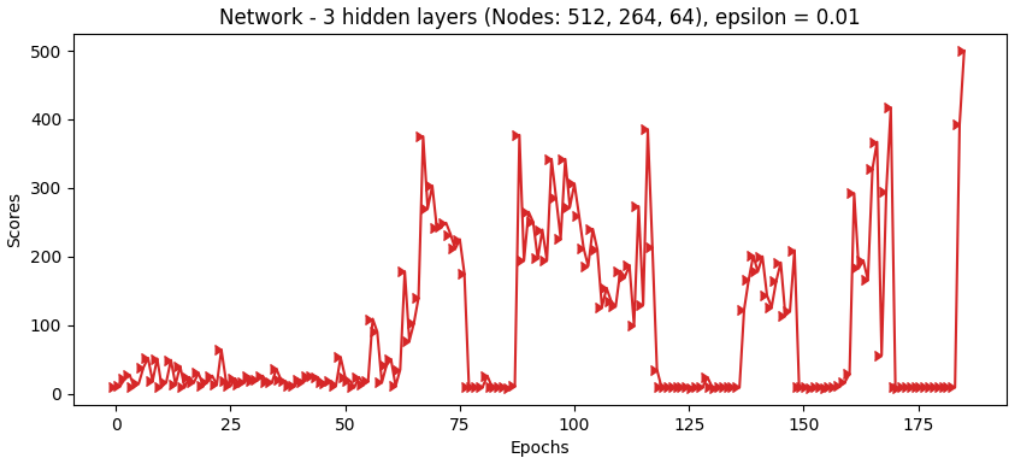
Plot of Scores (Moving Averages) against Epochs (64, 24)

From this experiment, we can observe that when we use 3 hidden layers, the algorithm reached convergence faster. But would we conclude that it is best to use 3 hidden layers since it performs better? The answer is **NO.** This is because when we ran multiple experiments with both 2 hidden layer and 3 hidden layer networks, we observed that sometimes the convergence can be achieved earlier, in other experiments it can take a longer time irrespective of the number of hidden layers. Therefore, the impact of the number of layers is not easily noticed for the CartPole game learning problem. This experiment was done with an epsilon value of 0.001.

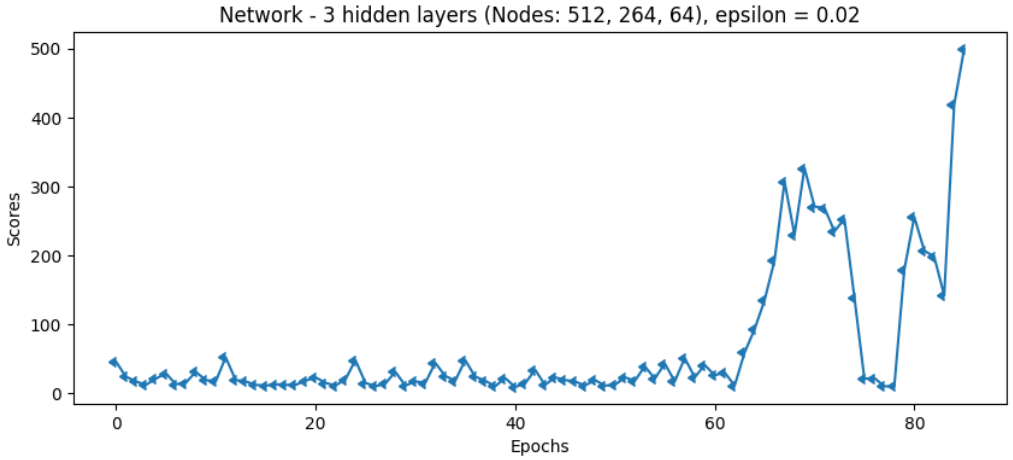
Experiment 2 - (epsilon – 0.001, 0.01, 0.02, 0.1, 0.5)

Please note that the previous experiment was done with an epsilon value of 0.001. So, we also performed experiments for epsilon values of 0.01, 0.02, 0.1 and 0.5 while keeping the number of layers **fixed at 3 layers (512, 256, 64 neurons)**.

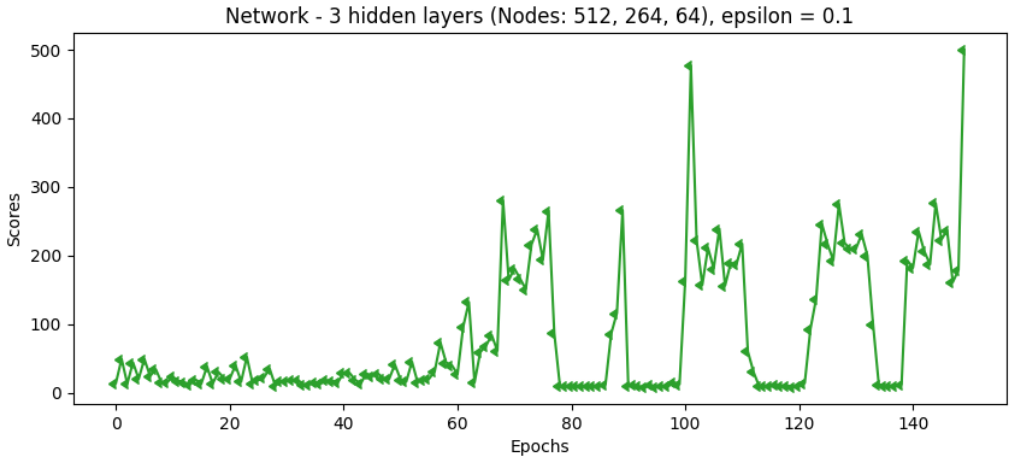
epsilon 0.01



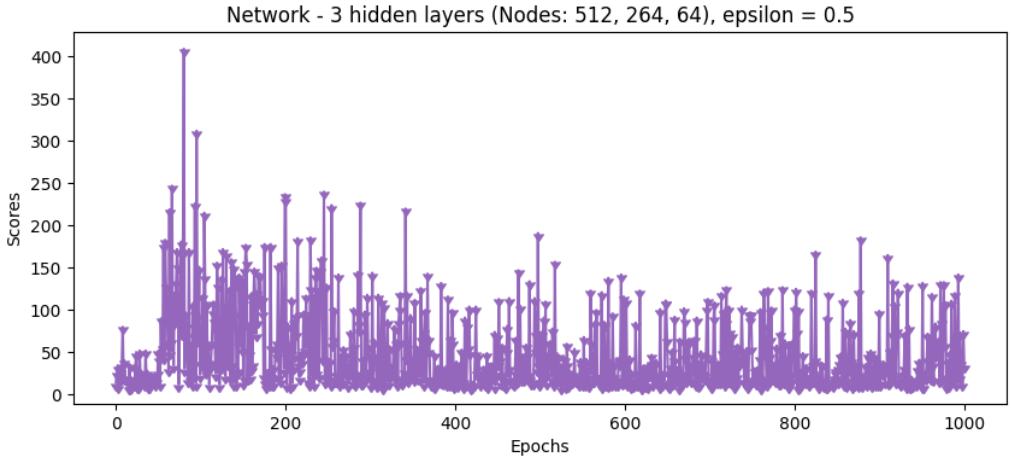
epsilon 0.02



epsilon 0.1



epsilon 0.5



Comparing these results, it is evident that for small initial epsilon values, the agent learns faster. That is, with increasing epsilon values, the agent took much longer time to reach convergence (or may not reach convergence at all as is the situation when epsilon value = 0.5). We can therefore deduce here that the epsilon value used during training impacts how fast our agent learns.

Conclusion

In conclusion, it is worth noting that the Deep Q network is not guaranteed to converge every time because of the instability caused by bootstrapping, sampling, and value function approximation. Also, CartPole can be considered a relatively difficult environment for a deep Q network to learn. As observed, the learning often fails and the results from testing above were achieved upon several attempts. Sometimes you may be lucky to achieve convergence earlier.

Also, we noticed that to achieve fast learning, we can use a small initial epsilon value or a fast decay rate. An epsilon value of 0.99 means the policy will render random actions for a long time before

epsilon is decayed to a small value (especially if you use a slow decay rate). The downside to this is that a small epsilon value might cause the learning to be suboptimal due to lack of exploration.

It is also worthy to note that the Deep Q Network is an off-policy algorithm and Q values can converge even with random policies if the Robbin-Monro algorithm is satisfied and all states and actions have been visited an infinite number of times i.e., the Q values might have already converged and ready to be applied to make predictions after many epochs, even if the rewards do not look perfect.

To improve these results, we can explore more model structures (more layers, other activation functions) and hyperparameter settings (experience size, epsilon decay values, etc.)

Ultimately, the deep Q network is successful in improving the decisions of the agent.