Machine Learning Project Report | BITS F464

Avishree Khare 2017A7PS0112G Parantak Singh 2017A7PS0109G

Description

The given problem had three tasks which provided gradually increasing modifications to the existing CartPole-v1 environment. The first task had variations in gravity and friction. The second task included noisy controls and the third task included noisy observations of pole angles apart from the previous modifications. Hence, the difficulty increased with each task with the third task being the most unpredictable.

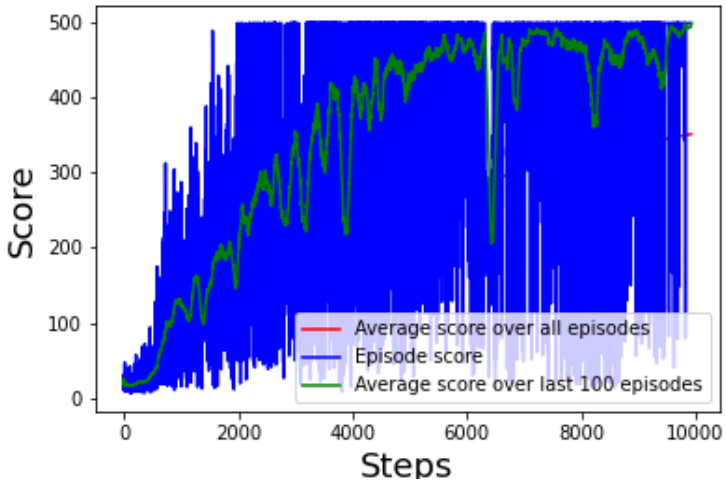
We decided to try our slants on the third task as this inevitably ensured similar if not better performance on the remaining two tasks. The convergence goal set for our models was to achieve a mean score of 500 over the last 100 episodes of their training. We implemented REINFORCE, i.e. the Vanilla Policy Gradient (VPG) method using PyTorch and the Deep Q-Network (DQN) models using Keras. The Reinforcement Learning (RL) algorithms we implemented have been listed and described along with their corresponding shortcomings in the sections below.

REINFORCE (VPG)

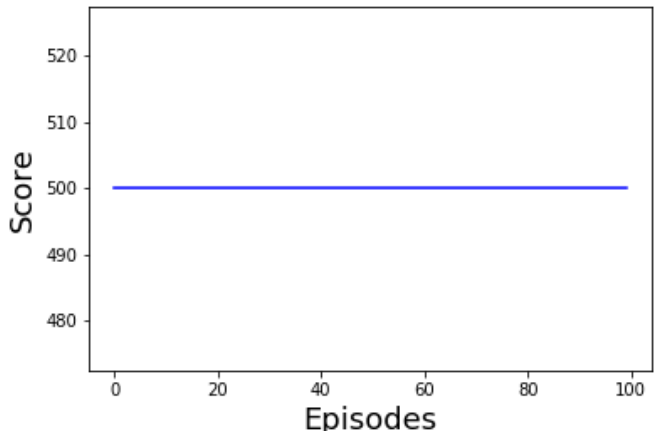
We initially implemented the most elementary policy gradient method, commonly referred to as the Vanilla Policy Gradient. We tried the classical REINFORCE[1] algorithm ensuing PyTorch's implementation of the same [4]. The policy network had 2 hidden layers of size 32 each. ReLU was used as the activation for the intermediate layers and SoftMax for the last layer. Adam was chosen as the optimizer with a learning rate of 0.001. The loss for an episode was calculated as the sum of the products of rewards and log-probabilities of the actions. The summary of the model is as follows:

```
Agent(
  (model): Sequential(
    (0): Linear(in_features=4, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=32, bias=True)
    (3): ReLU()
    (4): Linear(in_features=32, out_features=2, bias=True)
    (5): Softmax(dim=1)
  )
)
```

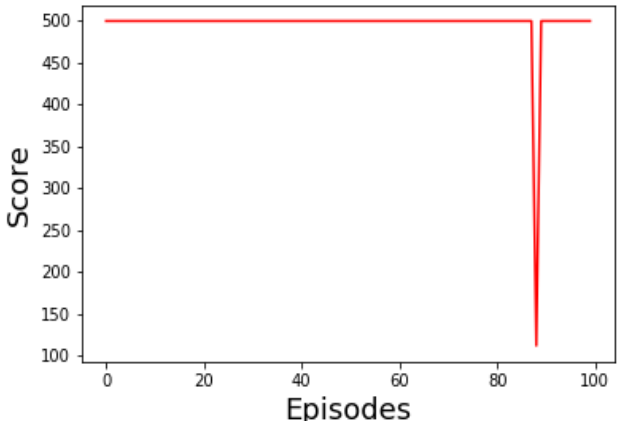
The agent took 9924 episodes to report an average score of 500 over the last 100 episodes. The training graph is as follows (Here, steps == episodes):



For Task 1 and Task 2, the model gave an average score of 500 over the last 100 episodes during evaluation. The evaluation graph is as follows:



On evaluation for Task 3, the model gave an average score of 496.1 over the last 100 episodes. The evaluation graph is as follows:



We, however, decided to try DQN models because of the following reasons:

a. As is evident from the training graph, the scores from REINFORCE were inconsistent and varied widely throughout. This also reflected during the evaluation as the model gave different results in each trial for Task 3.
b. The model took around 10000 episodes to converge to our requirements results and we wished to lower the number.
c. The policy network any smaller would require an even greater of episodes to converge which was undesirable.

DEEP Q-NETWORKS (DQN)

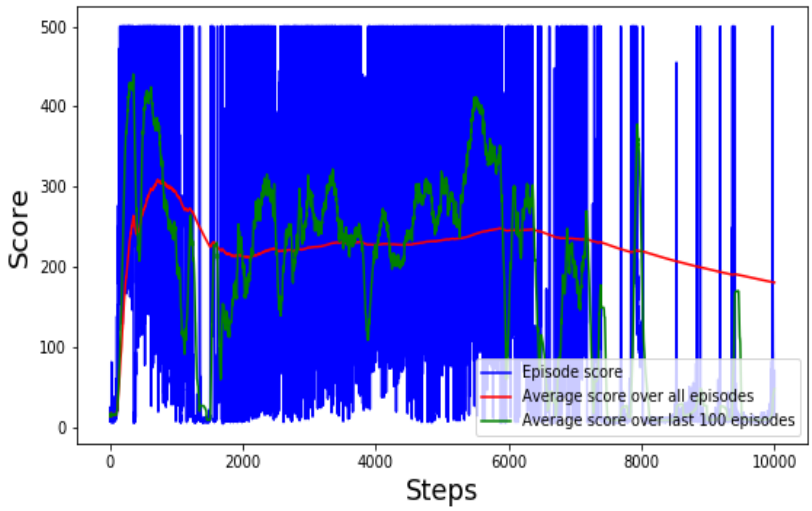
We then implemented the DQN algorithm, which aims at scaling the standard Q table by accounting for the infinitesimally large number of states and estimating the Q-values using neural networks. We implemented the neural network using Keras. The neural network had 2 hidden layers of size 32 each. ReLU was used as the activation for the intermediate layers and a linear layer was used as the activation function for the last layer. "He uniform" initializer was used to initialize the weights of the network Adam was chosen as the optimizer with a learning rate of 0.001. The loss for an episode was calculated as the squared error between the prediction and the target.

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

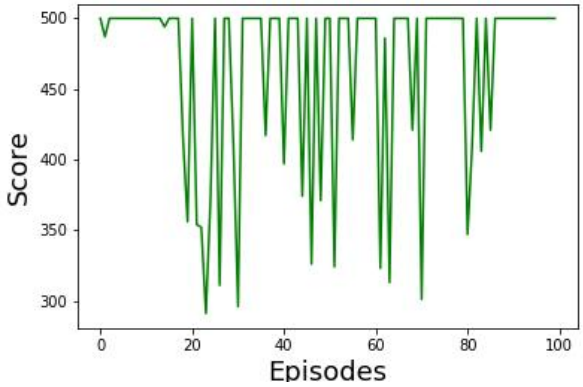
The summary of the model is as follows:

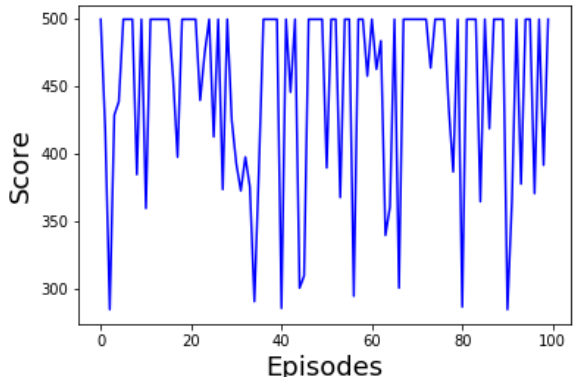
```
Model: "cartpole-v2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_9 (InputLayer)         (None, 4)                 0
_________________________________________________________________
dense_25 (Dense)             (None, 32)                160
_________________________________________________________________
dense_26 (Dense)             (None, 32)                1056
_________________________________________________________________
dense_27 (Dense)             (None, 2)                 66
=================================================================
Total params: 1,282
Trainable params: 1,282
Non-trainable params: 0
```

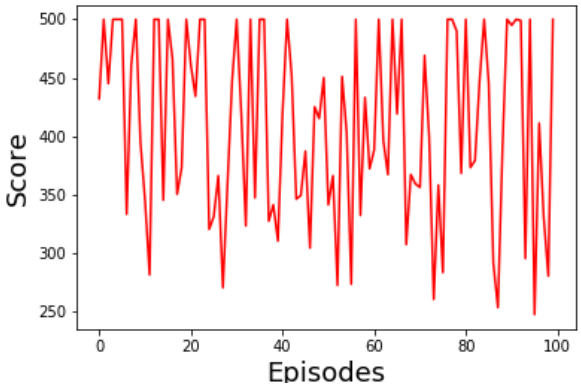
The agent did not converge over 10000 episodes to give an average score of 500 over the last 100 episodes. The evaluation graph is as follows:



The evaluation graphs for Task 1, Task 2 and Task 3 are as follows:







As we can evidently observe, Task 1 and Task 2 performed better than Task 3 for the same model that had been trained on the environment on Task 3. Besides, none of the tasks managed to attain a mean of 500 over their run of 100 episodes. In hindsight, we could've achieved a perfect score for all the evaluations had we set a weaker exit condition for the agent. However, the weights assigned to the model in that case would be more of an opportunistic coincidence than a systematic convergence.

We, decided to further explore the avenues of Deep Q-Learning and move onto Double Deep Q-Learning given the rather random performance of the DQN model.

DOUBLE DEEP Q-NETWORKS (DDQN)

We further implemented the DDQN algorithm; the algorithm's core structure is the same in the sense that it too uses a neural network to predict Q-values. The difference lies in the fact that a DDQN uses two identical neural net models. The first one is often referred to as an online network and is no different from a basic DQN, while the other one, known as the target model is a replica of the last episode's DQN.

We build the target model to prevent either of the actions from gaining immutable advantage upon being trained by the DQN. Hence, if we use Q-values from the previous episode, we could essentially reduce the difference between the output values for the 2 actions, giving each action a fairer chance at being opted for.

Standard DQN:

$$value = reward + discount\_factor * max(target\_network.predict(next\_state))$$

DDQN:

$$value = reward + discount\_factor * TN.predict(next_{state})[argmax(ON.predict(next_{state}))]$$

ON = ONLINE NETWORK

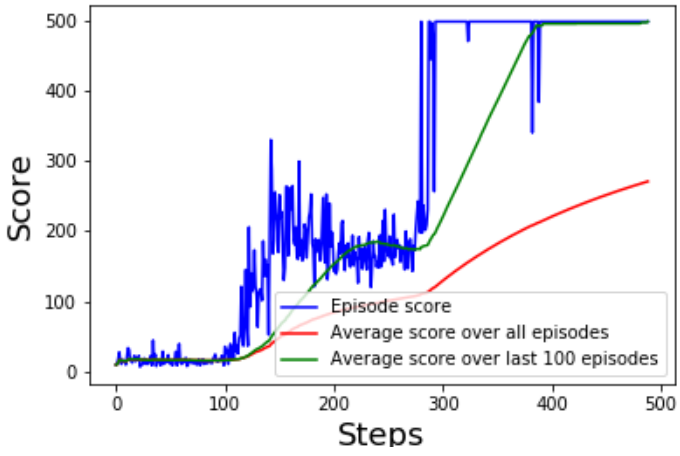
TN = TARGET NETWORK

We implemented the neural network using Keras. Both the neural networks had 2 hidden layers of size 32 each. ReLU was used as the activation for the intermediate layers and a linear layer was used as the activation function for the last layer. "He uniform" initializer was used to initialize the weights of the network Adam was chosen as the optimizer with a learning rate of 0.001 and mean squared error was used as the loss function.

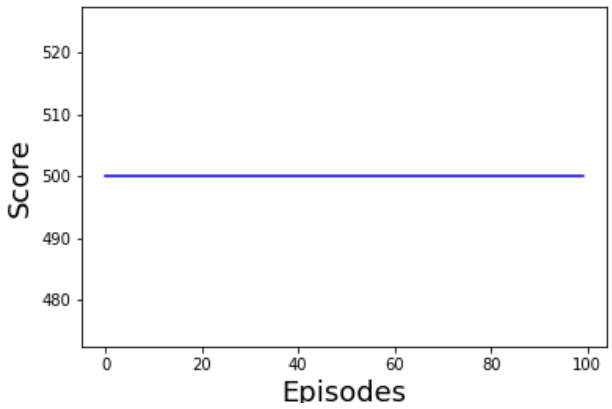
The summary of the model is as follows:

```
Model: "cartpole-v2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_11 (InputLayer)        (None, 4)                 0
_________________________________________________________________
dense_31 (Dense)             (None, 32)                160
_________________________________________________________________
dense_32 (Dense)             (None, 32)                1056
_________________________________________________________________
dense_33 (Dense)             (None, 2)                 66
=================================================================
Total params: 1,282
Trainable params: 1,282
Non-trainable params: 0
_________________________________________________________________
Model: "cartpole-v2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_12 (InputLayer)        (None, 4)                 0
_________________________________________________________________
dense_34 (Dense)             (None, 32)                160
_________________________________________________________________
dense_35 (Dense)             (None, 32)                1056
_________________________________________________________________
dense_36 (Dense)             (None, 2)                 66
```

The agent took 489 episodes to report an average score of 500 over the last 100 episodes. The training graph is as follows (Here, steps == episodes):



The evaluation graphs for Task 1, Task 2 and Task 3 is as follows:



As observed, the model evaluated to a perfect 500 for all three tasks, while simply being trained on the task 3 environment. Despite getting perfect results, we sought to make our model smaller, and we thereby implemented soft updates with DDQN as a fall back for reduced unpredictability.

DDQN with soft updates

After trying DDQN, we came across the soft-update based variant of the algorithm. Here, instead of completely changing the weights of the target model to those of the current model, the weights are updated gradually to prevent steep changes. The soft update parameter (tau) decides the extent to which the weights need to be changed.

Let *target_weights*, *current_weights* denote the weights of the target model and the current model respectively. Let *tau* denote the soft update parameter. Then the update is made as follows:

$$target\_weights = (1 - tau) * target\_weights + tau * current\_weights$$

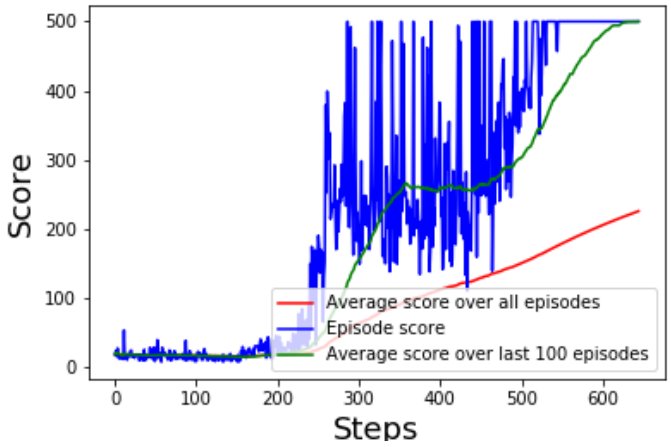
We tried various values of *tau* and found that *tau* = 0.9 worked well and resulted in the best results we obtained for the given tasks. Hence, the target model was updated largely to the current model except for a small fraction of its old weights which were retained.

We tried different sizes of the policy network and found that all the models converged using DDQN with soft update (tau=0.9).

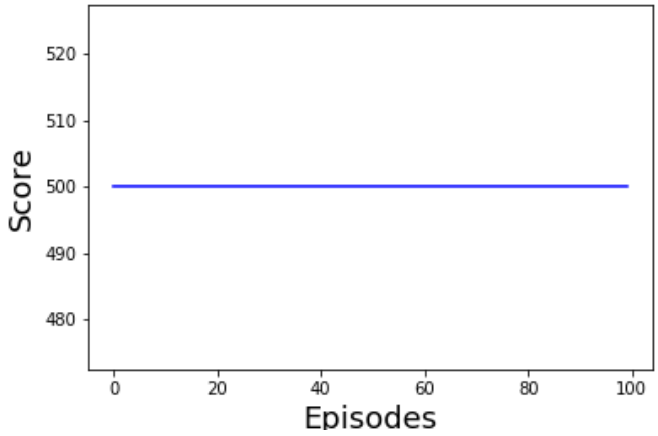
The smallest model was found to have 2 hidden layers of size 4 each with ReLU activation. “He uniform” initializer was used to initialize the weights of the network. Adam was used as the optimizer with a learning rate of 0.001 and Mean squared error was used as the loss function. The summary of the model is as follows:

```
Model: "CP_task3_DDQN"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 4)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 20
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 20
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 10
=================================================================
Total params: 50
Trainable params: 50
Non-trainable params: 0
_________________________________________________________________
```

The agent took 644 episodes to report an average score of 500 over the last 100 episodes. The training graph is as follows (Here, steps == episodes):



On evaluation, the model gave an average score of 500 over the last 100 episodes for all the three tasks. The evaluation graph is as follows:



Contributions

We divided our work equally throughout the project. We decided the approaches, wrote them individually and discussed the results with each other later. Finally, it would suffice to say that we helped each other understand the approaches we tried, discussed the results and problems, and took decisions together to arrive at common conclusions.

References/Resources:

[1] Williams, R. J. *Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach. Learn.*, 1992

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra M. Riedmiller *Playing Atari with Deep Reinforcement Learning*, 2013

[3] H. V. Hasselt, A. Guez, D. Silver, *Deep Reinforcement Learning with Double Q-learning*, 2015

[4] PyTorch REINFORCE (examples/reinforce.py at master · pytorch/examples)

[5] RL tutorials by Python Lessons (https://pylessons.com/CartPole-reinforcement-learning/)