CS4287

DQN - Cartpole

Team members:

Oliver Nagy / 19269749
Emmet Browne / 18238637
Robert Mooney / 19278772

Link to notebook
:https://colab.research.google.com/drive/1APK9Z02JKpvKfBNMV13pTeFnEcl3vXNc#scrollTo=T_ov57WAgob4

Table of Contents

1. Why we chose Reinforcement Learning:

Reinforcement learning ~ “A machine learning method that rewards desired behaviours and punishes undesired ones”

Types of Learning paradigms that could be considered for this problem:

- **Supervised learning:** In supervised learning, an algorithm is trained on a labelled dataset that consists of input-output pairs. The algorithm learns to map the inputs to the corresponding outputs by minimising some measure of error. This approach could potentially be used to solve the Cartpole problem by training a supervised learning algorithm on a dataset of states and actions that result in successful balancing of the pole.

  <u>However, this approach would require a large amount of labelled data and may not be able to adapt to changes in the environment.</u>

- **Unsupervised learning:** In unsupervised learning, an algorithm is trained on an unlabeled dataset and learns to identify patterns and structures in the data. This approach could potentially be used to solve the Cart Pole problem by training an unsupervised learning algorithm on a dataset of states and actions that result in successful balancing of the pole. The algorithm could then learn to identify patterns in the data that correspond to successful balancing and use these patterns to guide its actions.

  <u>However, this approach may not be able to learn a control policy that is specifically designed to maximise long-term reward.</u>

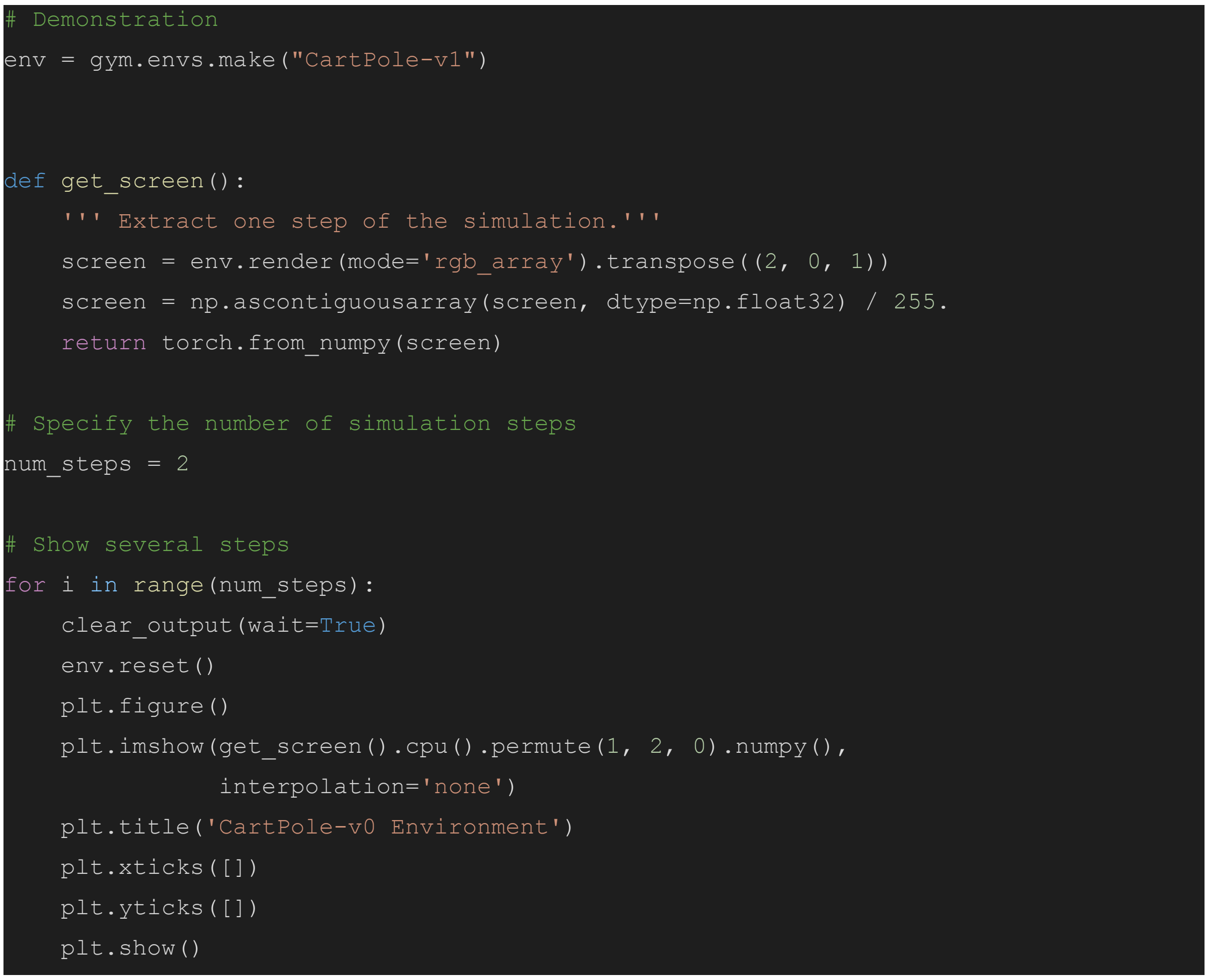
- **Reinforcement learning** is the machine learning paradigm of choice for the Cartpole problem because it is well-suited to solving problems that involve making decisions in complex, dynamic environments. In the case of the Cartpole problem, reinforcement learning can be used to train an agent to balance a pole on a moving cart by taking a series of actions that will maximise some notion of long-term reward.

  As reinforcement learning algorithms can learn to optimise control policies directly from raw sensory inputs, they can be an effective way of solving control problems like the Cartpole problem without the need for a detailed mathematical model of the system. Additionally, reinforcement learning algorithms can learn from experience, which means that they can improve their performance over time as they interact with the environment. This is particularly useful for the Cartpole problem, where the optimal control policy is likely to depend on the specific details of the environment and the initial conditions of the system.

2. The Environment:

A pole moves down a frictionless track in the CartPole environment. Applying a force of +1 or -1 to the cart regulates the system. The objective is to keep the pendulum from toppling over since it starts off upright. Cart position, cart velocity, pole angle, and pole tip velocity are the four values that make up the state space. Going left or moving right are the only two options available in the action space.

For each timestep that the pole stays upright, a reward of +1 is given. When the cart moves more than 2.4 units away from the centre or the pole deviates more than 15 degrees from vertical, the episode is over.



```
# Demonstration
env = gym.envs.make("CartPole-v1")


def get_screen():
    ''' Extract one step of the simulation.'''
    screen = env.render(mode='rgb_array').transpose((2, 0, 1))
    screen = np.ascontiguousarray(screen, dtype=np.float32) / 255.
    return torch.from_numpy(screen)

# Specify the number of simulation steps
num_steps = 2

# Show several steps
for i in range(num_steps):
    clear_output(wait=True)
    env.reset()
    plt.figure()
    plt.imshow(get_screen().cpu().permute(1, 2, 0).numpy(),
               interpolation='none')
    plt.title('CartPole-v0 Environment')
    plt.xticks([])
    plt.yticks([])
    plt.show()
```

Figure 2.1 Code snippet of model

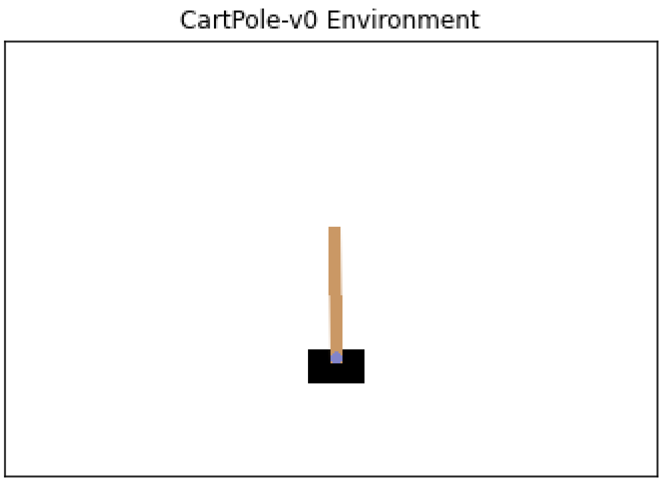
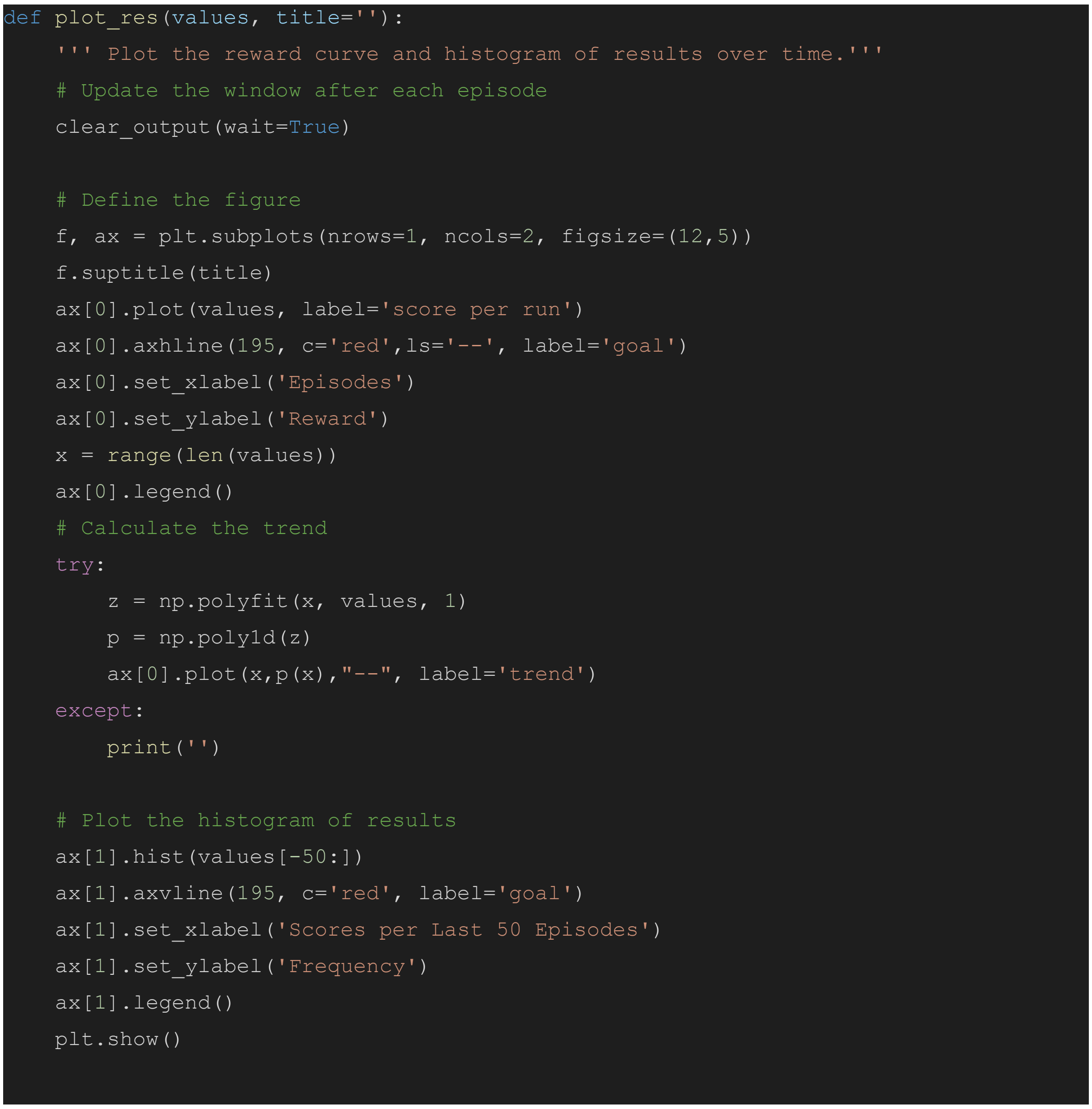


Figure 2.2 Image of Cartpole demonstration

3. Implementation

A. Capture of Data:

It will be possible to examine the agent's ongoing learning processes using this function. Two subplots make up the plot that results. The first one displays the agent's cumulative total reward while the second displays a histogram of the agent's cumulative total reward over the last 50 episodes.



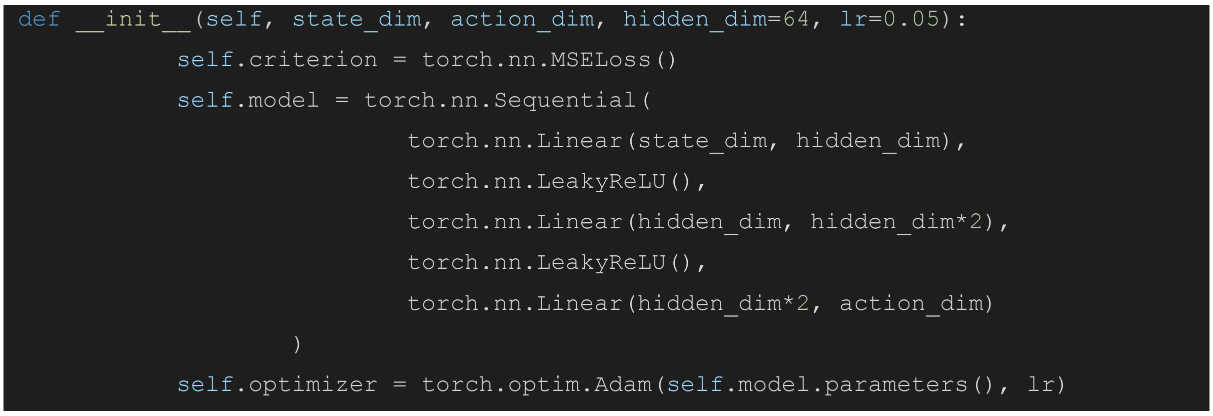
```python
def plot_res(values, title=''):
    ''' Plot the reward curve and histogram of results over time.'''
    # Update the window after each episode
    clear_output(wait=True)

    # Define the figure
    f, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,5))
    f.suptitle(title)
    ax[0].plot(values, label='score per run')
    ax[0].axhline(195, c='red',ls='--', label='goal')
    ax[0].set_xlabel('Episodes')
    ax[0].set_ylabel('Reward')
    x = range(len(values))
    ax[0].legend()
    # Calculate the trend
    try:
        z = np.polyfit(x, values, 1)
        p = np.poly1d(z)
        ax[0].plot(x,p(x),"--", label='trend')
    except:
        print('')

    # Plot the histogram of results
    ax[1].hist(values[-50:])
    ax[1].axvline(195, c='red', label='goal')
    ax[1].set_xlabel('Scores per Last 50 Episodes')
    ax[1].set_ylabel('Frequency')
    ax[1].legend()
    plt.show()
```

Figure 3.1 Code snippet of histogram plotting

B. The Network Structure:

First, we define a class named __init__ that we use to initialise an object of that class. This class takes four arguments: *state_dim, action_dim, hidden_dim* and *lr* (learning rate). We use the state_dim and action_dim arguments to declare the dimensions of the input and output tensors. The *hidden_dim* arguments defines the size of the hidden layers in the neural network, and lr is used by the Adam optimizer.

The __*init*__ function has three key parts. Firstly, a loss function is made using the mean squared error (*MSE*) loss. Next, a neural network is created with three linear layers followed by leaky ReLu nonlinearities. The sizes of the first two layers are determined by the *hidden_dim argument*, and the final layer's size is determined by the *action_dim argumen*t. Then, it creates an Adam optimizer using the model's parameters and the learning rate specified by the lr argument.



```
def __init__(self, state_dim, action_dim, hidden_dim=64, lr=0.05):
           self.criterion = torch.nn.MSELoss()
           self.model = torch.nn.Sequential(
                           torch.nn.Linear(state_dim, hidden_dim),
                           torch.nn.LeakyReLU(),
                           torch.nn.Linear(hidden_dim, hidden_dim*2),
                           torch.nn.LeakyReLU(),
                           torch.nn.Linear(hidden_dim*2, action_dim)
                   )
           self.optimizer = torch.optim.Adam(self.model.parameters(), lr)
```

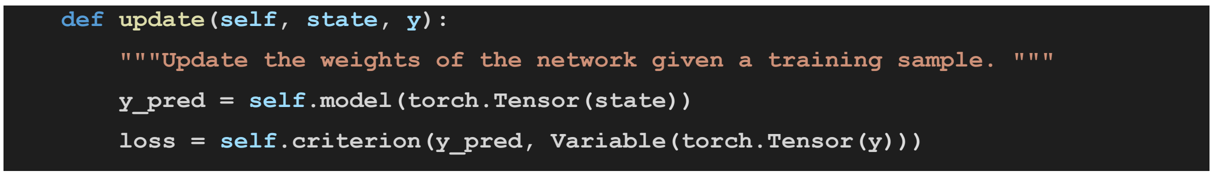
Figure 3.2 Code snippet showing network structure

C. Discussion Of Q-Learning Update

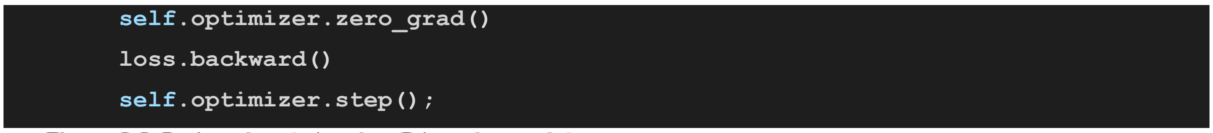
Q-Learning Update:

The update function is a method of the __init__ class that is used to update the weights of the neural network given a training sample. The function takes two arguments: *state* and *y*. state is a tensor that contains the input data, and y is a tensor that contains the expected output.

The function first applies the neural network to the input data in order to make a prediction. This prediction is stored in the variable y_pred. Then, the function calculates the loss between the predicted output and the expected output using the mean squared error loss function that was created in the __init__ function.



```
    def update(self, state, y):
        """Update the weights of the network given a training sample. """
        y_pred = self.model(torch.Tensor(state))
        loss = self.criterion(y_pred, Variable(torch.Tensor(y)))
```



```
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step();
```

Figure 3.3 Code snippet showing Q learning update

Next, the function uses the optimizer to perform backpropagation and update the weights of the neural network. First, it zeros out the gradients of the optimizer. Then, it uses the backward method to compute the gradients of the loss with respect to the model's parameters. Finally, it uses the step method to update the model's parameters based on the computed gradients and the learning rate. This process adjusts the weights of the model in a way that reduces the loss on the training sample.

Calculating Loss:

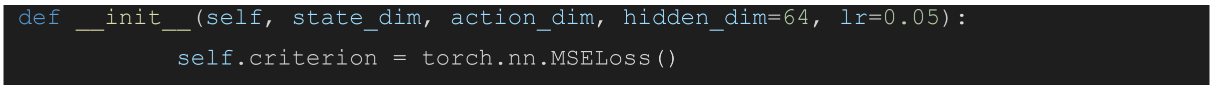
To calculate the error, we use *torch.nn.MSELoss().* Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input *x* and target *y*. The unreduced (i.e. with reduction set to ‘none’) loss can be described as:

$$\ell\ (x,y) = L = \{l1,\ldots,lN\}\ ^\top,\ ln = (xn - yn)\ 2$$

- Where N is the batch size. If reduction is not ‘none’ (default ‘mean’), then:

$$\ell\ (x,y) = \{\text{mean}\ (L),\ \text{sum}(L),\ \text{if reduction} = \text{‘mean’};\ \text{if reduction} = \text{‘sum’}$$

- x and y are tensors of arbitrary shapes with a total of *n* elements each. The mean operation still operates over all the elements, and divides by *n*.
- The division by *n* can be avoided if one sets reduction = ‘sum’. [1]



```
def __init__(self, state_dim, action_dim, hidden_dim=64, lr=0.05):
            self.criterion = torch.nn.MSELoss()
```

Figure 3.4 Code snippet showing calculations of loss

Q-Learning with Replay:

The DQN_replay class is a subclass of the DQN class that adds a replay function. This function is used to train the DQN model using a batch of experiences from the replay memory.

The replay function takes in the following parameters:

Memory: a list of experiences that will be used to train the model.

Size: the number of experiences to use in the batch.

Gamma: the discount factor used in the Q-learning algorithm. This determines how much the agent values future rewards over current rewards. [2]

- The function first checks if the memory list has enough experiences to sample a batch of the specified size. If so, it samples a batch of experiences and transposes the list to separate the state, action, next state, reward, and done values. It then converts the values to tensors and calculates the Q-values for the current and next states using the DQN model.

- Next, the function updates the Q-values for the actions taken in the experiences in the batch. It updates the Q-values for non-terminal states using the reward and the maximum Q-value for the next state. It updates the Q-values for terminal states using the reward only.

Finally, the function updates the model using the updated Q-values and the states from the batch. This allows the model to learn from the experiences in the replay memory.

```python
# Expand DQL class with a replay function.
class DQN_replay(DQN):


    #new replay function
    def replay(self, memory, size, gamma=0.9):
        """New replay function"""
        #Try to improve replay speed
        if len(memory)>=size:
            batch = random.sample(memory,size)
            batch_t = list(map(list, zip(*batch))) #Transpose batch list
            states = batch_t[0]
            actions = batch_t[1]
            next_states = batch_t[2]
            rewards = batch_t[3]
            is_dones = batch_t[4]

            states = torch.Tensor(states)
            actions_tensor = torch.Tensor(actions)
            next_states = torch.Tensor(next_states)
            rewards = torch.Tensor(rewards)
            is_dones_tensor = torch.Tensor(is_dones)

            is_dones_indices = torch.where(is_dones_tensor==True)[0]

            all_q_values = self.model(states) # predicted q_values of all states
            all_q_values_next = self.model(next_states)
            #Update q values

all_q_values[range(len(all_q_values)),actions]=rewards+gamma*torch.max(all_q_values
_next, axis=1).values
            all_q_values[is_dones_indices.tolist(),
actions_tensor[is_dones].tolist()]=rewards[is_dones_indices.tolist()]



            self.update(states.tolist(), all_q_values.tolist())
```
Figure 3.5 Code snippet of Q-Learning with replay

Double Q learning

Traditional Deep Q Learning has a propensity to overestimate the reward, which causes training to be unstable and policy to be of lower quality. Let's take a look at the Q value equation:

$$Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}[r + \gamma\ \max_{a'} Q^*(s',a')|\ s,a]$$

[3]

The greatest value's estimated value is used in the equation's last component. Through systematic overestimation caused by this process, a maximising bias is introduced. Such overestimation is particularly concerning because Q-learning requires learning estimates from estimates.

- We then set up a new target network to prevent this from happening. This new network, which is intended to represent the condition of the primary DQN, will be used to determine the Q values. Due to the fact that it is only updated after a particular amount of episodes, it does not have the same weights. The first publication of this concept was in Hasselt et al. (2015).

- Given that the target network is not constantly updated, the inclusion of the target network could cause the training to slow down. Over time, though, it ought to work with greater sturdiness.

- n_update parameter specifies the interval, after which the target network should be updated.

```
class DQN_double(DQN):

    def __init__(self, state_dim, action_dim, hidden_dim, lr):

        super().__init__(state_dim, action_dim, hidden_dim, lr)

        self.target = copy.deepcopy(self.model)


    def target_predict(self, s):

        ''' Use target network to make predictions.'''
        with torch.no_grad():
            return self.target(torch.Tensor(s))

    def target_update(self):
        ''' Update target network with the model weights.'''
        self.target.load_state_dict(self.model.state_dict())

    def replay(self, memory, size, gamma=1.0):
        ''' Add experience replay to the DQL network class.'''
        if len(memory) >= size:
            # Sample experiences from the agent's memory
            data = random.sample(memory, size)
            states = []
            targets = []
            # Extract datapoints from the data
            for state, action, next_state, reward, done in data:
                states.append(state)
                q_values = self.predict(state).tolist()
                if done:
                    q_values[action] = reward
                else:
                    # The only difference between the simple replay is in this line
                    # It ensures that next q values are predicted with the target
network.
                    q_values_next = self.target_predict(next_state)
                    q_values[action] = reward + gamma *
torch.max(q_values_next).item()

                targets.append(q_values)

            self.update(states, targets)
# Get replay results
dqn_double = DQN_double(n_state, n_action, n_hidden, lr)
double =  q_learning(env, dqn_double, episodes, gamma=.9,
                    epsilon=0.2, replay=True, double=True,
                    title='Double DQL with Replay', n_update=10)
```

Figure 3.6 Code snippet of Double Q-Learning

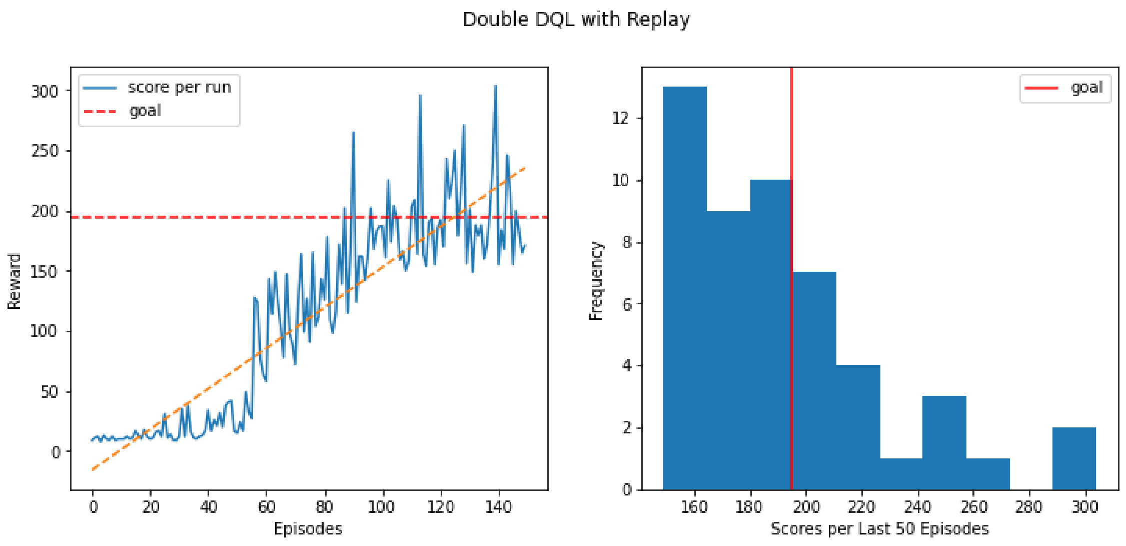


Figure 3.7 Graphs showing results of Double DQL with replay

The new version of Double DQL with playback outperformed the old one and regularly performed above 300 steps. The separation of action selection and evaluation has also led to a little increase in performance stability. Let's now examine the most recent change to the DQL agent.

Double Soft Target Update

The original DQN publication introduced the technique used to update the target network discussed above. We will examine yet another tried-and-true technique for modifying the target network weights in this section. We will incrementally update the target network after each run rather than updating weights after a predetermined number of runs, using the following formula:

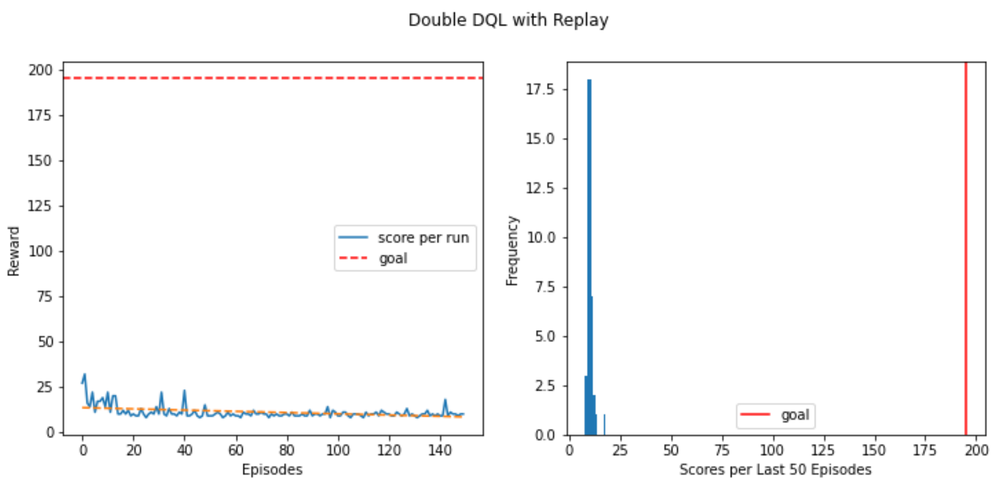
- ***target_weights = target_weights * (1-TAU) + model_weights * TAU where 0 < TAU < 1***

"Soft target network updates" are a technique for upgrading the target network that was first presented by Lillicrap et al. in 2016. The following illustrates how the method was used.

```python
class DQN_double_soft(DQN_double):

    def target_update(self, TAU=0.1):

        ''' Update the targer gradually. '''

        # Extract parameters

        model_params = self.model.named_parameters()

        target_params = self.target.named_parameters()


        updated_params = dict(target_params)


        for model_name, model_param in model_params:

            if model_name in target_params:

                # Update parameter
```

```
updated_params[model_name].data.copy_((TAU)*model_param.data +
(1-TAU)*target_params[model_param].data)

        self.target.load_state_dict(updated_params)

dqn_double_soft = DQN_double_soft(n_state, n_action, n_hidden, lr)
double =  q_learning(env, dqn_double_soft, episodes, gamma=.9,
                    epsilon=0.2, replay=True, double=True,
                    title='Double DQL with Replay', n_update=10,
soft=True)
```

Figure 3.8 Code snippet of Double Soft Target Update



The soft target updates network operated admirably. After a certain number of steps, it doesn't appear to be any better than hard weight updates.

Conclusion on Q Learning Update

The performance of a Deep Q Learning agent in the OpenAI CartPole environment has been greatly enhanced by the introduction of the experience replay and the target network. To enhance the performance of the agent, other modifications could be made, such as Dueling Network Architectures (Wang et al., 2015). The algorithm can also be applied in different settings. As a result, it is easy to test how well it works on various jobs.

Q-Learning Function:

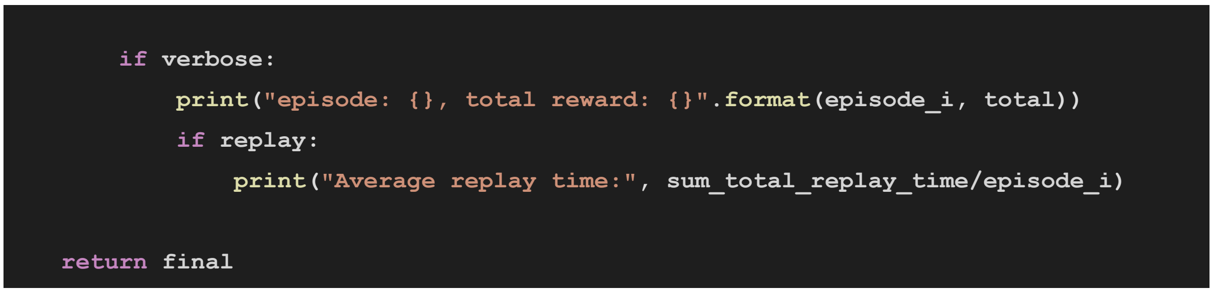
The function enters a loop that will run for the number of episodes specified in the input. For each episode, the function first checks if the double DQN or soft update is being used. If so, it updates the target network accordingly. Then it resets the environment and initialises the done and total variables.

- The function then enters a loop that will run until the episode is complete. In each step, the function first implements the epsilon-greedy policy to decide whether to take a random action or the action with the highest predicted value. Then it takes the action and updates the total reward and the memory list with the current experience.

If the episode is complete, the function checks whether the replay memory is being used. If not, it updates the model using the last experience. If the replay memory is being used, the function updates the model using the replay memory and calculates the time spent on the update.

If the episode is not complete, the function updates the model using the last experience or the replay memory, depending on the value of the `

```
def q_learning(env, model, episodes, gamma=0.9,
               epsilon=0.3, eps_decay=0.99,
               replay=False, replay_size=20,
               title = 'DQL', double=False,
               n_update=10, soft=False, verbose=True):
    """Deep Q Learning algorithm using the DQN. """
    final = []
    memory = []
    episode_i=0
    sum_total_replay_time=0
    for episode in range(episodes):
        episode_i+=1
        if double and not soft:
            # Update target network every n_update steps
            if episode % n_update == 0:
                model.target_update()
        if double and soft:
            model.target_update()
        # Reset state
        state = env.reset()
        done = False
        total = 0

        while not done:
            # Implement greedy search policy to explore the state space
            if random.random() < epsilon:
                action = env.action_space.sample()
            else:
                q_values = model.predict(state)
                action = torch.argmax(q_values).item()

            # Take action and add reward to total
            next_state, reward, done, _ = env.step(action)

            # Update total and memory
            total += reward
            memory.append((state, action, next_state, reward, done))
            q_values = model.predict(state).tolist()

            if done:
                if not replay:
                    q_values[action] = reward
                    # Update network weights
                    model.update(state, q_values)
                break

            if replay:
                t0=time.time()
                # Update network weights using replay memory
                model.replay(memory, replay_size, gamma)
                t1=time.time()
                sum_total_replay_time+=(t1-t0)
            else:
                # Update network weights using the last step only
                q_values_next = model.predict(next_state)
                q_values[action] = reward + gamma * torch.max(q_values_next).item()
                model.update(state, q_values)

            state = next_state

        # Update epsilon
        epsilon = max(epsilon * eps_decay, 0.01)
        final.append(total)
        plot_res(final, title)
```



```
        if verbose:
            print("episode: {}, total reward: {}".format(episode_i, total))
            if replay:
                print("Average replay time:", sum_total_replay_time/episode_i)

    return final
```

Figure 3.9 Code snippet of Q-Learning Function

D. Other Concepts

Random Seed Initialization:

Random seed initialization is the process of setting a specific value for the random number generator, which is used to initialise the weights of the neural network in a Deep Q-Network and generate random actions during the exploration phase of training. This ensures that the results of the reinforcement learning algorithm are reproducible, as using the same random seed consistently will yield the same results each time the algorithm is run.

Impact on Regularisers on the Score:

Regularisation can be used to improve the generalisation ability of the model and prevent it from overfitting to the training data. This can be achieved by adding a regularisation term to the loss function, which penalises the model for having large weights.

The impact of regularisation on scores in a DQN for the Cartpole game can be significant. In general, the use of regularisation can improve the performance of the model by reducing overfitting and improving the generalisation ability of the model. This can result in better scores, as the model is able to generalise to unseen states in the game and make more accurate predictions. However, the exact impact of regularisation on scores will depend on the specific regularisation techniques used and the hyperparameters chosen. It's important to carefully tune these hyperparameters to find the best balance between regularisation and performance.

Some common regularisation techniques include:

L1 regularisation, which adds a penalty term to the loss function based on the sum of the absolute weights of the model.

L2 regularisation, which adds a penalty term to the loss function based on the sum of the squared weights of the model.

Dropout, which randomly drops out units (neurons) from the network during training, effectively reducing the complexity of the model and preventing overfitting.

Early stopping, which stops the training process early if the performance of the model on the validation set starts to deteriorate, preventing the model from overfitting to the training data.

Duelling Network Architectures:

Duelling network architectures are a type of architecture used in Deep Q-Networks (DQN) to improve the performance of the model. In a standard DQN, the output of the network is a single scalar value, representing the predicted value of the current state. In a duelling network architecture, the output of the network is divided into two separate streams: one stream estimates the value of the current state, while the other stream estimates the advantage of each action in the current state. These two streams are then combined to produce the final Q-values for each action, allowing the model to better estimate the relative value of each action in a given state.

In our implementation we have a `DQN_double(DQN)` class which uses duelled network architecture.

Techniques to Counter Maximization Bias:

Maximisation bias is a common problem in reinforcement learning, where an agent overestimates the rewards of certain actions.

Duelling Network Architectures:

One approach to counter this bias is to use a Double DQN, which uses two separate networks to decouple the action selection from the value estimation. This can help prevent the network from overestimating the values of certain actions.

Boltzmann Exploration:

Another approach is to use a technique called the "boltzmann exploration", which adds a level of randomness to the action selection process by sampling actions from a boltzmann distribution. This can help prevent the agent from getting stuck in a suboptimal policy by encouraging it to explore other actions. [4]

In the context of reinforcement learning, the boltzmann exploration works by sampling actions from a boltzmann distribution, which is defined as follows:

P(a) = exp(Q(a)/T) / sum(exp(Q(a)/T))

where P(a) is the probability of selecting action a, Q(a) is the estimated value of action a, and T is a temperature parameter that controls the level of randomness. The higher the value of T, the more random the action selection will be.

Smaller Learning Rate:

Additionally, using a smaller learning rate for the Q-values can also help reduce the effects of maximisation bias.

4. Results

In our implementation we had 1 Random search which was our baseline for all other strategies. Furthermore we had 4 variations of different Q-Learning algorithms. These include:

1. Deep Q-Learning
2. Deep Q-Learning with Replay
3. Double Deep Q-Learning
4. Double Deep Q-Learning with Soft Target Update.

Random search

For Random search we ran it for 150 episodes. On the left hand side you can see a graph where the y-axis represents the reward achieved for each episode and the x-axis represents the episode. The graph on the right hand side shows a histogram of the agent's total rewards for the last 50 episodes.

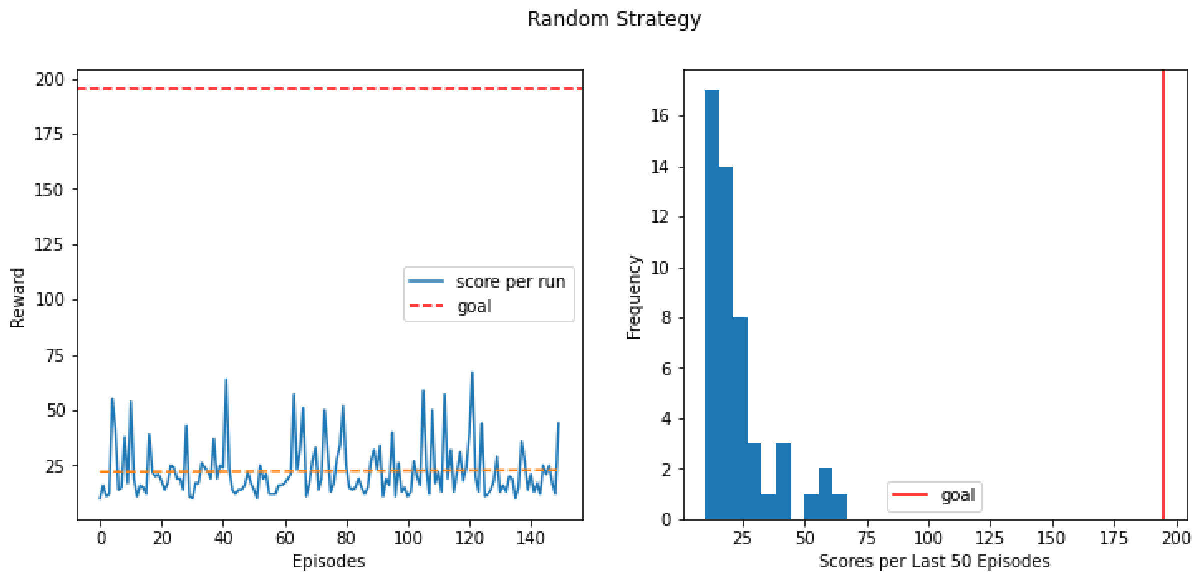


Figure 4.1 Graphs showing results of Random Search

Deep Q-Learning

For Deep Q-Learning we ran it for 150 episodes. On the left hand side you can see a graph where the y-axis represents the reward achieved for each episode and the x-axis represents the episode. side shows a histogram of the agent's total rewards for the last 50 episodes.

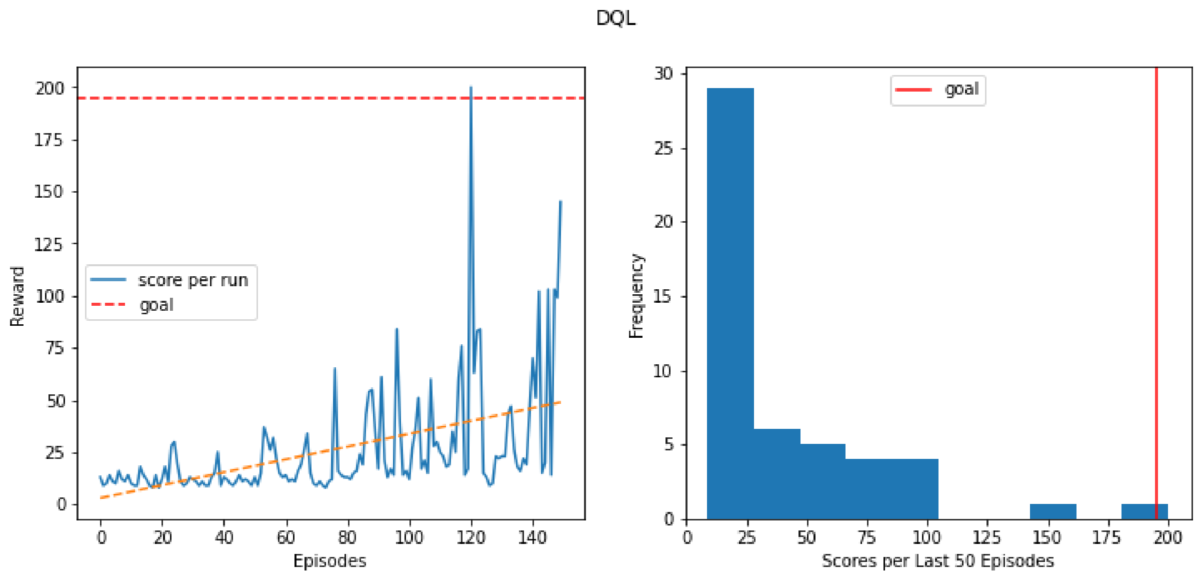


Figure 4.2 Graphs showing results of Deep Q-Learning

Deep Q-Learning with Replay

For Deep Q-Learning we ran it for 150 episodes. On the left hand side you can see a graph where the y-axis represents the reward achieved for each episode and the x-axis represents the episode. side shows a histogram of the agent's total rewards for the last 50 episodes.

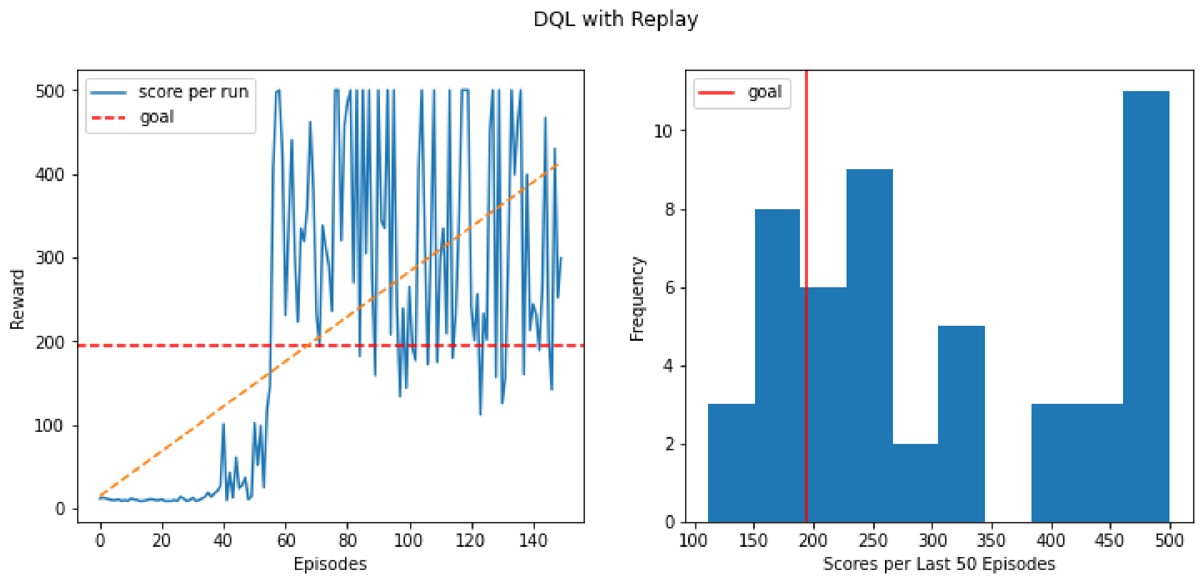


Figure 4.3 Graphs showing results of Deep Q-Learning with Replay

Double Deep Q-Learning

For Double Deep Q-Learning with Replay we ran it for 150 episodes. On the left hand side you can see a graph where the y-axis represents the reward achieved for each episode and the x-axis represents the episode. side shows a histogram of the agent's total rewards for the last 50 episodes.

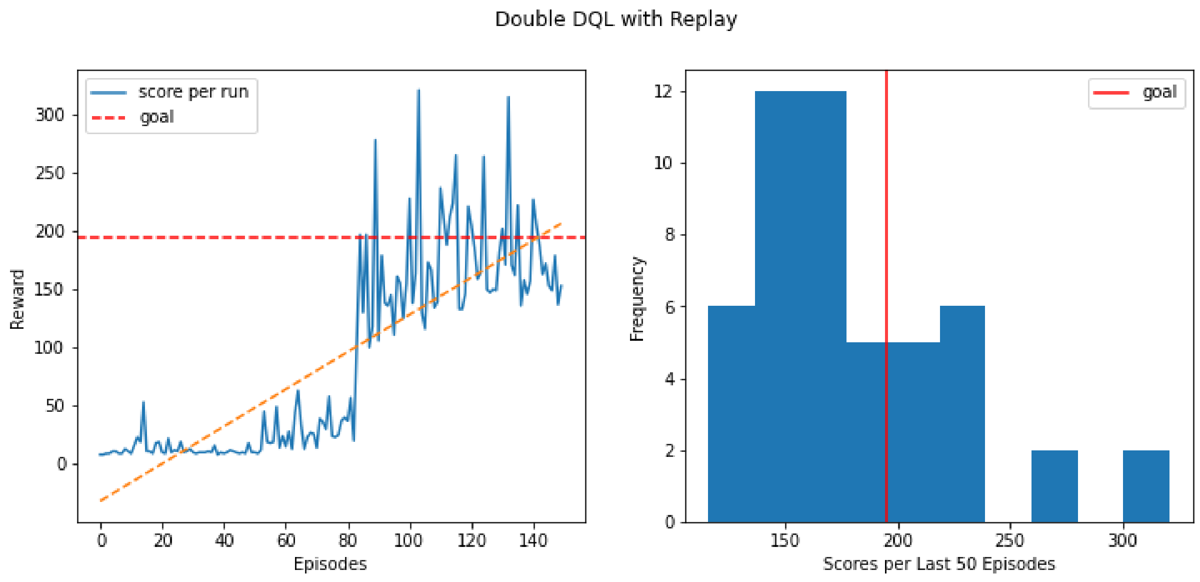


Figure 4.4 Graphs showing results of Double Deep Q-Learning

Double Deep Q-Learning with Soft Update

For Double Deep Q-Learning with Soft Update we ran it for 150 episodes. On the left hand side you can see a graph where the y-axis represents the reward achieved for each episode and the x-axis represents the episode. side shows a histogram of the agent's total rewards for the last 50 episodes.

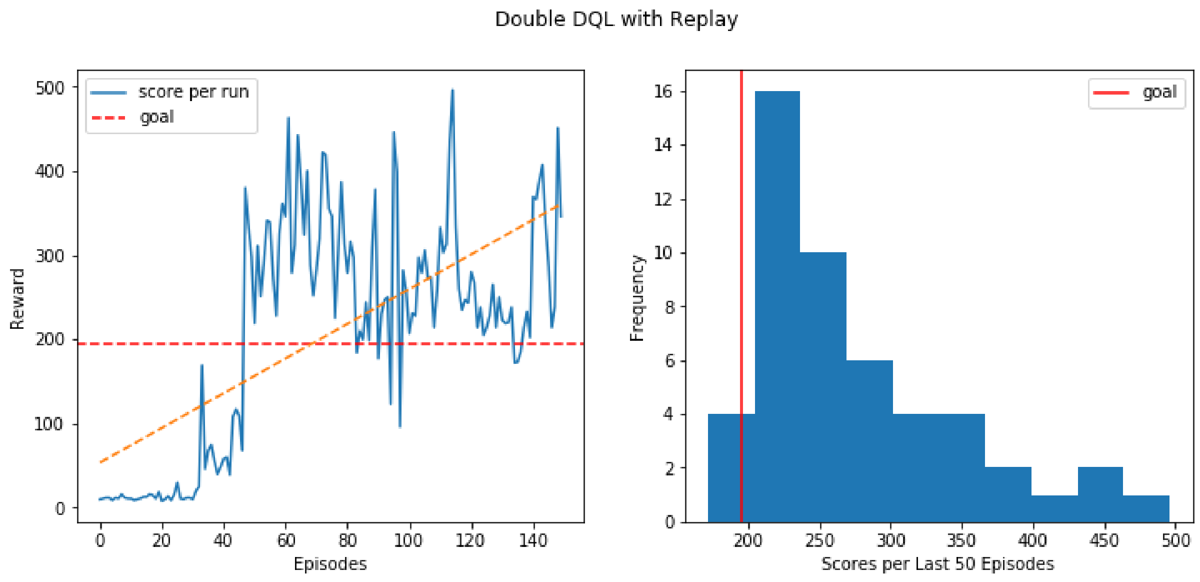


Figure 4.5 Graphs showing results of Double Deep Q-Learning with Soft Update

5. Evaluation of Results

After acquiring the results above, we were satisfied with how well our model performed during each variation of training.

Random Search

We ran Random Search for 150 episodes, and found the average performance to be around 10 steps. There were some outliers, where this search got lucky, achieving a reward of over 70. As expected, it's impossible to solve the environment using this approach. The agent is not learning from their experience. This search was mainly used as a baseline for the other strategies.

Deep Q-Learning

Our DQN improved immensely from the random search technique. We finished episode 150 with a total reward of 145. It's also noteworthy that the model improved in a positive direction during training as shown by the yellow trend line.

```
episode: 150, total reward: 145.0
```

Deep Q-Learning with Replay

To increase network stability and make sure that prior experiences are not lost but rather employed in training, we introduced experience replay. With a reward of 299 and an average replay time of 0.294329(..) we found that this method was much more robust than its counterpart. Its total rewards improved by over 150.

```
episode: 150, total reward: 299.0
Average replay time: 0.2943291505177816
```

Double Deep Q-Learning

We then attempted the cartpole problem by using Double Deep Q-Learning. We found that this outperformed Deep Q-Learning, and the separation of action selection and evaluation also made the performance appear to be a little bit more stable. Its total rewards were higher than Deep Q-learning by 13.

```
episode: 150, total reward: 153.0
Average replay time: 0.32362342993418375
```

Double Deep Q-Learning with Soft Target Update.

After evaluating our results for soft target update we concluded that we get no benefits over hard target updates such that were used with previous methods.

6. References

[1]: https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html

[2]:
https://medium.com/@leosimmons/double-dqn-implementation-to-solve-openai-gyms-cartpole-v-0-df554cd0614d

[3]: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[4]: https://lilianweng.github.io/posts/2020-06-07-exploration-drl/