Assignment - 1

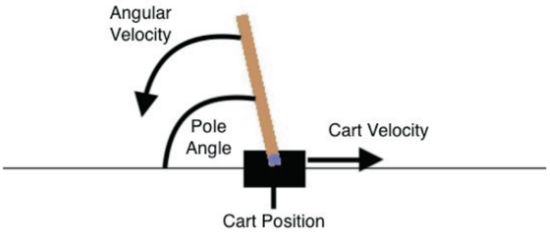
Mayank Chadha — IMT2020045

Instructor: Prof. Viswanath G *Date: March 17, 2024*

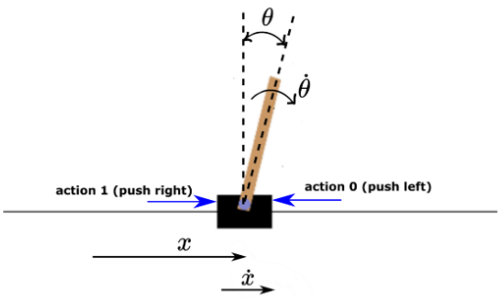
1 DQN for the Cartpole problem

1.1 Task

The agent has to decide between two actions - moving the cart left or right - so that the pole attached to it stays upright.



Cartpole- States



Cartpole- Actions

As the agent observes the current state of the environment and chooses an action, the environment transitions to a new state and also returns a reward that indicates the consequences of the action. In this task, rewards are +1 for every incremental timestep and the environment terminates if the pole falls over too far or the cart moves more than 2.4 units away from center. This means better-performing scenarios will run for a longer duration, accumulating larger returns.

The CartPole task is designed so that the inputs to the agent are 4 real values representing the environment state (position, velocity, etc.). We take these 4 inputs without any scaling and pass them through a small fully connected network with 2 outputs, one for each action. The network is trained to predict the expected value for each action, given the input state. The action with the highest expected value is then chosen.

1.1.1 Action Space

The action is a `ndarray` with shape `(1,)` which can take values {`0, 1`} indicating the direction of the fixed force the cart is pushed with.

- 0: Push cart to the left
- 1: Push cart to the right

Note: The velocity that is reduced or increased by the applied force is not fixed and it depends on the angle the pole is pointing. The center of gravity of the pole varies the amount of energy needed to move the cart underneath it

1.1.2 Observation Space

The observation is a `ndarray` with shape `(4,)` with the values corresponding to the following positions and velocities:

Note: While the ranges above denote the possible values for the observation space of each element, it is not reflective of the allowed values of the state space in an unterminated episode. Particularly:

- The cart x-position (index 0) can take values between `(-4.8, 4.8)`, but the episode terminates if the cart leaves the `(-2.4, 2.4)` range.
- The pole angle can be observed between `(-.418, .418)` radians (or $\pm 24^\circ$), but the episode terminates if the pole angle is not in the range `(-.2095, .2095)` (or $\pm 12^\circ$)

1.1.3 Rewards

Since the goal is to keep the pole upright for as long as possible, a reward of `+1` for every step taken, including the termination step, is allotted. The threshold for rewards is 500 for v1.

1.1.4 Episode End

The episode ends if any one of the following occurs:

- Termination: Pole Angle is greater than $\pm 12^\circ$
- Termination: Cart Position is greater than ± 2.4 (center of the cart reaches the edge of the display)
- Truncation: Episode length is greater than 500.

1.2 Components of DQN

1.2.1 Replay Memory

We'll be using experience replay memory for training our DQN. It stores the transitions that the agent observes, allowing us to reuse this data later. By sampling from it randomly, the transitions that build up a batch are decorrelated. It has been shown that this greatly stabilizes and improves the DQN training procedure.

For this, we're going to need two classes:

1. `Transition` - a named tuple representing a single transition in our environment. It essentially maps (state, action) pairs to their (next_state, reward) result, with the state being the screen difference image as described later on.
2. `ReplayMemory` - a cyclic buffer of bounded size that holds the transitions observed recently. It also implements a `.sample()` method for selecting a random batch of transitions for training.

1.2.2 Deep Q-network (`Class DQN`)

Our model will be a feed-forward neural network that takes in the difference between the current and previous screen patches. It has two outputs, representing `Q(s, left)` and `Q(s, right)` (where `s` is the input to the network). In effect, the network is trying to predict the expected return of taking each action given the current input.

1.3 Training

This cell instantiates our model and its optimizer, and defines some utilities:

- `select_action` - will select an action according to an epsilon greedy policy. Simply put, we'll sometimes use our model for choosing the action, and sometimes we'll just sample one uniformly. The probability of choosing a random action will start at `EPS_START` and will decay exponentially towards `EPS_END`. `EPS_DECAY` controls the rate of the decay.
- `plot_durations` - a helper for plotting the duration of episodes, along with an average over the last 100 episodes (the measure used in the official evaluations). The plot will be underneath the cell containing the main training loop, and will update after every episode.

At the main training loop. We reset the environment and obtain the initial `state` Tensor. Then, we sample an action, execute it, observe the next state and the reward (always 1), and optimize our model once. When the episode ends (our model fails), we restart the loop.

The `num_episodes` is set to 600 if a GPU is available, otherwise, 50 episodes are scheduled so training does not take too long. However, 50 episodes are insufficient to observe good performance on CartPole. You should see the model constantly achieve 500 steps within 600 training episodes. Training RL agents can be a noisy process, so restarting training can produce better results if convergence is not observed.

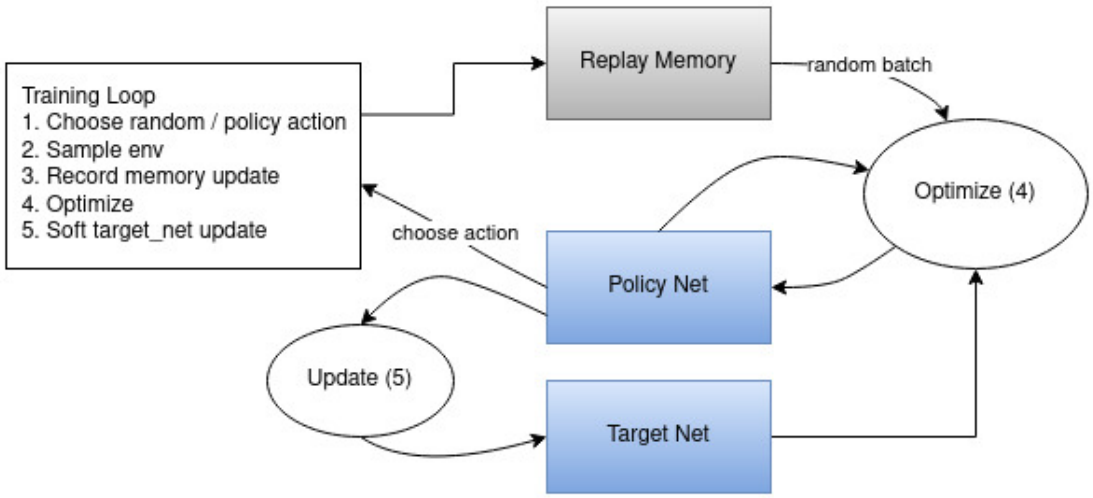


Figure 1: The overall resulting data flow

Actions are chosen either randomly or based on a policy, getting the next step sample from the gym environment. We record the results in the replay memory and also run the optimization step on every iteration. Optimization picks a random batch from the replay memory to do training on the new policy. The "older" `target_net` is also used in optimization to compute the expected Q values. A soft update of its weights is performed at every step.

1.4 Results

There are two lines plotted on the same graph:

- **Blue Line (Episode Duration):** This line represents the duration of each episode during training. It shows how many steps the agent was able to keep the pole balanced before the episode ended (either due to the pole falling or reaching a maximum episode length). Fluctuations or trends in this line indicate changes in the agent's behaviour or learning progress over episodes. **This can be inferred as the reward in each episode.**
- **Orange Line (Moving Average):** This line represents the moving average of episode durations. It calculates the average duration of the last 100 episodes at each point in training. This moving average is useful for smoothing out fluctuations and providing a clearer trend of the agent's learning progress over time. The moving average helps in understanding the overall learning progress of the agent, filtering out short-term fluctuations. **This can be inferred as the moving average reward in 100 episodes.**

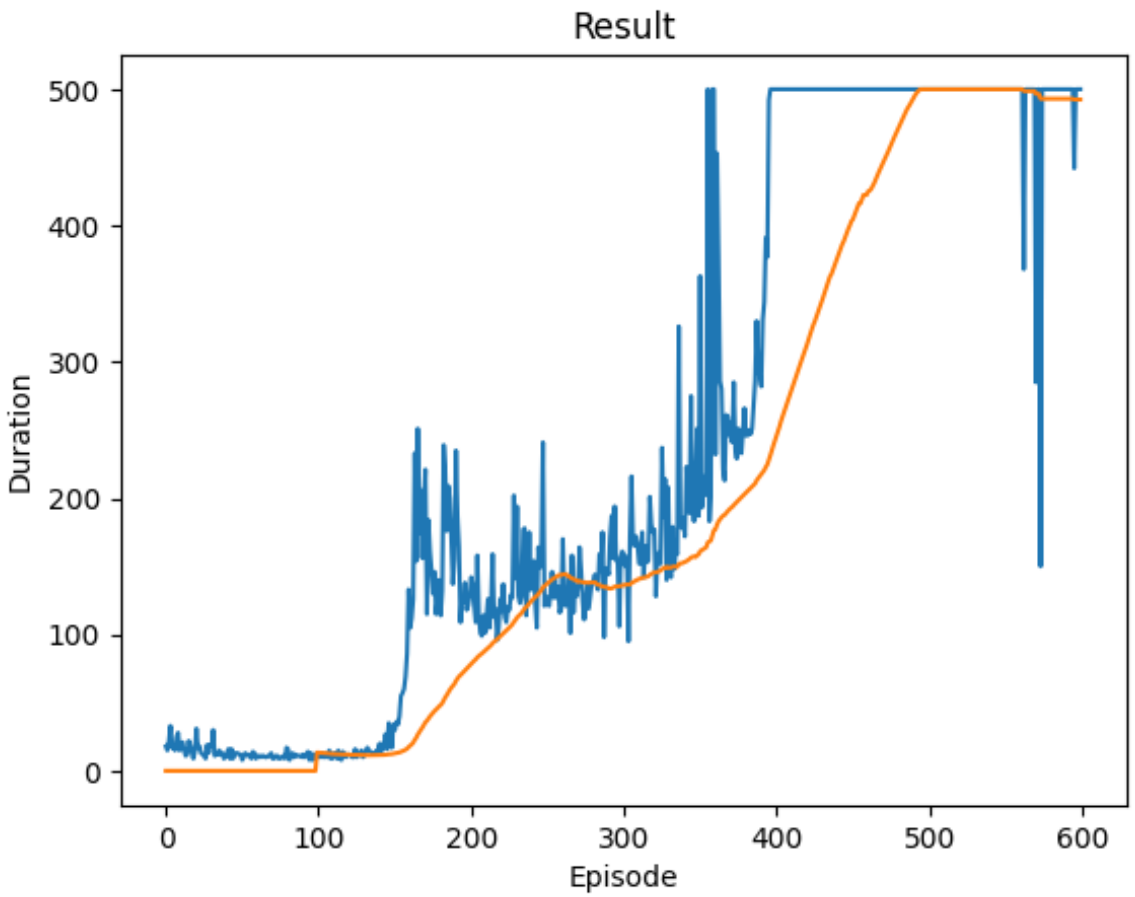


Figure 2: Example plot

2 Hyper-parameter Tuning on the DQN

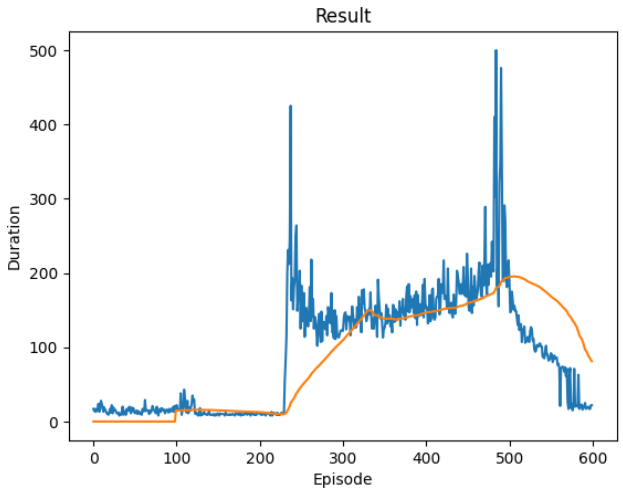
2.1 Hyper-parameters

Now, we will tune all the hyper-parameters and look for adjustments to gain better rewards on the training level. We will use Figure 2 as our base for comparison. The hyper-parameters taken for Figure 2 were:

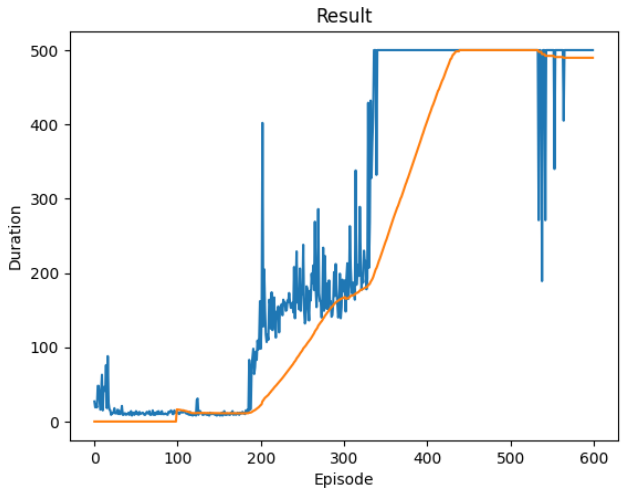
```
BATCH_SIZE, GAMMA, EPS_START, EPS_END, EPS_DECAY, TAU, LR, Network, replay_memory =
        initializer(128, 0.99, 0.9, 0.05, 1000, 0.005, 1e-4, DQN, 10000)
```

2.1.1 Tweaking Batch Size (BATCH_SIZE)

Experimenting with various batch sizes, such as 32, 64, 128, 512, reveals differing effects on the learning process. Smaller batch sizes may introduce more noise but can aid convergence in certain scenarios. However, in our experimentation with a batch size of 32, we observed suboptimal rewards at the conclusion of 600 episodes, failing to reach the performance peak achieved with a batch size of 128. Conversely, larger batch sizes might yield more stable updates but could potentially decelerate the learning process. In our investigation, we found that a batch size larger than 128 facilitated quicker attainment of the performance peak and maintained it for a prolonged period.



Batch Size decreased to 32 from 128

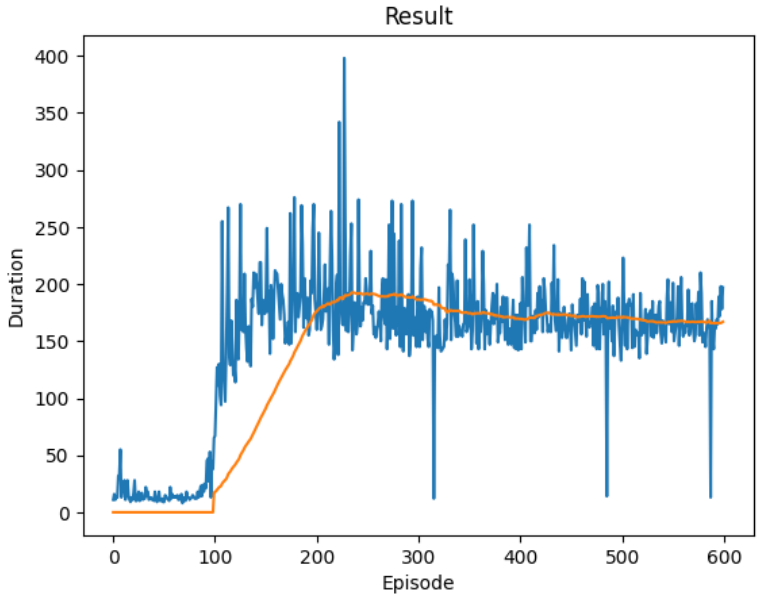


Batch Size increased to 512 from 128

Therefore, a higher batch size is good in the case of Cartpole_v1.

2.1.2 Tweaking Discount Factor (GAMMA)

This parameter, known as gamma, influences the emphasis placed on future rewards within the learning process. Values closer to 1 prioritize longer-term rewards, while values closer to 0 favour shorter-term rewards. However, setting gamma too high or too low can result in unstable learning or sluggish convergence. In our experimentation, reducing the gamma value from 0.99 to 0.75 revealed a notable impact. We observed that with the decreased gamma value, the model failed to achieve peak performance, and the overall performance stabilized at a lower reward level.

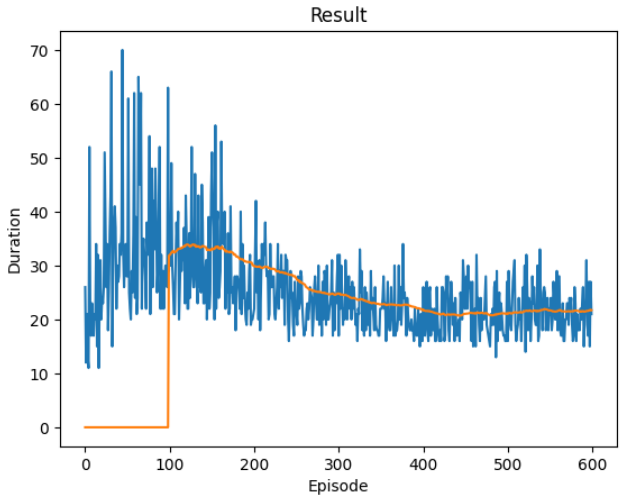


Gamma decreased to 0.99 from 0.75

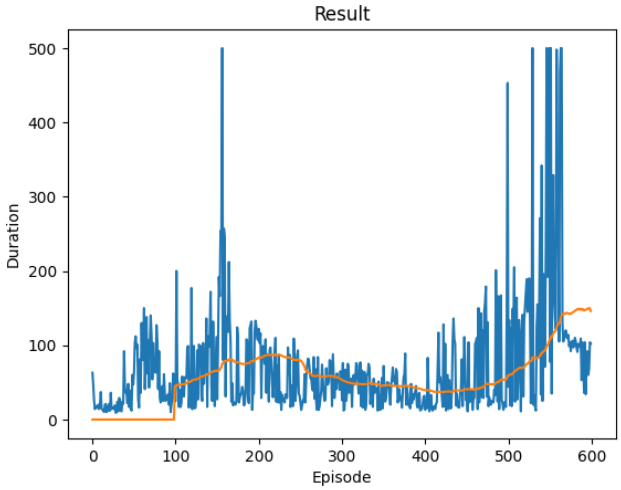
Therefore, higher gamma is good in the case of Cartpole_v1.

2.1.3 Tweaking Learning Rate (LR)

The learning rate plays a crucial role in determining the magnitude of steps taken during gradient descent. It's essential to experiment with various learning rates to strike a balance between convergence speed and stability. Setting the learning rate too high may induce oscillations or divergence, whereas a too-low learning rate could result in sluggish convergence. In our investigation, we observed significant effects when adjusting the learning rate. Decreasing the learning rate led to slower convergence, as evidenced by the failure to attain better rewards. Conversely, increasing the learning rate resulted in substantial divergence and erratic patterns, aligning with our expectations.



Learning Rate decreased to 1e-6 from 1e-4

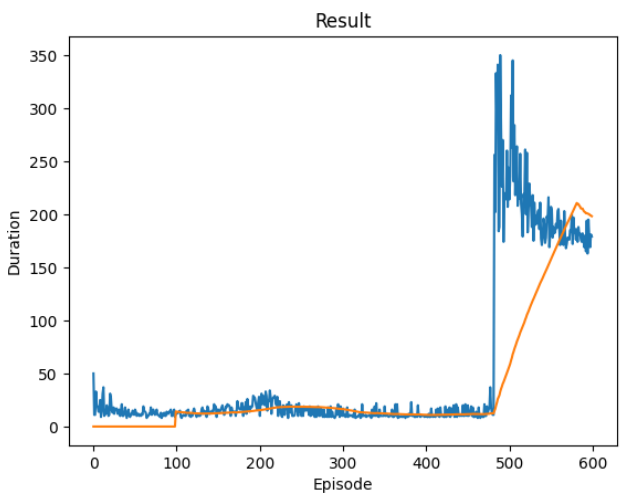


Learning Rate decreased to 1e-2 from 1e-4

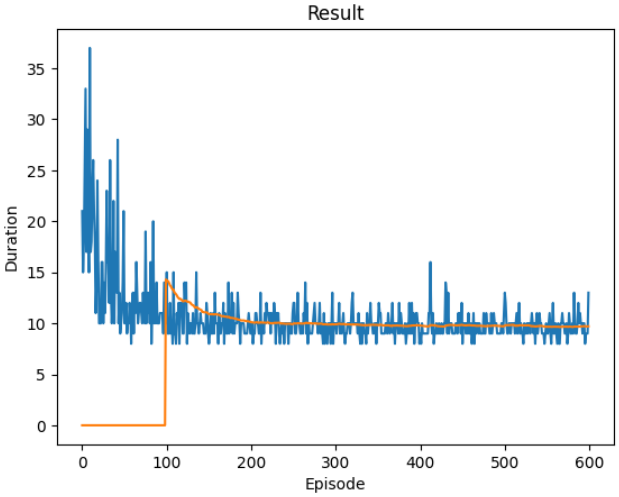
Therefore, the learning rate should not be very high and should not be very low it should be somewhere in between in the case of Cartpole_v1. Usually, a learning rate somewhere between 1e-3 and 1e-4 tends to work well. This is very important hyper-parameter.

2.1.4 Tweaking Target Network Update Rate (TAU)

This parameter governs the rate at which the target network adjusts towards the policy network. Higher values result in faster updates but may introduce instability. Conversely, lower values ensure a more stable target network but could potentially slow down the learning process. In our experiments, altering the Target Network Update Rate revealed significant impacts. Decreasing the rate caused delays in reaching peak rewards, slowing down the learning process. However, stability improved, with rewards consistently increasing. On the other hand, increasing the tau rate prevented the model from reaching the peak reward, stabilizing instead at a lower value. This adjustment also resulted in instability, as the network departed from its reward peak.



Target Network Update decreased to 0.001 from 0.005



Target Network Update decreased to 0.05 from 0.005

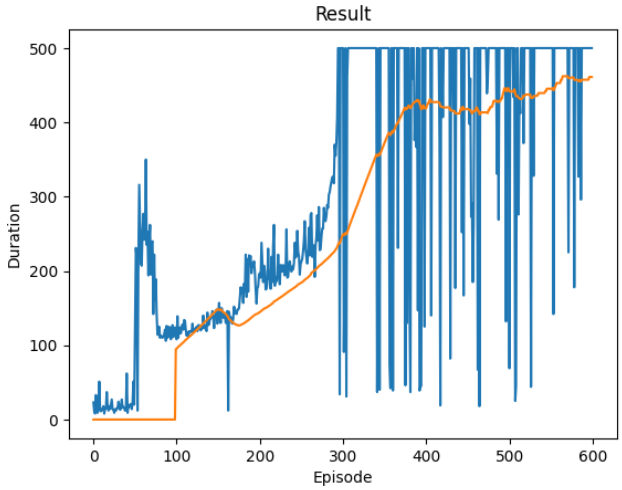
Therefore, the Target Network Update Rate (TAU) should not be very high and should not be very low it should be somewhere in between in the case of Cartpole_v1.

2.1.5 Tweaking Neural Network Architecture

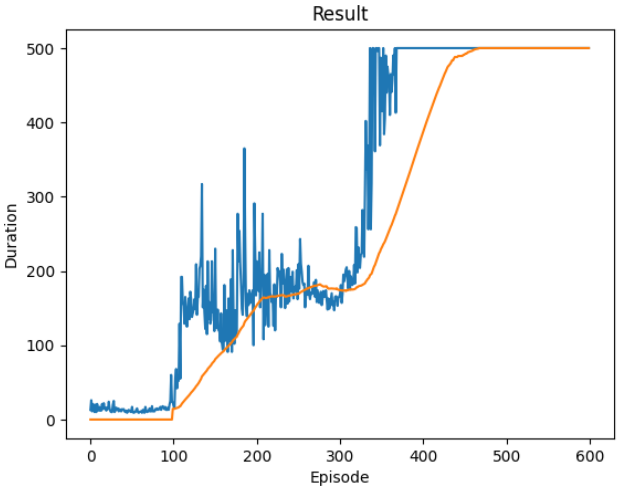
Experimentation with the neural network architecture, including variations in the number of layers, units per layer, and activation functions, is crucial. Deeper networks have the potential to capture more complex relationships but may be susceptible to overfitting. Conversely, simpler networks may generalize better but could underfit the data.

In our experiments, we observed that deep architectures reached peak rewards rapidly. However, they exhibited signs of overfitting, leading to oscillations in performance. Despite this, deep architectures consistently performed well, displaying higher rewards throughout the training process. The oscillations impacted the average reward, resulting in a slightly inferior performance compared to the non-deep architecture.

Regarding activation functions, we found that LeakyReLU performed exceptionally well. It facilitated faster attainment of peak rewards and maintained high rewards throughout training. In contrast, networks using ReLU exhibited slight oscillations in rewards, albeit to a lesser extent than those with deep architectures.



Neural Network Architecture is deep



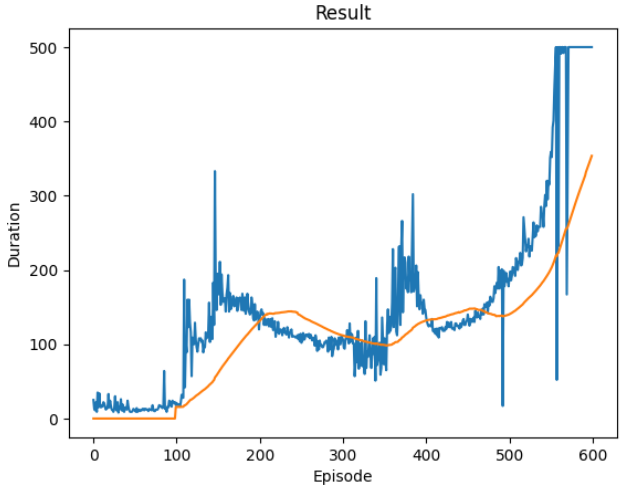
Neural Network has activation function as LeakyReLU

Hence, for tasks like Cartpole_v1, it's advisable to strike a balance in the Neural Network Architecture. An architecture that isn't overly deep can help prevent overfitting, while also ensuring it's not too shallow to adequately capture the underlying relationships within the data. Opting for activation functions like LeakyReLU, which outperforms ReLU, can also significantly impact performance. Therefore, careful consideration of both architecture depth and activation functions can greatly enhance the effectiveness of the neural network in such tasks.

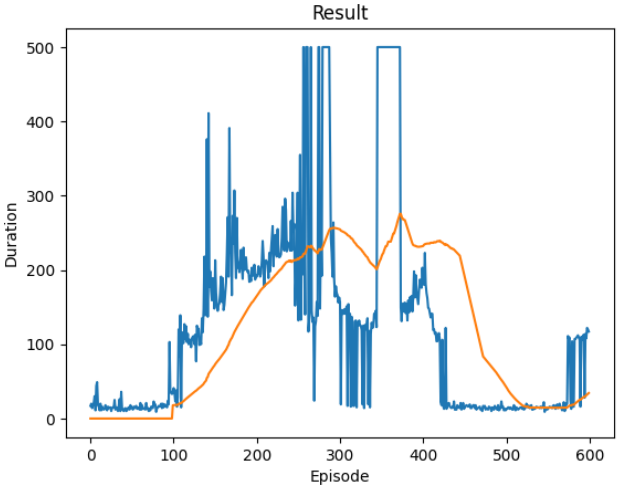
2.1.6 Tweaking Replay Memory Capacity

Fine-tuning the capacity of the replay memory, such as setting it to 10000 in your scenario, is crucial. Larger capacities offer the advantage of storing more experiences for learning, potentially enhancing the agent's performance. However, they may consume more memory resources. Conversely, smaller capacities could lead to rapid forgetting of useful experiences.

In our experiments, we observed distinct effects when adjusting the memory capacity. A larger memory capacity of 50000 resulted in a longer time to reach peak rewards. This delay can be attributed to the careful learning process necessitated by the abundance of stored experiences. Conversely, when the memory capacity was reduced to 4000, the agent swiftly reached peak performance compared to other parameters. However, it exhibited a tendency to miss the peak in consecutive episodes due to forgetting learned experiences caused by constrained memory capacity.



Replay Memory is decreased to 50000 from 10000



Replay Memory is decreased to 4000 from 10000

Therefore, the Replay Memory Capacity should not be very high and should not be very low it should be somewhere in between in the case of Cartpole_v1.

2.2 Other Factors

Other hyper-parameters that can be experimented are:

1. Exploration Rate (`EPS_START, EPS_END, EPS_DECAY`):
   These parameters are crucial for balancing exploration and exploitation in reinforcement learning. A higher exploration rate encourages the agent to explore the environment more extensively, aiding in discovering new strategies. However, setting the rate too high may hinder convergence. On the other hand, a lower exploration rate prioritizes exploitation of the current knowledge, potentially leading to faster convergence but risking the possibility of the agent getting stuck in suboptimal solutions. Careful tuning of these parameters is essential to strike the right balance and ensure effective learning. I set the epsilon decay factor so that it would hit min epsilon or 5% by about half a million steps. **EPS_DECAY is one of the most important factors in DQN.**

2. Optimizer and Loss Function:
   The choice of optimizer and loss function significantly impacts the training process. While `AdamW` optimizer with `Smooth L1 Loss (Huber Loss)` is commonly used and yields good results, it's beneficial to experiment with alternative optimizers such as `RMSprop` and different loss functions tailored to the specific characteristics of the problem at hand. Each optimizer and loss function has its strengths and weaknesses, and exploring different combinations can lead to improved performance and faster convergence.

3. Gradient Clipping:
   Gradient clipping is a technique used to mitigate the exploding gradient problem during training, where gradients become too large and cause instability. By capping the gradients to a predefined threshold, gradient clipping helps stabilize the training process and prevent divergence. Experimenting with different clipping thresholds allows finding the optimal balance between preventing gradient explosion and retaining learning dynamics. Adjusting the clipping threshold can have a significant impact on the stability and convergence speed of the training process.

2.3 Final Observation Table

The epsilon decay factor seems to be the most important parameter here. Increasing the learning rate also appears to negatively impact the test score. Min epsilon appears to be more forgiving, but it's important to note that this can be very environment-specific. In a game where a single wrong move can mean the end, a large minimum epsilon can effectively control the upper bound on performance.

Another important factor not added in the table is EPS_DECAY is one of the most important factors in DQN.

Everything that we discussed in the section above is summarized in the table below:

	Batch Size	Gamma	Learning Rate	TAU	Network Depth and Activation	Replay Memory
What to do?	Increase	Increase	Somewhere in between	Somewhere in between	Appropriate number of layers and LeakyReLU and other activation functions	Somewhere in between
Time affected?	Increase	Increase	Can't Say	Can't Say	Deep network takes more time, activation functions affect very little.	Can't Say

3 Modified DQN for input images

3.1 Major Modifications

The relevant design changes to your DQN system so that the system is trained using images only are listed below:

3.1.1 Update the Environment Setup

In addition to the standard libraries such as numpy and matplotlib, it's crucial to ensure your environment setup incorporates the necessary libraries for handling image inputs effectively. While working with image data in the CartPole problem, you'll find libraries like OpenCV invaluable for image processing tasks. These libraries provide comprehensive functionalities for image loading, manipulation, and preprocessing, which are essential for preparing image inputs before feeding them into the Deep Q-Network (DQN) model. Moreover, OpenCV offers a wide range of image processing techniques, including resizing, normalization, and color space conversions, which can be particularly useful for optimizing the input data for neural network training. Therefore, integrating these libraries into your environment setup ensures that you have the necessary tools to handle image inputs efficiently throughout your experimentation process.

```
import numpy as np
import gym

import collections
import random
import cv2

import tensorflow as tf
import keras
from keras.models import Model
from keras import layers
from keras import backend as K
from keras import optimizers

import pickle
```

Code change for Updating the Environment Setup

3.1.2 Modify the Input Preprocessing

Transitioning from the original DQN implementation, where inputs were represented as vectors of state values, to utilizing images requires a significant adjustment in the preprocessing step. Now, each image frame needs to be transformed into a format suitable for input to the neural network. This process typically involves several steps:

1. **Resizing**: Images captured from the CartPole environment may come in various sizes, but neural networks usually require inputs of a fixed size. Therefore, resizing the images to a consistent dimension is necessary. This ensures uniformity across the dataset and facilitates efficient processing by the neural network.
2. **Normalization**: Normalizing pixel values is a common practice in deep learning to ensure that the input data falls within a similar scale. This involves scaling pixel values to a range, often between 0 and 1 or -1

and 1. Normalization helps in stabilizing and accelerating the training process, making it less sensitive to the scale of input features.

3. **Stacking Frames**: CartPole is a dynamic environment where sequential frames contain valuable temporal information about the motion of the pole and cart. Therefore, stacking multiple frames together creates a temporal context, enabling the neural network to understand motion and make informed decisions. This stacking process creates a multi-channel input similar to how video frames are processed in convolutional neural networks (CNNs).

By performing these preprocessing steps, each image frame is transformed into a suitable format for input to the neural network. This modified input format effectively captures both spatial and temporal information from the environment, enabling the DQN model to learn and make decisions based on the visual observations of the CartPole system.

```python
def process_image(self, image):
    # Simple processing: RGB to GRAY and resizing keeping a fixed aspect ratio
    if len(image.shape) == 3:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    image = cv2.resize(image, (self.image_width, self.image_height))

    return image
```

Code change for input image Preprocessing

3.1.3 Update the Neural Network Architecture

When adapting your neural network architecture to accommodate image inputs, a pivotal adjustment lies in integrating convolutional layers. These layers play a crucial role in feature extraction from images, enabling the network to discern hierarchical representations of visual input. By applying filters to input images, convolutional layers excel at capturing features like edges and textures, essential for understanding the environment dynamics. Their ability to capture spatial hierarchies makes them particularly well-suited for tasks involving image processing, such as the CartPole problem.

Following convolutional layers, the architecture typically incorporates pooling layers to downsample feature maps while retaining pertinent information. Pooling techniques like max pooling help in reducing spatial dimensions, thus enhancing computational efficiency without sacrificing critical features. This downsampling process is vital for effective feature extraction, ensuring that the neural network focuses on the most salient aspects of the input images while discarding redundant information.

Finally, the adapted architecture includes fully connected layers, which are pivotal for decision-making based on the extracted features. Positioned after the convolutional and pooling layers, these fully connected layers map the high-level features to the desired outputs, such as Q-values for different actions in the CartPole environment. By integrating these layers, the network learns to associate visual patterns with optimal actions, enabling it to make informed decisions in dynamic environments.

```python
def create_CNN_model(self):

    input_shape = (self.stack_depth, self.image_height, self.image_width)
    actions_input = layers.Input((self.num_actions,), name = 'action_mask')

    frames_input = layers.Input(input_shape, name='input_layer')
    conv_1 = layers.Conv2D(32, (8,8), strides=4, padding ='same'\
    ,activation = 'relu', name='conv_1',kernel_initializer='glorot_uniform',bias_initializer='zeros')(frames_input)

    conv_2 = layers.Conv2D(64, (4,4), strides=2, padding='same', activation='relu',name='conv_2'\
       ,kernel_initializer='glorot_uniform',bias_initializer='zeros')(conv_1)

    conv_3 = layers.Conv2D(64, (3,3), strides=1, padding='same',name='conv_3', activation='relu'\
       ,kernel_initializer='glorot_uniform',bias_initializer='zeros')(conv_2)

    flatten_1 = layers.Flatten()(conv_3)

    dense_1 = layers.Dense(512, activation='relu', name='dense_1',
        kernel_initializer='glorot_uniform',bias_initializer='zeros')(flatten_1)
    output = layers.Dense(self.num_actions, activation='linear', name='output',
        kernel_initializer='glorot_uniform',bias_initializer='zeros')(dense_1)
    masked_output = layers.Multiply(name='masked_output')([output, actions_input])

    model = Model(inputs=[frames_input, actions_input], outputs=[masked_output])

    optimizer = optimizers.Adam(learning_rate=self.learning_rate)
    model.compile(optimizer, loss=huber_loss)

    return model
```

Code change for Updating the Neural Network Architecture

3.1.4 Adjust the Training Pipeline

Adapt your training pipeline to accommodate image inputs. This includes modifying the data pipeline to handle image frames and updating the training loop to process image inputs and calculate losses accordingly.

```python
step_rewards = []

#Loop for the number of episodes
for episode in range(1,episodes):

    #At the beginning of each new episode the first state is just formed by staking up
    #the same frame 4 times and saving it bufer
    seq_memory.clear()
    initial_state = env.reset()
    current_image = env.render()
    frame = agent.process_image(current_image)
    frame = frame.reshape(1, frame.shape[0], frame.shape[1])
    current_state = np.repeat(frame, stack_depth, axis=0)
    seq_memory.extend(current_state)

    #The episode reward is set to zero at the beginning of each new episode
    episode_reward = 0
    for time in range(time_steps):
```

Code change for Adjusting the Training Pipeline

3.2 Observations

Transitioning to image processing significantly impacts the computational demands of the training pipeline, resulting in prolonged training times. Due to these increased requirements, it becomes impractical to visualize the training progress for every iteration. Thus, in our approach, we opt to display the training progress for the initial 1000 iterations. However, it's essential to note that achieving optimal model efficiency with image inputs often necessitates training for a more extensive duration. In our case, the CNN model we employ is relatively deep, requiring approximately 9800 iterations before notable improvements are observed. This observation underscores the importance of prolonged training periods when working with image data, as deeper networks typically require more iterations to learn complex visual features effectively.

Furthermore, beyond the model architecture's depth, several additional factors contribute to the prolonged training times and the necessity for higher computational resources. For instance, image data tends to be more complex and high-dimensional compared to vector-based inputs, necessitating more extensive computations during both forward and backward passes. Additionally, the increased memory requirements for storing and processing image data further strain computational resources. Consequently, to achieve optimal model performance, it becomes imperative to have access to higher configuration GPUs capable of faster computations and more memory to handle the increased data load efficiently.

In practice, our observations highlight the trade-off between model complexity, training duration, and computational resources. While deeper CNN architectures offer enhanced capacity to learn intricate visual representations, they also require more significant computational resources and time for training. As such, researchers and practitioners must carefully consider these factors when designing and training image-based models, ensuring a balance between model performance and resource constraints.