1 Question 1: Implementing a functional DQN

1.1 Implementing DQN (See Appendix)

1.2 Implementing Atari game technical features

Firstly we implement a replay buffer to model experience replay. The replay buffer contains a collection of experience tuples (s, a, s‘, r). The tuples are gradually added to the buffer as we are interacting with the Environment. We implement a fixed size buffer, with new data added to the end of the buffer so that it pushes the oldest experience out of memory. A 'deque' data structure was used from Python's built-in collections library, this is a list that you can set a maximum size on. For training, we randomly sample the batch of transitions from the replay buffer, which allows us to break any correlation between subsequent steps in the environment.

Next I created a target network, by which we keep a copy of our neural network and use it for the Q(s', a') value in the Bellman equation in order to avoid instability from the neural network updating parameters at every step. This is done by first initialising the a target network along with the policy network (lines 229-238). The predicted Q values of this second Q-network are used to backpropagate through and train the main Q-network. The target network's parameters are not trained, but they are periodically synchronized with the parameters of the main Q-network, at a set amount of episodes which can be considered another hyperparameter.

The final technique used was 'stacking frames' as used in the Atari DQN implementation. This effectively takes advantages of pushing information from four previous states to the memory at every step. This means that the neural net takes in input dimension of k*4 features (i.e. a 4k x 1 vector), alternatively this could have been pushed as a 4 x k matrix, but linear neural network layers cannot take vector inputs meaning we have to flatten the four state values for each frame into one vector. This increased number of features should give the neural net more information to draw an action from, for example drawing conclusions about how the total reward is increased by adjacent frames and how the pole or cart accelerates. This could be done with the gym FrameStack wrapper but I chose to write it myself, with a k_states deque (of state tensors) with max length k. This is initialised with the environment (lines 261-263) as the first state for the first two elements. Then when an action is taken, the deque is flattened into a vector (line 266) and fed to the neural net, selecting an action from the maximum Q values. If the action selected is not terminal, the next state is appended to the end of the deque and the earliest state is shifted out of k_states (line 285). If the states are terminal then a tensor of zeros is added to k_states. k_states is again flattened and we push flattened_k_states, action, next_flattened_k_states and reward to memory). There is more manipulation in the optimiser to ensure that the states with a final terminal states are removed from the non_final_mask as per the original code.

1.3 Design Decisions

We have chosen to implement a feed-forward multi-layer neural network. This is composed of two hidden linear layers of 164 neurons with a ReLu activation function on each layer. The selection of this architecture was done by attempting a few different configurations within the range of 1-5 layers and 10-200 neurons. This experimentation demonstrated that, generally, wider networks gave a more stable learning curve alongside the other parameters. The choice of this number is dependent on the value chosen for k which changes the input dimension by factor k, and so this width and depth of network was as far as I could 'push' the architecture parameters before some over-fitting was seen with a dramatic drop off once the agent reached high rewards. In Figure 1 the basic structure of the network is shown, with a pair of 128 neuron wide layers. I

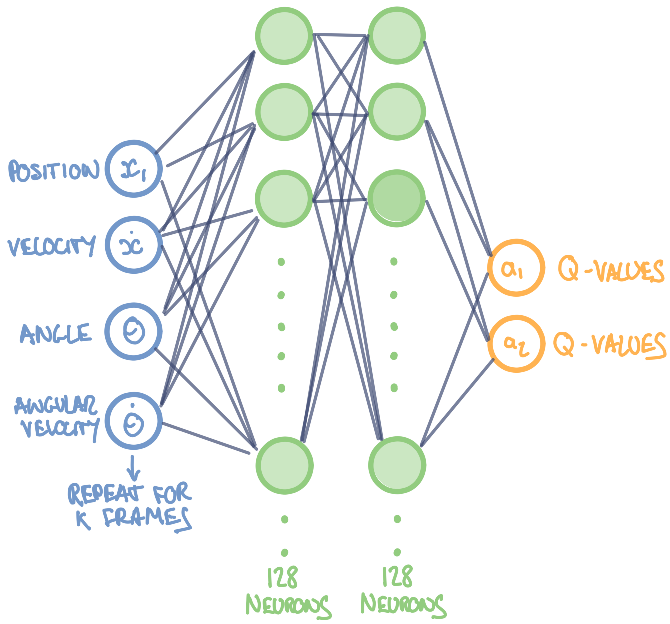


Figure 1: Neural Network Architecture

chose Adam as my optimizer. This is often the preferred optimizer used for neural nets in both classification and regression tasks using pytorch. The idea behind this method of gradient descent is to utilize the momentum concept from some Stochastic Gradient Descent implementations, and adaptive learning rate from another technique called "Ada delta". This also appeared to give the most stable results, and particularly results that converge. I used the pytorch 'Smooth L1 Loss' criterion. Also known as Huber loss, it uses a squared term if the absolute error falls below 1 and an absolute term otherwise. It has the advantage over the mean squared error loss that it is less sensitive to outliers and in some cases prevents exploding gradients.

1.4 Learning Curve

Figure 2 shows the mean reward values over 10 repetitions. The hyperparameters chosen gave a steady solution that converged as well as anything tried. Whilst some runs with different parameters were able to achieve scores of 500 the agent catastrophically worsened, this can be seen in figure 3, early stopping could be implemented, but this has the problem of stopping at different points and removing the ability to replicate for the same number of episodes. In the chosen solution an average rewards of 300 was reached, with the agent stabilising after this point, and taking 125 episodes to reach 90% of the final value.

2 Question 2: Hyperparameters of the DQN

2.1 Choice of ϵ

ϵ indicates the proportion of random actions relative to taken actions that are informed by existing agent "knowledge" accumulated during the episode. This strategy is called a "Greedy Search

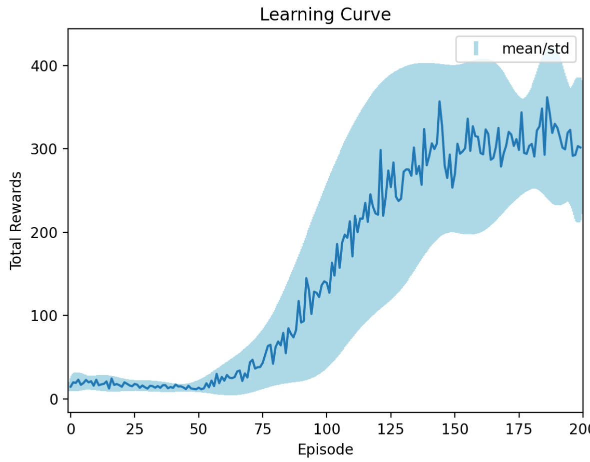


Figure 2: Learning Curve for chosen DQL parameters, showing mean and standard deviation over 10 replications for 200 episodes

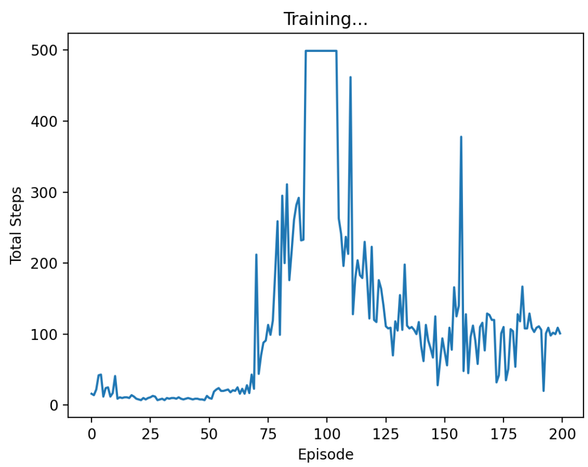


Figure 3: Learning Curve for alternative DQL parameters for one run of 200 episodes

Policy." Before playing the game, the cart agent doesn't have any experience, so it is common to set epsilon to higher values and then gradually decrease its value. This can decay in a number of ways, in literature usually by a constant value e.g. 0.99 every step, I have chosen a similar step but ensure that there is decay until a minimum value by creating a power law depreciation. This reduces to a value very close to zero (0.05) as more steps are taken in Figure 4 below with two other decay schedule options.

Immediately we see from the plot that for the '1/k' decay, i.e. ϵ is divided by the step number in the episode, the value for ϵ reduces drastically very quickly. This will have the effect of reducing the amount of exploration the agent undertakes. The learning curve for this decay can be seen in Figure 5c). The agent can be seen to start by chance with a higher reward but then does no exploration and therefore soon has no useful knowledge. Figure 5a) shows the effect of choosing a constant ϵ value of 0.6 which was seen to work fairly well. However as epsilon doesn't continue to decay there is still large degree of randomness after receiving a relatively high reward, the rewards fluctuate far more as it is still exploring instead of using its knowledge. The linear decay

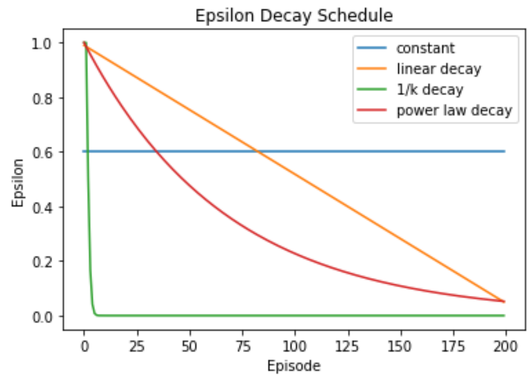
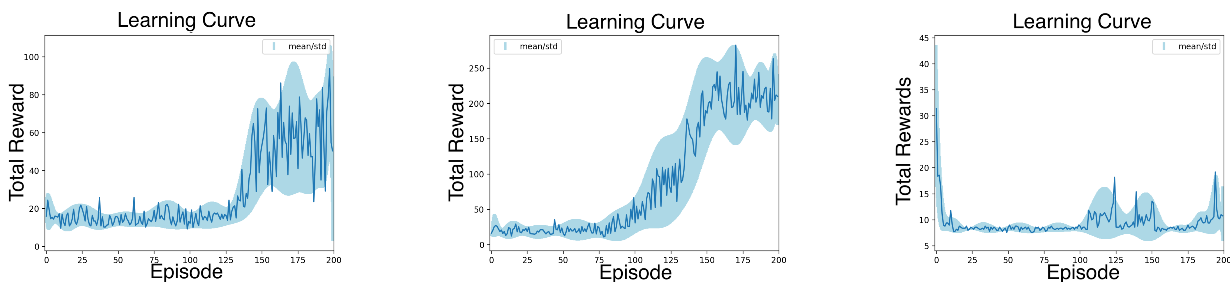


Figure 4: ϵ Decay Schedules, ϵ against N episodes

in Figure 5b) is improved but the mean fluctuates more then the power law decay, we need more exploitation sooner, and so we make a good choice for the epsilon decay schedule.



(a) ϵ Constant: 0.6 (b) ϵ Linearly decays (c) ϵ Decays with 1/k

Figure 5: Variation of ϵ decay schedule on learning curve for 5 replications

2.2 Choice of Replay Buffer Size

The replay buffer gives the agent the ability to store experiences in memory. Batches of experiences are randomly sampled from memory and subsequently used to train the neural network. The size of the replay buffer is the number of experiences that are fed into the network update. It therefore makes sense that the larger the buffer, the more experiences the agent can draw from and the rewards deviate less from the maximum experienced. Figure 6 demonstrates this decrease, and particularly the large jump from 1000 to 10000 replays. The standard deviation then increases slightly for the next increment which shows that 10000 is sufficient for the agent to draw from in this context, and requires less computational resources. The standard deviation was calculated by taking 67% of the maximum total reward across all runs (for different buffer size), then working out the std with this value rather than taking the difference between each reward and the mean shown in line 480.

2.3 Choice of k

The various learning curves show that for this time period of 200 episodes, fewer frames results in quicker learning from the agent and a higher mean reward. This is shown in figure 7 below It does however also show that for higher values of k the total rewards fluctuate less as the agent

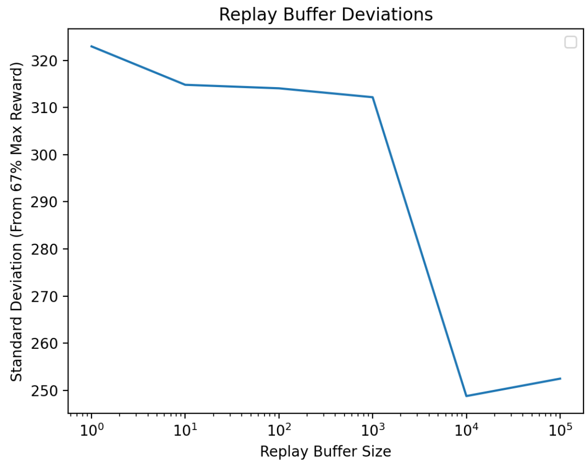


Figure 6: Standard Deviations from 67% of maximum reward (across all runs), against replay buffer size on logarithmic scale

acquires knowledge, giving a more stable learning curve. The speed of learning is expected as for lower k values the input dimension into the neural network is much smaller, with the hidden layers drawing inferences from fewer features. In practice the network architecture for each k value should be tuned to suit. In this example all values of k result in learning, and though k=1 appears optimal and so this value has been selected to maximise rewards. With more episodes we may have seen an improvement by using more frames, or perhaps by increasing the number of features into a flattened vector it makes the network too complex to draw intelligent features from in the neural net. It could be the case also that the CartPole environment doesn't benefit from this frame stacking as much as an Atari game using frames of pixel inputs and convolutional neural networks in comparison to our physics problem.

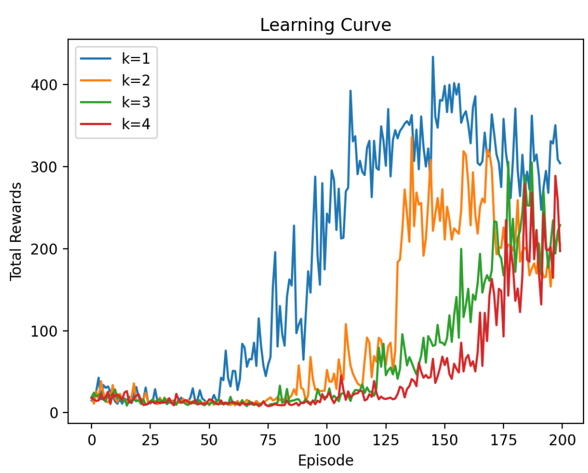


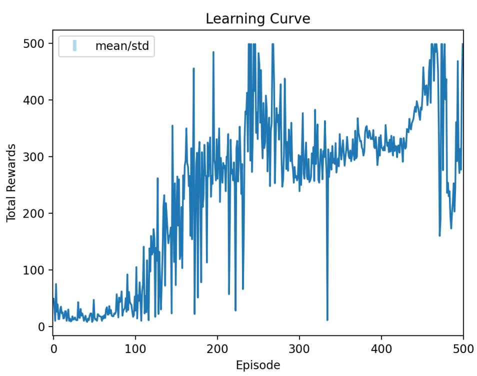
Figure 7: Variation of k on learning curve for DQL, each value of k was used for three runs and a mean value taken keeping all other hyperparameters constant

3 Question 3: Ablation/Augmentation experiments

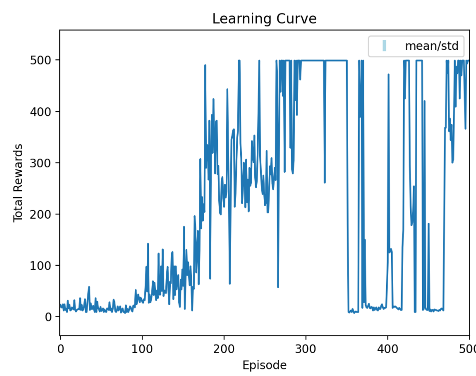
3.1 Double Deep Q-Learning Implementation

Q-learning when introduced in a Deep Reinforcement learning context is known to overestimate the action values in certain conditions. The Double Q learning algorithm is known to reduce susceptibility to this overestimation by adjusting the error calculation using the weights of the target network to fairly evaluate the value of this policy. The implementation can be seen in lines 124-127. If the DDQL condition is True, instead of selecting the next state values from doing a forward pass of the target network and selecting the maximum value, we select the maximum action index from using the policy network weights then use the second set of weights (of the target network) to fairly evaluate the value of this policy. Note that we can switch the roles of the target and policy set of weights symmetrically according to the Deepmind paper (Van Hesselt 2015) where the proposition of using DDQL is advocated.

However when experimenting with this I found huge variations between implementing DDQL when the target network and policy network roles are switched. Figure 8 demonstrates this, and actually the reverse case appears to get maximum rewards more often than the action selection method specified in the paper. I chose to implement the conventional one as it still shows an improvement on the DQL learning curve with more consistency. As for all of these experiments, more episodes and some form of early stopping would vastly improve the performance of an agent but for demonstration purposes the selected hyperparameters and run conditions will suffice.



(a) Target network evaluation, policy network action indices



(b) Policy network evaluation, target network action indices

Figure 8: DDQL Implementation Reverse

3.2 Learning Curve Comparison

The chosen value of k has a similar effect as for the single DQL network, this can be seen in figure 9 with k=1 clearly performing best and reaching maximum reward numerous times, again fluctuation may be seen as an issue but the agent evidently maximises rewards when using only 4 input features.

Comparing the different ablation/augmentation experiments we can draw a number of conclusions from incorporating additional features to our network. Firstly, removing the replay buffer makes this a simple temporal difference learning model taking information only from the last step. This results in some learning but the speed of learning is slow. We saw a similar effect in figure 6.

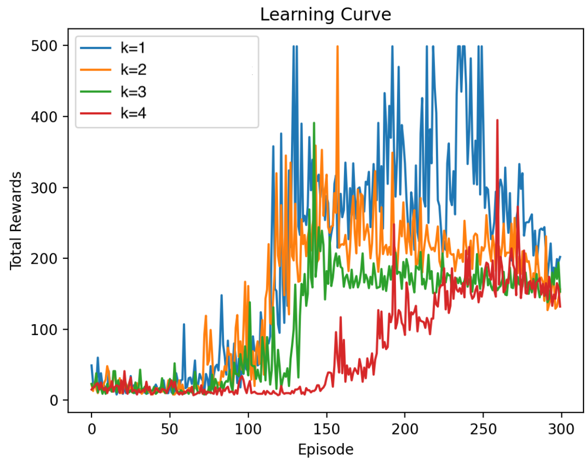


Figure 9: Variation of k on learning curve for DDQL, each value of k was used for three runs and a mean value taken keeping all other hyperparameters constant

For the other three curves we see the neural network with the replay seems far more robust and intelligent compared to its counterpart that only remembers the last action. Next we see removing the target network feature, removes the 'resonance damper' effect on the the normal curve, we see more fluctuations beyond the 150 episode mark, and while the curve may reach some of the maximum rewards experienced with the target network enabled, the curve doesn't converge with the normal or DDQN curves. This relaxation time can be seen to improve the performance and learning speed of the network. Implementing the DDQN can be seen to have positive effects on the learning of the agent, both in terms of speed and maximum reward. For a larger number of episodes as seen in figure 8a), we can see that the overall performance seems to increase to a high reward, hitting the maximum multiple times. We therefore can conclude that this method of ensuring no over-reliance on one network results on in a better network using DDQL.

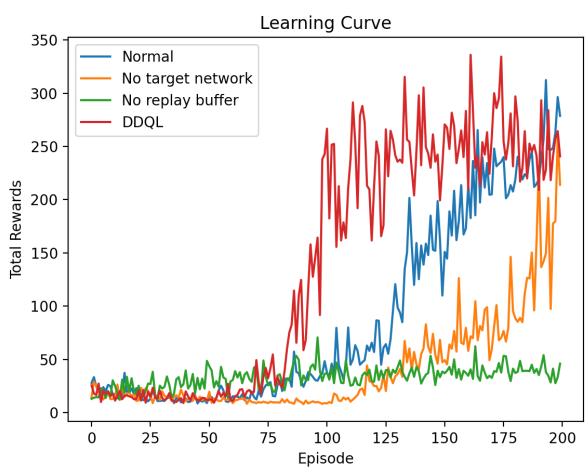


Figure 10: Learning curves incorporating ablation of target network, replay buffer and augmentation of DDQL against original DQL learning curve; using 3 runs and taking a mean score for each and k=1

A Appendix - Code

Notes on running the code:

- The code has been set up to run various experiments and therfore has been refactored so that there are only 4 lines of code to run at the bottom of the script (641 for a normal DQL learning curve, 650 to run the replay buffer experiement, 654 to run the k frames experiment and 658 to run the ablation/augmentation experiment). These will need to be uncommented to run, other than run_learning_curve which is left active for 10 runs
- Hyperparameters are actually set in the functions themselves which is not the best practice but convenient for running these experiments
- One input to the training loop is the 'record' toggle, which I have set to True by default now. Note that the error will be raised that video/video.mp4 filepath does not exist, this will need to be adjusted if you want to see the video to a valid filepath.
- There also exists separate plot functions, the inputs of which should be intuitive.

```
1  from IPython.display import clear_output
2  import gym
3  from gym.wrappers.monitoring.video_recorder import VideoRecorder  # records videos of episodes
4  import numpy as np
5  import matplotlib.pyplot as plt  # Graphical library
6  import torch
7  import torch.optim as optim
8  import torch.nn as nn
9  import torch.nn.functional as F
10 
11 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # Configuring Pytorch
12 from collections import namedtuple, deque
13 from itertools import count
14 import random
15 
16 clear_output()
17 
18 random.seed(10)
19 Transition = namedtuple('Transition',
20                         ('state', 'action', 'next_state', 'reward'))
21 
22 
23 class ReplayBuffer(object):
24 
25     def __init__(self, capacity):
26         self.memory = deque([], maxlen=capacity)
27 
28     def push(self, *args):
29         """Save a transition"""
```

```
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)


class DQN(nn.Module):

    def __init__(self, inputs, outputs, num_hidden, hidden_size):
        super(DQN, self).__init__()
        self.input_layer = nn.Linear(inputs, hidden_size)
        self.hidden_layers = nn.ModuleList([nn.Linear(hidden_size, hidden_size) \
                                    for _ in range(num_hidden - 1)])
        self.output_layer = nn.Linear(hidden_size, outputs)

    def forward(self, x):
        x.to(device)

        x = F.relu(self.input_layer(x))
        for layer in self.hidden_layers:
            x = F.relu(layer(x))

        return self.output_layer(x)


def optimize_model(memory,
                   BATCH_SIZE,
                   state_dim,
                   policy_net,
                   target_net,
                   GAMMA,
                   optimizer,
                   DDQL,
                   target_net_on):
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
```

```
    # (a final state would've been the one after which simulation ended)
    # Alteration from original code to account for k frame stacking, now
    # treats a state as final if the last 4 elements are equal to zero.
    non_final_mask = torch.tensor(tuple(map(lambda s: torch.sum(s[0][len(s[0]) - 4:])
                                        .absolute().item() > 0, batch.next_state)),
                                  device=device,
                                  dtype=torch.bool)

    # Can safely omit the condition below to check that not all states in the
    # sampled batch are terminal whenever the batch size is reasonable and
    # there is virtually no chance that all states in the sampled batch are
    # terminal. Similar change to account for k frame stacking final states
    if sum(non_final_mask) > 0:
        non_final_next_states = torch.cat([s for s in batch.next_state if torch.sum(
            s[0][len(s[0]) - 4:]).absolute().item() > 0])
    else:
        non_final_next_states = torch.empty(0, state_dim).to(device)

    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    actions_test = policy_net(state_batch)
    # Compute V(s_{t+1}) for all next states.
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)

    with torch.no_grad():
        # Once again can omit the condition if batch size is large enough
        if sum(non_final_mask) > 0:
            if not DDQL:
                # Single DQL case, with additional condition for using target
                # network updates (for final ablation question)
                if target_net_on:
                    next_state_values[non_final_mask] = target_net(
                        non_final_next_states).max(1)[0].detach()
                else:
                    next_state_values[non_final_mask] = policy_net(
                        non_final_next_states).max(1)[0].detach()
            else:
```

```
                # DDQL case, select action from policy network and then values from
                # target network
                policy_actions = \
                    policy_net(non_final_next_states).max(1)[1].view(-1, 1)
                next_state_values[non_final_mask] = \
                    target_net(non_final_next_states).gather(1, policy_actions).view(-1, )
        else:
            next_state_values = torch.zeros_like(next_state_values)

    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute loss using mean squared error loss criterion CHANGE IN REPORT
    criterion = nn.MSELoss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()

    # Limit magnitude of gradient for update step (THIS LIMITATION HAS BEEN REMOVED,
    # SLOWING DOWN RUN TIME BUT IMPROVING RESULTS)
    # for param in policy_net.parameters():
    #     param.grad.data.clamp_(-1, 1)

    optimizer.step()


def plot_total_rewards(N_episodes, total_rewards):
    plt.figure(2)
    episodes = np.arange(N_episodes)
    plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('Total Steps')
    plt.plot(episodes, np.array(total_rewards))
    print(f"Average Reward: {sum(total_rewards) / N_episodes}")


def select_action(state=None, current_eps=0, n_actions=2, policy_net=None):
    sample = random.random()
    eps_threshold = current_eps
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return policy_net(state).max(1)[1].view(1, 1)
```

```
    else:
        return torch.tensor([[random.randrange(n_actions)]],
                            device=device,
                            dtype=torch.long)


def eps_decay(NUM_EPISODES, EPS_START, EPS_END, decay_type):
    '''
    Function to produce different epsilon decay schedules
    '''

    eps_linear = np.linspace(EPS_START, EPS_END, NUM_EPISODES)
    eps_const = 0.6 * np.ones(eps_linear.shape)
    eps_glie = np.ones(eps_linear.shape)
    eps_factor = np.ones(eps_linear.shape)
    factor = np.power(EPS_END, EPS_START / NUM_EPISODES)

    for i in range(1, len(eps_linear)):
        eps_glie[i] = eps_glie[i - 1] / i
        eps_factor[i] = eps_factor[i - 1] * factor

    if decay_type == 'linear':
        return eps_linear
    elif decay_type == '1/k':
        return eps_glie
    elif decay_type == 'const':
        return eps_const
    else:
        return eps_factor


def train(NUM_EPISODES=100,
          BATCH_SIZE=128,
          GAMMA=0.99,
          EPS_START=0.9,
          EPS_END=0.05,
          EPS_DECAY='power',
          LR=0.0001,
          num_hidden_layers=2,
          size_hidden_layers=128,
          network_sync_freq=10,
          k=1,
          replays=10000,
          target_net_on=True,
          DDQL=False,
          record=False):
    '''
```

```python
    Main training loop for agent,
    input: all hyperparameters and ablation/augmentation toggles
    return: (np array) total rewards for each episode
    '''

    # Get number of states and actions from gym action space
    env = gym.make("CartPole-v1")
    env.reset()
    state_dim = k * len(env.state)  # x, x_dot, theta, theta_dot
    n_actions = env.action_space.n
    env.close()

    # Initialise two networks, policy net and identical target net
    policy_net = DQN(state_dim,
                     n_actions,
                     num_hidden_layers,
                     size_hidden_layers).to(device)
    target_net = DQN(state_dim,
                     n_actions,
                     num_hidden_layers,
                     size_hidden_layers).to(device)
    target_net.load_state_dict(policy_net.state_dict())
    target_net.eval()

    optimizer = optim.Adam(policy_net.parameters(), LR)
    memory = ReplayBuffer(replays)

    # Empty list to append total rewards for each episode to
    # (same as duration of episode)
    durations = []
    epsilon = EPS_START

    # Use custom function to get decay schedule
    eps_schedule = eps_decay(NUM_EPISODES, EPS_START, EPS_END, EPS_DECAY)

    for i_episode in range(NUM_EPISODES):

        if i_episode % 20 == 0:
            print("episode ", i_episode, "/", NUM_EPISODES)

        # Initialize the environment and state
        env.reset()
        state = torch.tensor(env.state).float().unsqueeze(0).to(device)

        # Initialise frame stacker and set first k frames as initial state
        k_states = deque([], maxlen=k)
        for i in range(k):
```

```
            k_states.append(state)

        for t in count():
            flattened_k_states = torch.stack(
                list(k_states)).reshape(-1).unsqueeze(0)  # Flatten
            action = select_action(flattened_k_states,
                                   epsilon,
                                   n_actions,
                                   policy_net)  # Select action from network
            _, reward, done, _ = env.step(action.item())
            reward = torch.tensor([reward], device=device)

            # Observe new state
            if not done:
                next_state = torch.tensor(
                    env.state).float().unsqueeze(0).to(device)
            else:
                # If terminal set next state as zeros
                next_state = torch.zeros_like(state)

            # Store the transition in memory
            # Append state to frame stacker deque, pushing out oldest frame
            k_states.append(next_state)
            next_flattened_k_states = torch.stack(
                list(k_states)).reshape(-1).unsqueeze(0)  # Flatten k states again
            memory.push(flattened_k_states,
                        action,
                        next_flattened_k_states,
                        reward)  # Push (s,a,s',r) to memory

            # Move to the next state
            state = next_state

            # Perform one step of the optimization (on the policy network)
            optimize_model(memory,
                           BATCH_SIZE,
                           state_dim,
                           policy_net,
                           target_net,
                           GAMMA,
                           optimizer,
                           DDQL,
                           target_net_on)
            if done:
                break

```

```
        # Move onto next epsilon value in schedule
        epsilon = eps_schedule[i_episode]

        durations.append(t)

        # Sync target network with policy net every set number of episodes
        if i_episode % network_sync_freq == 0:
            target_net.load_state_dict(policy_net.state_dict())

    print('Complete')

    env.close()

    # Plot rewards against episodes
    # plot_total_rewards(NUM_EPISODES, durations)
    # plt.show()

    # If record toggle on, record the episode
    if record:
        record_cart(policy_net, k)

    return durations


def record_cart(policy_net, k):
    '''
    Record function separated to be toggled in main training loop, if you want
    to see the video.
    '''
    env = gym.make("CartPole-v1")
    file_path = 'video/video.mp4'
    recorder = VideoRecorder(env, file_path)

    observation = env.reset()
    done = False

    state = torch.tensor(env.state).float().unsqueeze(0)
    k_states = deque([], maxlen=k)
    for i in range(k):
        k_states.append(state)

    duration = 0

    while not done:
        recorder.capture_frame()

        # Select and perform an action
```

```
        flattened_k_states = torch.stack(
            list(k_states)).reshape(-1).unsqueeze(0)
        action = select_action(flattened_k_states,
                               current_eps=0,
                               n_actions=2,
                               policy_net=policy_net)

        observation, reward, done, _ = env.step(action.item())
        duration += 1
        reward = torch.tensor([reward], device=device)

        # Observe new state
        state = torch.tensor(env.state).float().unsqueeze(0)
        k_states.append(state)

    recorder.close()
    env.close()
    print("Episode duration: ", duration)


def plot_learning_rate_full(total_rewards):
    '''
    Function to plot learning curves for a number of differnet training runs,
    taking mean and standard deviation and plotting this on the figure as a
    line with errorbars.
    Total rewards given as a numpy array (R x N) with each row as a full
    raining run under certain conditions:
    R = number of runs
    N = number of episodes per run
    '''
    n_episodes = np.shape(total_rewards)[1]
    episodes = np.arange(0, n_episodes, 1)

    # Take mean and std across diffent runs
    mean = np.mean(total_rewards, axis=0)
    std = np.std(total_rewards, axis=0)

    # MEAN
    x = episodes
    y1 = mean
    y2 = std

    # calculate polynomial
    z1 = np.polyfit(x, y1, 30)
    z2 = np.polyfit(x, y2, 30)
    f1 = np.poly1d(z1)
```

```
    f2 = np.poly1d(z2)

    # calculate new x's and y's
    x_new = np.linspace(x[0], x[-1], 500)
    y_new1 = f1(x_new)
    y_new2 = f2(x_new)

    plt.plot(x, mean, '-', markersize=1)
    plt.errorbar(x_new,
                 y_new1,
                 yerr=y_new2,
                 fmt='none',
                 color='blue',
                 ecolor='lightblue',
                 elinewidth=3,
                 capsize=0,
                 label="mean/std")

    plt.xlim([x[0] - 1, x[-1] + 1])
    plt.xlabel('Episode')
    plt.ylabel("Total Rewards")
    plt.legend()
    plt.title("Learning Curve")
    plt.show()


def plot_k_learning_rates(k_total_rewards):
    '''
    Function to plot mean learning curves for different values of k.
    k_total_rewards given as a numpy array (R x N) with each row as a full
    training run under certain conditions:
    R = number of runs
    N = number of episodes per run
    Also used for ablation and augmentation experiment
    '''

    # Select legend labels
    k_values = ["k=1", "k=2", "k=3", "k=4"]
    alterations = ["Normal", "No target network", "No replay buffer", "DDQL"]

    for i, total_rewards in enumerate(k_total_rewards):
        n_episodes = np.shape(total_rewards)[1]
        episodes = np.arange(0, n_episodes, 1)

        mean = np.mean(total_rewards, axis=0)

```

```
        x = episodes
        y = mean

        plt.plot(x, y, label=f"{alterations[i]}")

    plt.xlabel('Episode')
    plt.ylabel("Total Rewards")
    plt.legend()
    plt.title("Learning Curve")
    plt.show()


def plot_replay_deviation(replays_experiment, total_rewards):
    '''
    Function to plot standard deviation against size of replay buffer using a
    log x axis.
    total_rewards: given as a numpy array (R x N) with each row as a full
    training run under certain conditions
    R = number of runs
    N = number of episodes per run

    replays experiment: given as list of replay buffer sizes
    '''

    n_episodes = np.shape(total_rewards)[1]
    n_runs = np.shape(total_rewards)[0]
    max_reward = np.max(total_rewards)
    level = 0.67 * max_reward
    stds = []

    for run in range(n_runs):
        rewards = total_rewards[run, :]
        std = np.sqrt(np.sum((rewards - level) ** 2) / n_episodes)
        stds.append(std)

    x = replays_experiment
    y = np.array(stds)

    plt.plot(x, y)

    plt.xlabel('Replay Buffer Size')
    plt.xscale('log')
    plt.ylabel("Standard Deviation (From 67% Max Reward)")
    plt.legend()
    plt.title("Replay Buffer Deviations")
    plt.show()

```

```python

# LEARNING CURVE
def run_learning_curve(NUM_EPISODES, N_RUNS, k, replays, target, DDQL):
    '''
    Hyperparameters selected here. Train agent over specified number of N_RUNS
    and plot the averaged learning curve return total_rewards as an an (R x N)
    matrix:
    R = number of runs
    N = number of episodes per run
    '''
    total_rewards = np.zeros(NUM_EPISODES)

    for run in range(N_RUNS):
        print(f"Training run: {run}")
        rewards = train(NUM_EPISODES,
                        BATCH_SIZE=32,
                        GAMMA=0.99,
                        EPS_START=0.99,
                        EPS_END=0.05,
                        EPS_DECAY='power',
                        LR=0.0001,
                        num_hidden_layers=2,
                        size_hidden_layers=128,
                        network_sync_freq=10,
                        k=k,
                        replays=replays,
                        target_net_on=target,
                        DDQL=DDQL,
                        record=True)

        total_rewards = np.vstack((total_rewards, np.array(rewards)))

    total_rewards = np.delete(total_rewards, (0), axis=0)
    plot_learning_rate_full(total_rewards)

    return total_rewards


# REPLAYS EXPERIMENT
def run_replays_experiment(NUM_EPISODES, replays_experiment):
    '''
    Train agent over different replay buffer sizes as a list
    (replay_experiment), and plot results. Hyperparameters selected here.
    '''
    total_rewards = np.zeros(NUM_EPISODES)

    for run in range(len(replays_experiment)):
```

```
        replays = replays_experiment[run]
        print(f"Replay Buffer: {replays}")
        rewards = train(NUM_EPISODES=200,
                        BATCH_SIZE=32,
                        GAMMA=0.99,
                        EPS_START=0.99,
                        EPS_END=0.05,
                        EPS_DECAY='power',
                        LR=0.0001,
                        num_hidden_layers=2,
                        size_hidden_layers=128,
                        network_sync_freq=10,
                        k=3,
                        replays=replays,
                        DDQL=False,
                        record=False)

        total_rewards = np.vstack((total_rewards, np.array(rewards)))

    total_rewards = np.delete(total_rewards, (0), axis=0)
    plot_replay_deviation(replays_experiment, total_rewards)


# K EXPERIMENT
def run_k_experiment(NUM_EPISODES, N_RUNS, k_experiment):
    '''
    Train agent over different k values (number of frames to stack) as a list
    (k_experiment), run each for N_RUNS and plot mean results.
    '''
    k_total_rewards = []
    total_rewards = np.zeros(NUM_EPISODES)

    for run in range(len(k_experiment)):
        k = k_experiment[run]
        print(f"k value: {k}")

        total_rewards = run_learning_curve(NUM_EPISODES,
                                           N_RUNS,
                                           k,
                                           replays=10000,
                                           target=True,
                                           DDQL=True)
        k_total_rewards.append(total_rewards)

    plot_k_learning_rates(k_total_rewards)

```

```python

# ABLATION/AUGMENTATION EXPERIMENT
def run_ab_experiment(NUM_EPISODES, N_RUNS):
    '''
    Train agent ablating and augmenting different features. Run each for N_RUNS
    and plot all on same graph.
    '''
    altered_total_rewards = []
    total_rewards = np.zeros(NUM_EPISODES)

    for run in range(4):
        if run == 0:
            print("Normal run")
            total_rewards = run_learning_curve(NUM_EPISODES,
                                               N_RUNS,
                                               k=1,
                                               replays=10000,
                                               target=True,
                                               DDQL=False)
        elif run == 1:
            print("Ablating target network feature")
            total_rewards = run_learning_curve(NUM_EPISODES,
                                               N_RUNS,
                                               k=1,
                                               replays=10000,
                                               target=False,
                                               DDQL=False)
        elif run == 2:
            print("Ablating replay buffer")
            total_rewards = run_learning_curve(NUM_EPISODES,
                                               N_RUNS,
                                               k=1,
                                               replays=1,
                                               target=True,
                                               DDQL=False)
        elif run == 3:
            print("Implementing DDQN")
            total_rewards = run_learning_curve(NUM_EPISODES,
                                               N_RUNS,
                                               k=1,
                                               replays=10000,
                                               target=True,
                                               DDQL=True)

        altered_total_rewards.append(total_rewards)

    plot_k_learning_rates(altered_total_rewards)
```

```
# Run various 'experiments' in main run code below. Uncomment experiments that
# are not needed.

# Q1 IMPLEMENT DQN SOLUTION
run_learning_curve(NUM_EPISODES=200,
                   N_RUNS=10,
                   k=1,
                   replays=10000,
                   target=True,
                   DDQL=False)

# Q2 Hyperparameters of the DQN
# Run replays experiment for chosen parameters
# run_replays_experiment(NUM_EPISODES=200,
#                         replays_experiment = [1, 10, 100, 1000, 10000, 100000])

# Run k experiment for chosen parameters
# run_k_experiment(NUM_EPISODES=300, N_RUNS=1, k_experiment=[1, 2, 3, 4])

# Q3 Ablation/Augmentation Experiements
# Run ablation/augmentation experiment for chosen parameters
# run_ab_experiment(NUM_EPISODES=200, N_RUNS=3)
```