OReL Assignment 6

bkx591

March 2024

1 Deep Q-Learning (50 points)

1.1 Neural architecture (12 points)

The policy network definition contains the lines:

```
def forward(self, x_input):
    x = F.tanh(self.fc1(x_input))
    x = F.tanh(self.fc2(x))
    x = torch.cat((x_input, x), dim=1)
    x = self.output_layer(x)
    return x
```

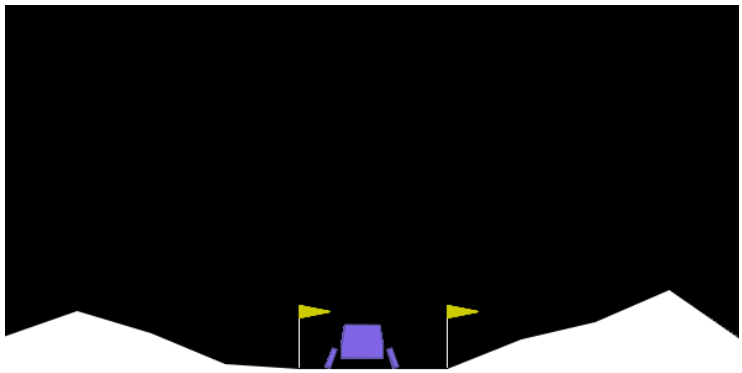

Figure 1: It landed!

1.1.1 Structure (4 points)

The line in at question , `x = torch.cat((x_input, x), dim=1)`, is concatenating the input tensor `x_input` and the output of the hidden layers `x` along

the feature tensor in PyTorch's tensors (dimension 1). Notice the input size of the output layer `self.output_layer =nn.Linear(hidden_size +state_size , action_size, bias)` is accounting for this. It combines the original state representation with the transformed stated, ater passing through two layers of network with `tanh` activations, before feeding it to the output layer. This means that the network can learn from both the raw and processed features, or in other words, both low- and high-level representations of the state.

1.1.2 Activation function (8 points)

What you probably want us to realize, is simply the output range of `tanh` being from -1 to 1 in contrast to a sigmoid functions output from 0 to 1. Considering the Lunar Lander environment, where actions can have opposite effects (left/right), a symmetric activation function like `tanh` can probably more naturally model these opposing actions. In terms of deep learning, I believe `tanh` has some interesting properties with the gradient. It can typically help with faster convergence and better mitigate the vanishing gradient problem than the sigmoid function because its derivatives values are larger and therefore less diluted in deeper networks. As this network is fairly simple/shallow, I'm not sure this is having that big of an impact. Nevertheless it was worth a mention as well.

1.2 Adding the Q-learning (16 points)

Adding the missing line is simply to add the computation of the targets following the application of the Bellman equation, or from the slides $r_i + \gamma \max_{a'} \left[\hat{Q}(s'_i)\right]_{a'}$

```
...
# Compute targets
targets_tensor = rewards_tensor + gamma * torch.max(target_Qs_tensor, dim=1).values
...
```

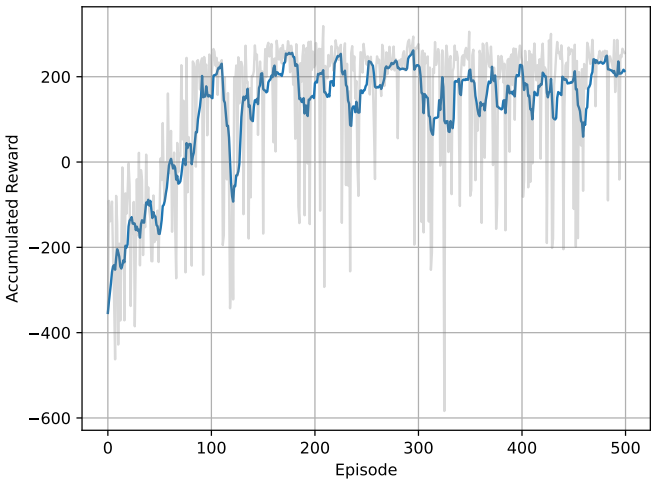

Figure 2:

1.3 Epsilon (4 points)

```
...
# Explore or exploit
explore_p = explore_stop + (explore_start - explore_stop)*np.exp(-decay_rate*ep)
...
```

As Christian talked about in the lecture, when implementing an ϵ-greedy approach, it is beneficial to add an exponential decay to reduce the number of random exploratory steps the algorithm makes, as time grows. Ideally we want an agent to explore and learn the environment sufficiently and to a certain extend. As the agent learns, we want it to exploit what it knows more frequently and explore less frequently. The `decay_rate` controls how fast the exploration rate decreases. This adaptive approach ensures that the agent continues to explore the environment to a certain degree, even as it increasingly relies on its learned policy, thereby preventing premature convergence to a suboptimal policy.

1.4 Gather (8 points)

```
...
# Only the Q values for the actions taken
Q_tensor = torch.gather(output_tensor, 1, actions_tensor.unsqueeze(-1)).squeeze()
...
```

This operation selects the Q-values from the `output_tensor` corresponding to the actions taken, as indicated by the `actions_tensor`. It is a crucial operation

for calculating the loss only for the actions that were actually taken in each states, instead of all possible actions. The `actions_tensor` is batch-sampled from memory, and for training, we only want to update the Q-values for chosen actions - to better match the calculated targets.

1.5 No grad (10 points)

```
...
with torch.no_grad():
...
```

This function disables gradient calculations, making code run faster, using less memory, and is typically employed during the evaluation or inference phase - when it is necessary to perform operations on tensors without keeping track of the operations for backpropagation. In this setting, for Q-learning, it is used when the target Q-values are being calculated. Since the targets are based on the current policy and are used as a baseline to evaluate the current outputs, they should not influence the gradient calculations when updating the network's weights. Therefore the targets are calculated in the context of this function. If they did, affect the network's weights, it could lead to a divergent learning behavior, as it would be chasing a moving target that it's simultaneously influencing.

2 A tighter analysis of the Hedge algorithm (20 points)

Given the potential function W_t as defined in the proof in the slides:

$$W_t = \sum_a e^{-\eta L_t(a)}$$

this W_t represents the total weight of all actions at time t, where each action's weight is exponentially decaysed by its cumulative loss L_t up to time t, scaled by the learning rate η. Following the steps from the slides, we'll examine the ratio of consecutive potential functions:

$$\frac{W_t}{W_{t-1}} = \frac{\sum_a e^{-\eta L_t(a)}}{\sum_{a'} e^{-\eta L_{t-1}(a')}}$$

Here, $L_t(a)$ is the cumulative loss for action a at time t, and $L_{t-1}(a')$ is the cumulative loss for action a' up to time $t-1$. We define the probability $p_t(a)$ as given in the slides:

$$p_t(a) = \frac{e^{-\eta L_{t-1}(a)}}{\sum_{a'} e^{-\eta L_{t-1}(a')}}$$

and recognize that:

$$\frac{W_t}{W_{t-1}} = \sum_a p_t(a) e^{-\eta \hat{l}_t(a)} \tag{1}$$

$$= \mathbb{E}[e^{-\eta \hat{l}_t(a)}] \tag{2}$$

where $\hat{l}_t(a)$ is the loss incurred by action a at time t. This is the point where Hoeffding's Lemma is applied to bound the expectation of the exponent, as given by hint (a). If we denote the loss $\hat{l}_t(a)$ by X and use Hoeffding's Lemma:

$$\mathbb{E}[e^{\lambda X}] \leq e^{\mathbb{E}[X] + \frac{\lambda^2 (b-a)^2}{8}}$$

Given that $\hat{l}_t(a) \in [0, 1]$, the lemma can be applied with $\lambda = -\eta$, $a = 0$, and $b = 1$:

$$\mathbb{E}[e^{-\eta \hat{l}_t(a)}] \leq e^{-\eta \mathbb{E}[\hat{l}_t(a)] + \frac{\eta^2}{8}}$$

Substituting this into the weight ratio, we get:

$$\frac{W_t}{W_{t-1}} \leq e^{-\eta \mathbb{E}[\hat{l}_t(a)] + \frac{\eta^2}{8}}$$

Such that

$$\frac{W_T}{W_0} = \frac{W_1}{W_0} \frac{W_2}{W_1} ... \frac{W_T}{W_{T-1}} \tag{3}$$

$$\leq e^{-\eta \sum_{t-1}^{T} \mathbb{E}[\hat{l}_t(a)] + \sum_{t-1}^{T} \frac{\eta^2}{8}} \tag{4}$$

$$= e^{-\eta \sum_{t-1}^{T} \mathbb{E}[\hat{l}_t(a)] + \frac{T\eta^2}{8}} \tag{5}$$

$$\tag{6}$$

and

$$\frac{W_T}{W_0} \geq \frac{e^{-\eta L_t(a^*)}}{K}$$

Combining both sides, taking the natural logarithm and normalizing with respect to η gives us a regret bound with equal upper and lower bound (which is rare) s.t.:

$$\frac{e^{-\eta L_t(a^*)}}{K} \leq e^{-\eta \sum_{t-1}^{T} \mathbb{E}[\hat{l}_t(a)] + \frac{T\eta^2}{8}} \tag{7}$$

$$-\eta L_T(a^*) - \ln K \leq -\eta \sum_{t=1}^{T} \mathbb{E}[\hat{l}_t(a)] + \frac{T\eta^2}{8} \tag{8}$$

$$-L_t(a^*) - \frac{\ln K}{\eta} \leq -\sum_{t=1}^{T} \mathbb{E}[\hat{l}_t(a)] + \frac{T\eta^2}{8} \tag{9}$$

$$\sum_{t=1}^{T} \mathbb{E}[\hat{l}_t(a)] - L_t(a^*) \leq \frac{\ln K}{\eta} + \frac{T\eta}{8} \tag{10}$$

And to find the value of η that minimizes this bound:

$$\frac{d}{d\eta}\left(\frac{\ln K}{\eta}+\frac{\eta T}{8}\right)=-\frac{\ln K}{\eta^2}+\frac{T}{8}=0$$

$$\eta_{\min}=\sqrt{\frac{8\ln K}{T}}$$

$$\frac{d^2}{d\eta^2}\left(\frac{\ln K}{\eta}+\frac{\eta T}{8}\right)=\frac{2\ln K}{\eta^3}>0$$

s.t.

$$\mathbb{E}[R_T]\leq\frac{\ln K}{\sqrt{\frac{8\ln K}{T}}}+\frac{T\sqrt{\frac{8\ln K}{T}}}{8} \tag{11}$$

$$=\frac{\ln K}{\sqrt{\frac{8\ln K}{T}}}+\frac{T\sqrt{\frac{8\ln K}{T}}}{8} \tag{12}$$

$$=\sqrt{\frac{8\ln K}{T}}+\sqrt{\frac{8\ln K}{T}} \tag{13}$$

$$=\sqrt{\frac{1}{2}T\ln K} \tag{14}$$

3 Empirical evaluation of algorithms for adversarial environments (5 points)

Evaluating algorithms in adversarial environments can be conducted by simulating conditions where the algorithm's decisions are systematically countered by an opposing entity. The primary measure of quality in such experiments, would be performance stability, quantified through regret as we have done so far. We must run the simulation through a statistically significant number of trials, against various adversarial strategies, by an opposing entity or entities, to ascertain that the algorithm is in fact feasible in most possible variations. Two examples could be hide and seek and cat and mouse. In which the reward for one comes at the expense of the other.

4 Empirical comparison of UCB1 and EXP3 algorithms (25 points)

Performance comparison

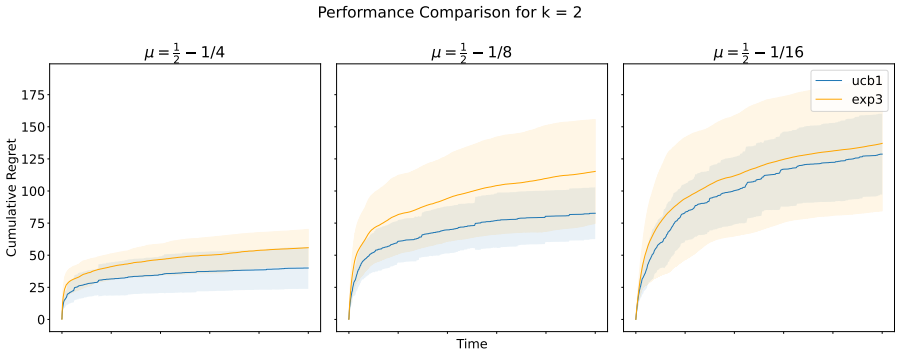

Figure 3: UCB1 and EXP3 plotted over $T = 100000$ normalized over 20 runs. For K=2 we see UCB1 accumulate less regret yet proportional to EXP3.

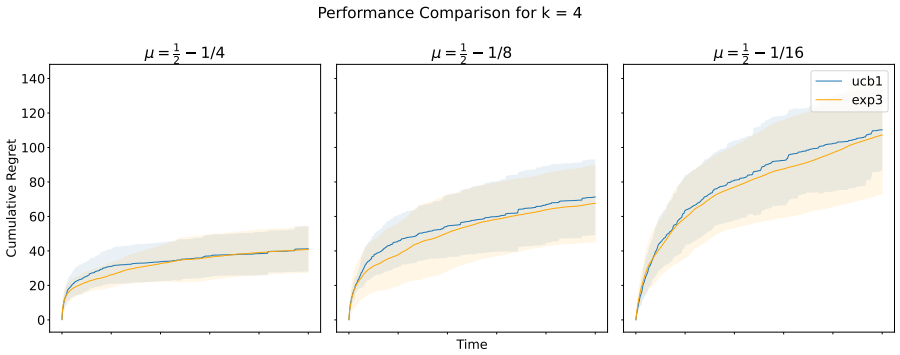

Figure 4: UCB1 and EXP3 plotted over $T = 100000$ normalized over 20 runs. For K=4 we see UCB1 and EXP3 accumulate regret almost identically.

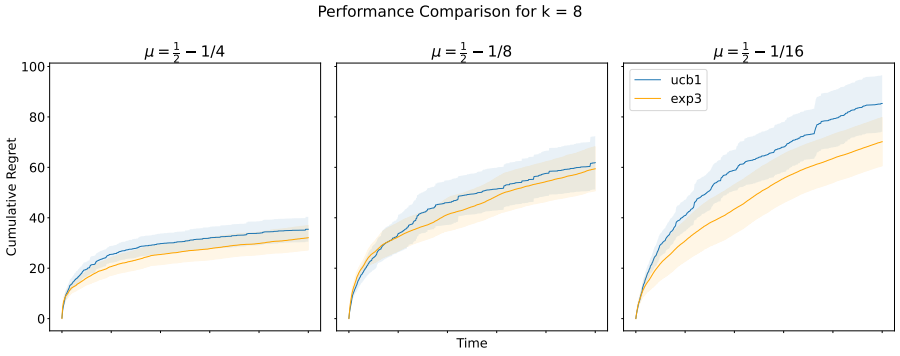

Figure 5: UCB1 and EXP3 plotted over $T = 100000$ normalized over 20 runs. For K=8 and low variance in optimality of arms we see a divergence in accumulated regret with low delta. Though in Figure 6 we see something else.

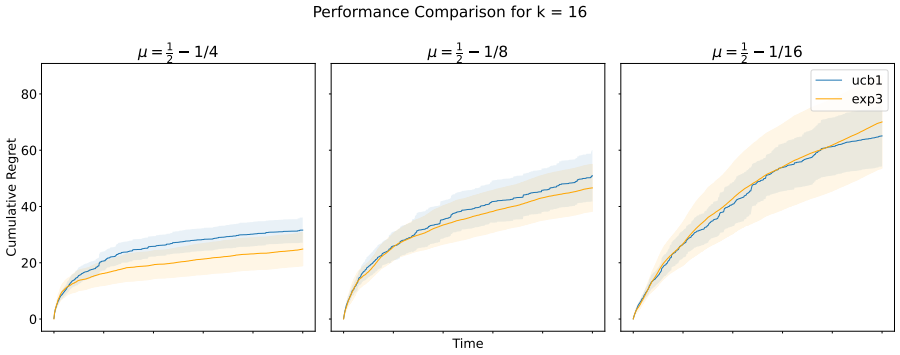

Figure 6: UCB1 and EXP3 plotted over $T = 100000$ normalized over 20 runs. For K=16 we see UCB1 accumulate regret almost exactly like EXP3.

Adversarial Sequence

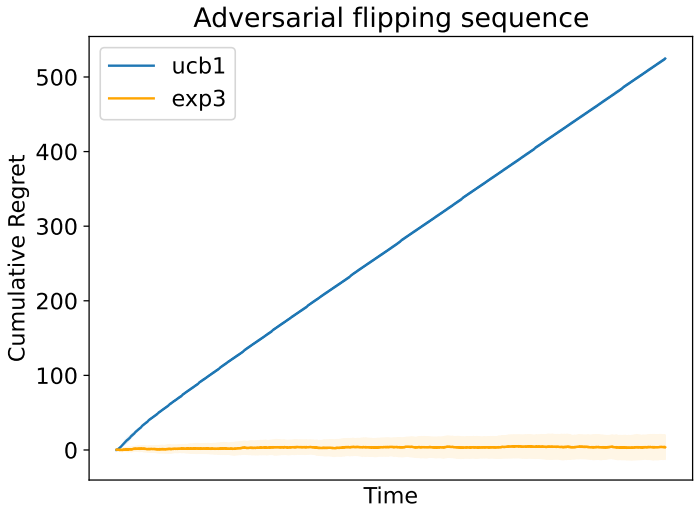

Figure 7: UCB1 and EXP3 plotted over $T = 100000$ normalized over 20 runs on an adversarial sequence of continuously flipping bias, resulting in linear regret for the UCB1. The reward probability of the optimal arm is 1

For my adversarial sequence, I designed it such that the optimal arm has a bias $\mu^* = 1.0$ and all other arms have $\mu = 0.0$ - initially. With any other value of μ this does not give linear pseudo regret. For each time step, I flip the reward proabilities (*np.roll*) which introduces a non-stationarity that systematically misleads UCB1, which relies on historical reward data to estimate the potential of each arm. Since UCB1 tends to exploit what it believes to be the best arm based on past rewards, the flipping mechanism ensures that the once optimal choice becomes suboptimal in subsequent rounds, leading to linear regret. Conversely, EXP3's probabilistic exploration is less impacted by the changing probabilities, allowing it to adapt and converge despite the adversarial conditions. When we play around with μ we suffer much more standard deviation over the number of repetitions. For $\mu = 1$ I got a max std of 3.6, for $\mu = 0.8$, this is much higher, which is seen in Figure 8.

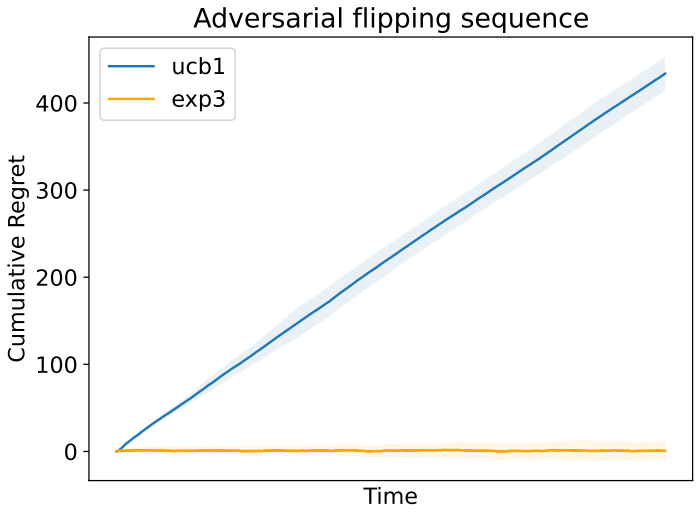

Figure 8: UCB1 and EXP3 plotted over $T = 100000$ normalized over 20 runs on an adversarial sequence of continuously flipping bias, resulting in linear regret for the UCB1. The reward probability of the optimal arm is 0.8