COMPSCI 687 Homework 2 - Fall 2024

Due **October 15, 2024**, 11:55pm Eastern Time

SHREYA BIRTHARE

1 Instructions

This homework assignment consists of a written portion and a programming portion. While you may discuss problems with your peers (e.g., to discuss high-level approaches), you must answer the questions on your own. In your submission, do explicitly list all students with whom you discussed this assignment. Submissions must be typed (handwritten and scanned submissions will not be accepted). You must use LaTeX. The assignment should be submitted on Gradescope as a PDF with marked answers via the Gradescope interface. The source code should be submitted via the Gradescope programming assignment as a .zip file. Include with your source code instructions for how to run your code. You **must** use Python 3 for your homework code. You may not use any reinforcement learning or machine learning-specific libraries in your code, e.g., TensorFlow, PyTorch, or scikit-learn. You *may* use libraries like numpy and matplotlib, though. The automated system will not accept assignments after 11:55pm on October 15. The tex file and starter code for this homework can be found here.

Before starting this homework, please review this course's policies on plagiarism by reading Section 10 of the syllabus.

Part One: Written (54 Points Total)

1. (**10 Points**) In class, we presented one possible Bellman equation for q^π:

$$q^\pi(s,a) = R(s,a) + \gamma \sum_{s'} p(s,a,s')\, v^\pi(s').$$

The equation above characterizes the expected return if the agent is in state s, takes action a, and continues following policy π thereafter; that is, $\mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \,|\, S_t = s,\, A_t = a,\, \pi]$.

We now wish to characterize the expected return/performance of an agent assuming it commits to executing not *one* action, as above, but *two* actions. In particular, assume that the agent is in state a, executes action a, transitions to some other state, executes action a', and follows π thereafter. To investigate this setting, let us define a new function, q_2^π, that characterizes the expected return in the above-mentioned case. Concretely,

$$q_2^\pi(s,a,a') := \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \,|\, S_t = s,\, A_t = a,\, A_{t+1} = a',\, \pi\right].$$

Derive the Bellman equation for q_2^π from first principles. The equation should be expressed in terms of $q^\pi(s,a)$ and should only involve $\mathcal{S}, \mathcal{A}, p, R, d_0, \gamma$, and π. Your solution should include a *step-by-step* explanation of how you arrived at this Bellman equation.

Hint: Start from the definition of q_2^π and use the definitions and properties of probability distributions discussed in Homework 1, as well as the Markov Property (when appropriate).

$$\begin{aligned} q_2^\pi(s,a,a') :=& \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \,|\, S_t = s,\, A_t = a,\, A_{t+1} = a',\, \pi\right] \\ &= \sum_{s' \in \mathcal{S}} \Pr(S_{t+1} = s' | S_t = s, A_t = a, A_{t+1} = a') * \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \,|\, S_t = s,\, A_t = a,\, A_{t+1} = a', S_{t+1} = s',\, \pi\right] \end{aligned} \tag{1}$$

Solving $\mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k} \,|\, S_t=s,\, A_t=a,\, A_{t+1}=a', S_{t+1}=s',\, \pi\right]$ first,

$$=\mathbb{E}\left[R_t+\sum_{k=1}^{\infty}\gamma^k R_{t+k} \,|\, S_t=s,\, A_t=a,\, A_{t+1}=a', S_{t+1}=s',\, \pi\right]$$

$$=\mathbb{E}\left[R_t+\gamma*\sum_{k=0}^{\infty}\gamma^k R_{(t+1)+k} \,|\, S_t=s,\, A_t=a,\, A_{t+1}=a', S_{t+1}=s',\, \pi\right]$$

$$=\mathbb{E}\left[R_t| \, S_t=s,\, A_t=a,\, A_{t+1}=a', S_{t+1}=s',\, \pi\right]+\gamma*\mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k R_{(t+1)+k} \,|A_{t+1}=a', S_{t+1}=s',\, \pi\right]$$

$$=\mathbb{E}\left[R_t| \, S_t=s,\, A_t=a,\, A_{t+1}=a', S_{t+1}=s',\, \pi\right]+\gamma q^{\pi}(s',a')$$

$$=R(s,a)+\gamma q^{\pi}(s',a') \tag{2}$$

(The reward at time **t** only depends on the action taken from a state at that time and not on the next state and action)

Substituting equation (2) in equation (1):

$$\sum_{s'\in\mathcal{S}}\Pr(S_{t+1}=s'|S_t=s,A_t=a,A_{t+1}=a')*\mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k} \,|\, S_t=s,\, A_t=a,\, A_{t+1}=a', S_{t+1}=s',\, \pi\right]$$

$$=\sum_{s'\in\mathcal{S}}\Pr(S_{t+1}=s'|S_t=s,A_t=a,A_{t+1}=a')*\left(R(s,a)+\gamma q^{\pi}(s',a')\right) \tag{3}$$

Solving $\sum_{s'\in\mathcal{S}}\Pr(S_{t+1}=s'|S_t=s,A_t=a,A_{t+1}=a')$ now by using Bayes' Theorem,

$$=\sum_{s'\in\mathcal{S}}\frac{\Pr(A_{t+1}=a'|S_{t+1}=s',S_t=s,A_t=a)\Pr(S_{t+1}=s'|S_t=s,A_t=a)}{\Pr(A_{t+1}=a'|S_t=s,A_t=a)}$$

$$=\sum_{s'\in\mathcal{S}}\frac{\pi(s',a')p(s,a,s')}{\sum_{s'\in\mathcal{S}}\Pr(S_{t+1}=s'|S_t=s,A_t=a)\Pr(A_{t+1}=a'|S_{t+1}=s',S_t=s,A_t=a)}$$

$$=\sum_{s'\in\mathcal{S}}\frac{\pi(s',a')p(s,a,s')}{\sum_{s'\in\mathcal{S}}p(s,a,s')\pi(s',a')} \tag{4}$$

Substituting equation (4) in equation (3):

$$=\sum_{s'\in\mathcal{S}}\frac{\pi(s',a')p(s,a,s')}{\sum_{s'\in\mathcal{S}}p(s,a,s')\pi(s',a')}*\left(R(s,a)+\gamma q^{\pi}(s',a')\right)$$

$$=\sum_{s'\in\mathcal{S}}\frac{\pi(s',a')p(s,a,s')}{\sum_{s'\in\mathcal{S}}p(s,a,s')\pi(s',a')}*R(s,a)+\sum_{s'\in\mathcal{S}}\frac{\pi(s',a')p(s,a,s')}{\sum_{s'\in\mathcal{S}}p(s,a,s')\pi(s',a')}*\gamma q^{\pi}(s',a')$$

$$=R(s,a)\frac{\sum_{s'\in\mathcal{S}}\pi(s',a')p(s,a,s')}{\sum_{s'\in\mathcal{S}}p(s,a,s')\pi(s',a')}*1+\frac{\sum_{s'\in\mathcal{S}}\pi(s',a')p(s,a,s')}{\sum_{s'\in\mathcal{S}}p(s,a,s')\pi(s',a')}*\gamma q^{\pi}(s',a')$$

(R(s,a) does not depend on summation over s'.)

(Also sending outer summation inside the fraction)

$$=R(s,a)+\gamma*\frac{\sum_{s'\in\mathcal{S}}\pi(s',a')p(s,a,s')q^{\pi}(s',a')}{\sum_{s'\in\mathcal{S}}p(s,a,s')\pi(s',a')}$$

2. (**8 Points**) Recall the Bellman Equation for v^{π} we discussed in class:

$$v^{\pi}(s)=\sum_{a}\pi(s,a)\sum_{s'}p(s,a,s')\left(R(s,a)\,+\,\gamma v^{\pi}(s')\right).$$

This equation assumes a reward function of the form $R(s, a)$, meaning it depends only on the current state and action. However, in many cases, we may want to reward the agent based on the state it reaches. In such situations, we use reward functions of the form $R(s, a, s')$.

Derive (from first principles) the Bellman equation for v^π in this setting, where the reward function is defined as $R(s, a, s') := \mathbb{E}[R_t \mid S_t = s,\ A_t = a,\ S_{t+1} = s',\ \pi]$.

As before, your Bellman equation should only involve $\mathcal{S}, \mathcal{A}, p, R, d_0, \gamma$, and π. Your solution should include a *step-by-step* explanation of how you arrived at this equation.

$$\begin{aligned} v^\pi(s) =& \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} | S_t = s, \pi\right] \\ =& \mathbb{E}\left[R_t + \sum_{k=1}^{\infty} \gamma^k R_{t+k} | S_t = s, \pi\right] \\ =& \mathbb{E}\left[R_t + \gamma \sum_{k=0}^{\infty} \gamma^k R_{(t+1)+k} | S_t = s, \pi\right] \\ =& \mathbb{E}[R_t | S_t = s, \pi] + \gamma \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{(t+1)+k} | S_t = s, \pi\right] \end{aligned} \tag{5}$$

Solving $\mathbb{E}[R_t | S_t = s, \pi]$,

$$\begin{aligned} &= \sum_{a \in \mathcal{A}} \Pr(A_t = a | S_t = s, \pi) \mathbb{E}[R_t | S_t = s, A_t = a, \pi] \\ &= \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} \Pr(S_{t+1} = s' | S_t = s, A_t = a) \mathbb{E}[R_t | S_t = s, A_t = a, S_{t+1} = s', \pi] \\ &= \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} p(s, a, s') R(s, a, s') \end{aligned} \tag{6}$$

Solving $\mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{(t+1)+k} | S_t = s, \pi\right]$ now,

$$\begin{aligned} &= \sum_{a \in \mathcal{A}} \Pr(A_t = a | S_t = s, \pi) \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{(t+1)+k} | S_t = s, A_t = a, \pi\right] \\ &= \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} \Pr(S_{t+1} = s' | S_t = s, A_t = a) \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{(t+1)+k} | S_t = s, A_t = a, S_{t+1} = s', \pi\right] \\ &= \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} p(s, a, s') \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{(t+1)+k} | S_{t+1} = s', \pi\right] \\ &= \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} p(s, a, s') v^\pi(s') \end{aligned} \tag{7}$$

Substituting equation (6) and (7) in equation (5):

$$\begin{aligned} &= \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} p(s, a, s') R(s, a, s') + \gamma \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} p(s, a, s') v^\pi(s') \\ &= \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} p(s, a, s') \Big(R(s, a, s') + \gamma v^\pi(s') \Big) \end{aligned}$$

3. (**10 Points**) In class, we discussed two ways of defining policy optimality:

   - **Value Function-Based Optimality**: π^* is an optimal policy iff $\pi^* \geq \pi$, for all $\pi \in \Pi$. This holds iff $v^{\pi^*}(s) \geq v^\pi(s)$, for all $\pi \in \Pi$ and for all $s \in \mathcal{S}$.
   - **J-based Optimality**: π^* is an optimal policy iff $\pi^* \in \arg\max_{\pi \in \Pi} J(\pi)$, where $J(\pi) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_k \mid \pi]$.

Note that, in general, a policy that is optimal under one definition may not necessarily be optimal under another. Let Π be the set of all possible policies. Prove that if $\pi \geq \pi'$ for all $\pi' \in \Pi$, then it necessarily follows that $J(\pi) = \max_{\pi' \in \Pi} J(\pi')$. That is, prove that value function-based optimality implies J-based optimality.

Hint: Start from the definition of J (in terms of an expectation) and rewrite it so it explicitly accounts for d_0.

$$\begin{aligned} J(\pi) =& \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_k \,|\, \pi\right] \\ &= \sum_{s_0 \in \mathcal{S}} \Pr(S_0 = s_0) \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_k \,|\, S_0 = s_0, \pi\right] \\ &= \sum_{s_0 \in \mathcal{S}} d_0 v^{\pi}(s_0) \end{aligned} \tag{8}$$

Now, given that $\pi \geq \pi'$ for all $\pi' \in \Pi$, that means π is an optimal policy. So, $v^{\pi}(s) \geq v^{\pi'}(s)$ for all $\pi' \in \Pi$ and for all $s \in \mathcal{S}$ i.e (v^{π} is optimal value function). As this holds true for all states then starting from any initial state s_0, this should hold true as well i.e.

$= v^{\pi}(s_0) \geq v^{\pi'}(s_0)$

$= \sum_{s_0 \in \mathcal{S}} d_0 v^{\pi}(s_0) \geq \sum_{s_0 \in \mathcal{S}} d_0 v^{\pi'}(s_0)$

(No matter from what initial state we start, optimal value function of policy π will be greater than all. Doing summation over starting states still hold this inequality as it is optimal value function)

$= J(\pi) \geq J(\pi')$

(Substituting equation (8) here)

This means that the objective function value for policy π is the greatest amongst all objective function value for all policies π'. i.e.

$$J(\pi) = \max_{\pi' \in \Pi} J(\pi') \text{So, we can say}$$

$$\pi \in \arg\max_{\pi' \in \Pi} J(\pi')$$

This implies that π is J-based optimal policy.

4. **(6 Points)** Prove that although many optimal policies may exist, they all share the same optimal action-value function, q^*. You can find hints on how to start by reviewing the material covered in lecture 08.

Let $\pi_1, \pi_2 \in \Pi$ both be optimal policies and Π be a set of all possible policies. Let A be the set of action space and S be the set of state space.
By definition of optimal value function, $v^{\pi_1}(s) \geq v^{\pi_2}(s)$ and $v^{\pi_2}(s) \geq v^{\pi_1}(s)$ for a state s. This is only possible when $v^{\pi_1}(s) = v^{\pi_2}(s)$.
Now we can also represent optimal value function $v^{\pi}(s)$ as: $\sum_{a \in A} \pi(s,a) q(s,a)$ where q is optimal action value function. So,

$v^{\pi_1}(s) = v^{\pi_2}(s)$

$=: \sum_{a \in A} \pi_1(s,a) q^{\pi_1}(s,a) = \sum_{a \in A} \pi_2(s,a) q^{\pi_2}(s,a)$

If a policy is optimal then for all actions 'a', it'll only take the action that gives maximum reward.

So, $\sum_{a \in A} \pi_1(s,a) = \sum_{a \in A} \pi_2(s,a) = 1$

$=: 1 * q^{\pi_1}(s,a) = 1 * q^{\pi_2}(s,a)$

$=: q^{\pi_1}(s,a) = q^{\pi_2}(s,a)$

This proves that both the optimal policies have the same optimal action-value function.

5. **(5 Points)** In class, we discussed how to construct/characterize the optimal policy using knowledge of the optimal state-value function (v^*) or action-value function (q^*). Prove that if $\gamma = 0$, π^* be can be expressed analytically (i.e., via a closed-form mathematical expression) using *only* knowledge about the MDP itself—that is, only $\mathcal{S}, \mathcal{A}, p, R, d_0$, and γ.

Let A be the set of action space and S be the set of state space and Π be a set of policies We know that for any policy π, its value function is:

$$v^\pi = \sum_{a\in A} \pi(s,a) \sum_{s'\in S} p(s,a,s') \Big(R(s,a) + \gamma v^\pi(s') \Big)$$
$$q^\pi(s,a) = R(s,a) + \gamma \sum_{s'\in S} p(s,a,s') \sum_{a'\in A} \pi(s',a') q^\pi(s',a')$$
$$q^\pi(s,a) = R(s,a) + \gamma \sum_{s'\in S} p(s,a,s') v^\pi(s')$$
$$v^\pi(s) = \sum_{a\in A} \pi(s,a) q^\pi(s,a)$$

So for any optimal policy π^*, we have:

$$v^{\pi^*}(s) = \sum_{a\in A} \pi^*(s,a) q^{\pi^*}(s,a)$$

If a policy is optimal, then $\sum_{a\in A} \pi^*(s,a) = 1$, i.e., it always takes the action with the maximum reward. It doesn't take the rest of the actions that result in lesser rewards. Therefore,

$$v^{\pi^*}(s) = max_{a\in A} q^{\pi^*}(s,a) \tag{9}$$

$$\begin{aligned} q^{\pi*}(s,a) =& R(s,a) + \gamma \sum_{s'\in S} p(s,a,s') max_{a'\in A} q^{\pi*}(s',a') \\ =& \sum_{s'\in S} p(s,a,s') \Big(R(s,a) + \gamma * max_{a'\in A} q^{\pi*}(s',a') \Big) \\ & (\text{As} \sum_{s'\in S} p(s,a,s') = 1) \end{aligned} \tag{10}$$

Substituting equation (10) in (9):

$$\begin{aligned} v^{\pi^*}(s) =& max_{a\in A} \sum_{s'\in S} p(s,a,s') \Big(R(s,a) + \gamma * max_{a'\in A} q^{\pi*}(s',a') \Big) \\ =& max_{a\in A} \sum_{s'\in S} p(s,a,s')(R(s,a)) \quad (\text{Given } \gamma = 0) \\ =& max_{a\in A} R(s,a) \sum_{s'\in S} p(s,a,s') \\ =& max_{a\in A} R(s,a) * 1 \\ =& max_{a\in A} R(s,a) \end{aligned}$$

6. **(15 Points)** This question is divided into two parts.

   (a) **[5 Points]** First, let $R_{\text{Max}} := \max_{s\in\mathcal{S}} \max_{a\in\mathcal{A}} R(s,a)$. What is the maximum theoretical value of a state, $v^\pi(s)$, assuming a setting where returns are discounted and $\gamma < 1$?
   Hint: Start by writing the definition of $v^\pi(s)$ in terms of an expected value. Now consider what the "best-case scenario" would be in terms of the rewards that the agent could receive at any given time step.

Given that at any state S, performing any action a, the max reward we can get is R(s,a). Then in the best case scenario reward for any action on any state will be R(s,a). We know that:

$$
\begin{aligned}
v^{\pi}(s) =& \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^{k}R_{t+k}, |S_t = s, \pi\right] \\
=& \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^{k}R_{max}, |S_t = s, \pi\right] \\
=& \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^{k}R_{max}\right] \quad \text{(state and policy do not affect reward in best case)} \\
=& R_{max}\mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^{k}\right] \\
=& R_{max} * \frac{1}{1-\gamma} \\
=& \frac{R_{max}}{1-\gamma}
\end{aligned}
$$

(b) **[10 Points]** In supervised ML, normalizing data and features—such as when training a neural network—is often beneficial. Here, we will explore whether normalization techniques can also be advantageous in RL. Specifically, we will examine if *normalizing the reward function* can lead to improved (or potentially worse) performance. Let us define the normalized reward function as $R'(s,a) := \frac{R(s,a)}{R_{\text{Max}}}$, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Let π^* be an optimal policy with respect to the original reward function R. Here, optimality refers to value function-based optimality. Can π^* be *suboptimal* with respect to the normalized reward function R'?
Hint: Start by writing the definition of $v^{\pi}(s)$ under R, for any policy $\pi \in \Pi$, in terms of an expected value. Next, consider how this quantity would change for any policy, including the optimal one, if we replace R with R'. Think about the impact of this change in the context of the definition of value function-based optimality.
Here given that π^* is optimal then for all policies $\pi \in \Pi$, $v^{\pi^*}(s) \geq v^{\pi}(s)$. In the previous question we proved that if its an optimal policy then $v^{\pi^*}(s) = \sum_{a\in A}\pi^*(s,a)q^{\pi^*}(s,a)$ So,

$$
\begin{aligned}
&= \sum_{a\in A}\pi^*(s,a)R(s,a) + \gamma\sum_{a\in A}\pi^*(s,a)\sum_{s' in S}p(s,a,s')\sum_{a'\in A}\pi^*(s',a')R(s',a') + \\
&\gamma^2\sum_{a\in A}\pi^*(s,a)R(s,a) + \gamma\sum_{a\in A}\pi^*(s,a)\sum_{s' in S}p(s,a,s')\sum_{a'\in A}\pi^*(s',a')\sum_{s''\in S}p(s',a',s'')\sum_{a''\in A}\pi^*(s'',a'')R(s'',a'') \\
&\geq \\
&\sum_{a\in A}\pi(s,a)R(s,a) + \gamma\sum_{a\in A}\pi(s,a)\sum_{s' in S}p(s,a,s')\sum_{a'\in A}\pi(s',a')R(s',a') + \\
&\gamma^2\sum_{a\in A}\pi(s,a)R(s,a) + \gamma\sum_{a\in A}\pi(s,a)\sum_{s' in S}p(s,a,s')\sum_{a'\in A}\pi(s',a')\sum_{s''\in S}p(s',a',s'')\sum_{a''\in A}\pi(s'',a'')R(s'',a'')
\end{aligned}
$$

Dividing by R_{Max} to normalize it

$$
\begin{aligned}
&= \sum_{a\in A}\pi^*(s,a)\frac{R(s,a)}{R_{Max}} + \gamma\sum_{a\in A}\pi^*(s,a)\sum_{s' in S}p(s,a,s')\sum_{a'\in A}\pi^*(s',a')\frac{R(s',a')}{R_{Max}} + \\
&\gamma^2\sum_{a\in A}\pi^*(s,a)\frac{R(s,a)}{R_{Max}} + \gamma\sum_{a\in A}\pi^*(s,a)\sum_{s' in S}p(s,a,s')\sum_{a'\in A}\pi^*(s',a')\sum_{s''\in S}p(s',a',s'')\sum_{a''\in A}\pi^*(s'',a'')\frac{R(s'',a'')}{R_{Max}} \\
&\geq \\
&\sum_{a\in A}\pi(s,a)\frac{R(s,a)}{R_{Max}} + \gamma\sum_{a\in A}\pi(s,a)\sum_{s' in S}p(s,a,s')\sum_{a'\in A}\pi(s',a')\frac{R(s',a')}{R_{Max}} + \\
&\gamma^2\sum_{a\in A}\pi(s,a)\frac{R(s,a)}{R_{Max}} + \gamma\sum_{a\in A}\pi(s,a)\sum_{s' in S}p(s,a,s')\sum_{a'\in A}\pi(s',a')\sum_{s''\in S}p(s',a',s'')\sum_{a''\in A}\pi(s'',a'')\frac{R(s'',a'')}{R_{Max}}
\end{aligned}
$$

$\frac{R(s,a)}{R_{Max}} = R'(s,a)$ This is nothing but the normalized reward

So this indicates that if R_{Max} is positive then the inequality still holds true i.e. normalization has no effect, the optimal policy is still optimal as the value function of it is greater than equal to the value function of all the policies. However, if R_{max} was negative then the sign of inequality will get changed but as we have rewards in numerator too, we cant comment whether the policy would remain optimal or not. There are high chances it might become suboptimal in that case

---

Part Two: Programming (46 Points Total)

Implement the Inverted Pendulum Swing-Up domain. Click here to see an example of a physical implementation of this domain. This problem involves a pendulum with one of its ends attached to a fixed pivot on a wall. The agent can apply torques to the pivot, causing the pendulum to move and changing its angle and angular velocity. The goal is to swing the pendulum from its initial position to an upright position and then keep it balanced. This problem is a common benchmark in reinforcement learning, testing the agent's ability to learn control strategies in a nonlinear and dynamic environment. You will find, below, a complete description of this domain:

- **State**: $s = (\omega, \dot{\omega})$, where $\omega \in [-\pi, \pi]$ is the angle of the pendulum relative to the upright position and $\dot{\omega}$ is pendulum's angular velocity. Angles are in radians and $\omega = 0$ means that the pendulum is upright.
- **Actions**: $a \in [-2, 2]$, where a is a continuous value representing the torque applied to the pivot as the agent attempts to rotate the pendulum clockwise or counterclockwise.
- **Dynamics**: The dynamics are *deterministic*: taking action a in state s always produces the same next state, s'. Thus, $p(s, a, s') \in \{0, 1\}$. To characterize the dynamics, we first need to define the following constants:

$$\begin{aligned} G &= 10 && \texttt{(gravity)} \\ L &= 1 && \texttt{(length of the pendulum)} \\ M &= 1 && \texttt{(mass of the pendulum)} \\ \texttt{MAX\_SPEED} &= 8 && \texttt{(maximum angular velocity)} \\ \texttt{MAX\_TORQUE} &= 2 && \texttt{(maximum torque)} \\ \texttt{dt} &= 0.05 && \texttt{(time step)} \end{aligned}$$

  The next state, $s_{t+1} = (\omega_{t+1}, \dot{\omega}_{t+1})$, can be computed as follows:

$$\begin{aligned} \ddot{\omega}_{t+1} &\leftarrow \frac{3G}{2L}\sin(\omega_t) + \frac{3a_t}{ML^2} \\ \dot{\omega}_{t+1} &\leftarrow \texttt{clip}\big(\dot{\omega}_t + \ddot{\omega}_{t+1}\,\texttt{dt},\ \texttt{-MAX\_SPEED},\ \texttt{MAX\_SPEED}\big) \\ \omega_{t+1} &\leftarrow \omega_t + \dot{\omega}_{t+1}\,\texttt{dt} \end{aligned}$$

  where $\texttt{clip}(x, \texttt{min}, \texttt{max})$ is a function that clips the value x so that it stays in the closed interval $[\texttt{min}, \texttt{max}]$.
- **Terminal States**: Episodes terminate after 200 time steps.
- **Rewards**: Recall that ω_t is the current pendulum angle. Let $\omega_{\text{normalized}} = \Big((\omega_t + \pi)\bmod(2\pi)\Big) - \pi$, where mod is the modulo operator used to get the remainder of a division. Then, $R_t = -\Big((\omega_{\text{normalized}})^2 + 0.1(\dot{\omega}_t)^2 + 0.001(a_t)^2\Big)$.
- **Discount**: $\gamma = 1.0$.
- **Initial State**: $S_0 = (\Omega_0, \dot{\Omega}_0)$, where Ω_0 is an initial angle drawn uniformly at random from the interval $[\frac{-5\,\pi}{6}, \frac{5\,\pi}{6}]$ and $\dot{\Omega}_0$ is an initial angular velocity drawn uniformly at random from the interval $[-1, 1]$.

Next, you will implement a Black-Box Optimization (BBO) algorithm to search for optimal policies to control the Inverted Pendulum system. In particular, you will be implementing the Evolution Strategies algorithm, which is described below. **Notice that you may not use existing RL code for this problem: you must implement the agent and environment entirely on your own and from scratch.**

1.1 Evolution Strategies Method

The *Evolution Strategy* (ES) method for policy search is a simple BBO algorithm that has achieved remarkable performance in tasks such as learning to play video games and controlling simulated robots. Below, we present pseudocode for a simple variant of the ES algorithm originally introduced by Salimans et al. (2017). **We strongly recommend reading *at least* the first part of Section 2 of the paper before implementing the ES method.**

Intuitively, ES performs a "guess and check" process to search for a good policy. It starts with a policy $\theta_0 \in \mathbb{R}^n$ with random parameters as an initial *guess*. It then repeatedly **(1)** tweaks the guess a bit, randomly; and **(2)** moves the current guess slightly towards whatever tweaks worked better. Concretely, at each episode t, the algorithm generates many (say, *nPerturbations*= 100) random-noise perturbation vectors ϵ_i. These are then used to generate a population of perturbed policies, each of which is only slightly different than the current one. The parameters of each perturbed policy are $\theta_t + \sigma\epsilon_i$, where σ is an exploration parameter. The algorithm evaluates each of the perturbed policies by running it in the environment N times, using the function `estimate_J`. This produces the corresponding policy performance evaluations, J_i. The updated policy parameter vector θ_{t+1}, then, is the weighted sum of the *nPerturbations* random-noise vectors, where each weight is proportional to the performance J_i of the corresponding perturbed policy. In other words, perturbed policies with higher performance have a higher weight.

We present pseudocode for ES in Algorithm 1, which uses the `estimate_J` function defined in Algorithm 2. Notice that `estimate_J` is very similar to the function you implemented in Homework 1 to estimate the performance, J, of a given policy by averaging different sampled returns.

Algorithm 1: `Evolution Strategies` (ES) for Policy Search.
Input:
1) Initial policy parameter vector, $\theta_0 \in \mathbb{R}^n$.
2) $nPerturbations \in \mathbb{N}_{>1}$ is the number of perturbations that will be applied to the current policy to generate and identify (possibly better) policies [for example, $nPerturbations = 100$].
3) Exploration parameter, $\sigma \in \mathbb{R}$.
4) Step size α.

Let I be the $n \times n$ identity matrix;
for $t = 0, 1, 2, \ldots$ **do**
  $J_{\text{curr}} = \texttt{estimate\_J}(\theta_t)$;
  for $i = 1$ **to** $nPerturbations$ **do**
    $\epsilon_i \sim \mathcal{N}(0, I)$;
    $J_i = \texttt{estimate\_J}(\theta_t + \sigma\epsilon_i)$;
  Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{\sigma \, \text{nPerturbations}} \sum_{i=1}^{\text{nPerturbations}} (J_i - J_{\text{curr}}) \, \epsilon_i$;

Algorithm 2: `estimate_J` to evaluate the performance, J, of a policy.
Input:
1) Policy parameter vector, $\theta \in \mathbb{R}^n$

Run the parameterized policy using policy parameters θ for $N = 10$ episodes;
Compute the N resulting returns, $G^1, G^2, \ldots, G^N$, where $G^i = \sum_{t=0}^{\infty} \gamma^t R_t^i$;
Return $\frac{1}{N} \sum_{i=1}^{N} G^i$;

1.2 Policy Representation

To solve the Inverted Pendulum Swing-Up problem, we need to select a policy representation that works with continuous state spaces and continuous actions. In order to allow the following experiments to run faster, we will use a *deterministic* policy representation based on neural networks.

Recall that a neural network has a set of n adjustable weights, $\theta \in \mathbb{R}^n$. At each timestep t, the network receives the current state, $s_t = (\omega_t, \dot{\omega}_t)$, as input and outputs a continuous action, a_t, to be executed. Adjusting the network's weights, θ, changes how the state is mapped to an action, thus resulting in a different policy. See Figure 1 for an example of a neural network representing the policy in the Inverted Pendulum domain.

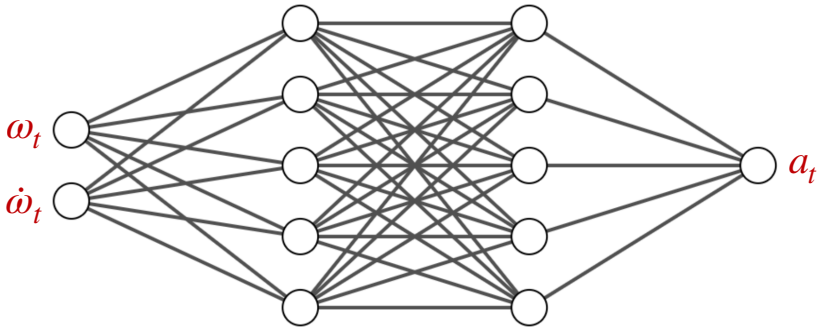

Figure 1: A possible neural network architecture used to represent the policy. It has two layers, each with five neurons.

In this assignment, we provide starter code that implements a neural network. Its API is described in the source code. See the file `main.py` for examples of various available functionalities, such as a function that returns a copy of the network's weights, θ, for use in the ES algorithm, and another that loads a new set of weights into the network. The starter code can be found here.

2. Questions (Programming)

1. (**22 Points**) You will first use the ES algorithm to learn efficient policies in the Inverted Pendulum domain. Doing so requires setting many hyperparameters of the algorithm: *nPerturbations*, σ, α, and the neural network architecture (i.e., the number of layers and the number of neurons in each layer). Search the space of hyperparameters for hyperparameters that work well. Choosing the range from which to sample/select hyperparameters is challenging. As an example, one could heuristically guess that *nPerturbations* is some value in $[50, 150]$; σ may be in $[0.1, 0.5]$; and α is a number from zero to one, possibly in the range $[0.01, 0.05]$. When experimenting with different neural network architectures, focus on simpler designs: networks with at most four layers and between 2 to 16 neurons per layer. Notice that these ranges/intervals and suggestions are just *examples*, and we are *not* suggesting they are necessarily the best ones to use.

   Note: *A near-optimal policy for the Inverted Pendulum domain achieves a return close to* 0 *(or perhaps only slightly below zero). This is because the agent receives a reward of* 0 *whenever the pendulum is balanced and upright. Any reasonable hyperparameter setting should achieve a return of at least* -200 *or better. In all questions, you should aim to get your algorithm's performance as close to the optimal as possible.*

   Report how you searched the hyperparameters and what hyperparameters you found worked best. Show a learning curve plot (i.e., return as a function of the number of iterations/updates performed by ES) for a few selected hyperparameters that you found to be relevant (or surprising) in terms of how they affected your algorithm. Each learning curve plot should present average performance results computed over **5 runs** of the ES algorithm (see *Question 2*, below, for details on how to do this).[1] You should report results for **at least 10 combinations of hyperparameter values** and discuss why you think the algorithm behaved as it did in each case.

   I used a random trial and error method to search for the hyper parameters. I started with the ones mentioned as example in the question and kept guessing it to get optimal results and try converging it. After a lot of trials and errors, i realized that setting a combination of smaller values of sigma and alpha, setting higher values of iterations, setting moderate value of number of perturbations and keeping the neural net relatively simple gave moderate results.

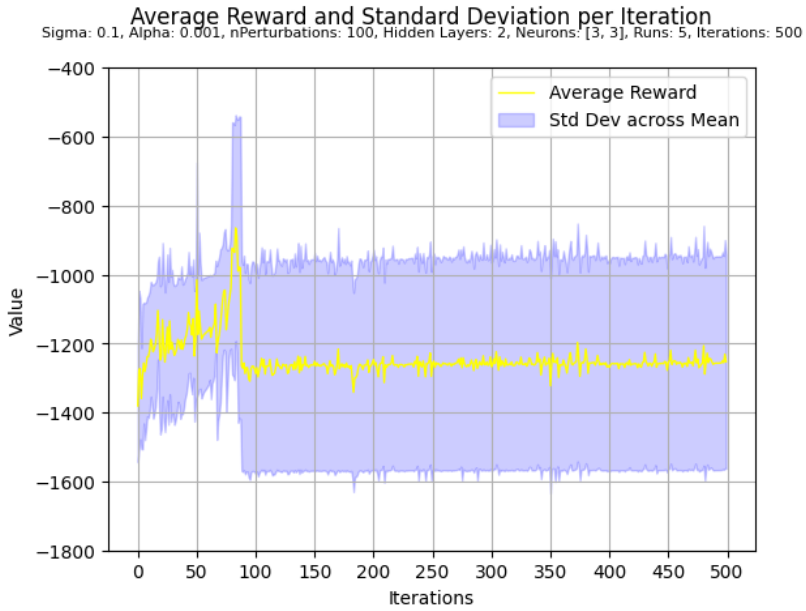

Figure 2

[1] Evaluating hyperparameters over just 5 runs is, in general, certainly not sufficient. You are allowed to do that in this question to lower the computation time required to perform these experiments and analyses. You will evaluate your selected solution/policy more accurately in a subsequent question.

```
Average Reward after 5 run: -1252.5981090851524
Standard Deviation after 5 run: 310.1262878951568
```

Figure 3

In this scenario we see poor average reward, quite high standard deviation deviation and the mean converged around -1100 to -1300. This is possibly because the neural net is too simple, leading to potential underfitting. Its therefore not learning the dynamics effectively.

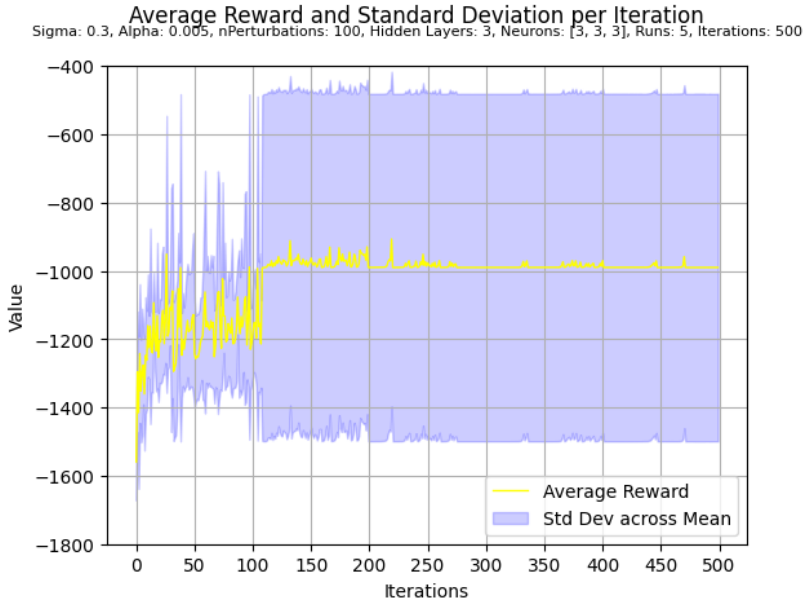

Figure 4

```
Average Reward after 5 run: -990.0463740477093
Standard Deviation after 5 run: 508.99943697430587
```

Figure 5

In this scenario we see poor average reward, still better than the above one, high standard deviation and the mean converges tp -1000 approx. The improvement is due to slighlty making the neural net more complex by adding more hidden layer. However the standard deviation is quite high here. This high standard deviation suggests excessive exploration, learning challenges, and sensitivity to noise in the environment, which can hinder effective learning.

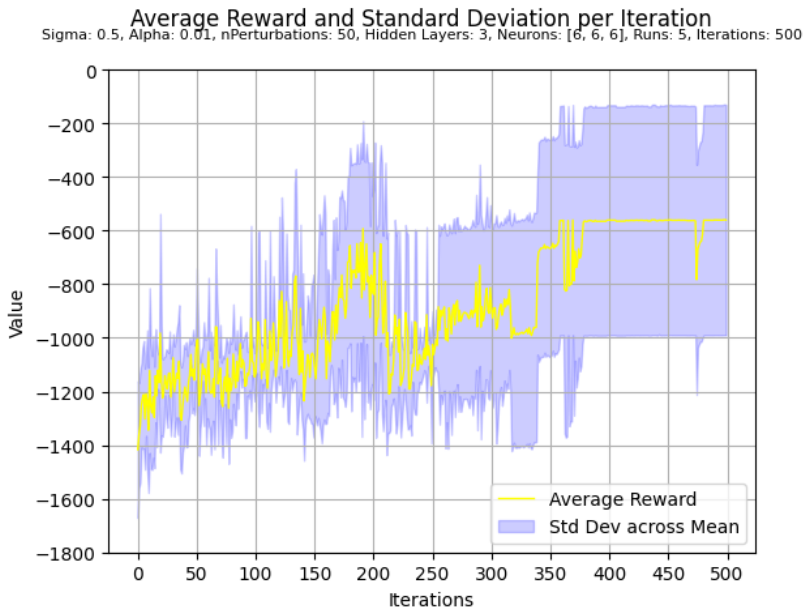

Figure 6

```
Average Reward after 5 run: -560.8858656496911
Standard Deviation after 5 run: 429.4913924531917
```

Figure 7

When we added more neurons while keeping hidden layer same as above, we see improved average reward of -560, however the standard deviation is still quite high. Also, the mean distributing is very erratic with dominant spikes. This is potentially because we made the model complex and also increased the values of sigma which helped increase the weights and noise to our model hence more spread results.

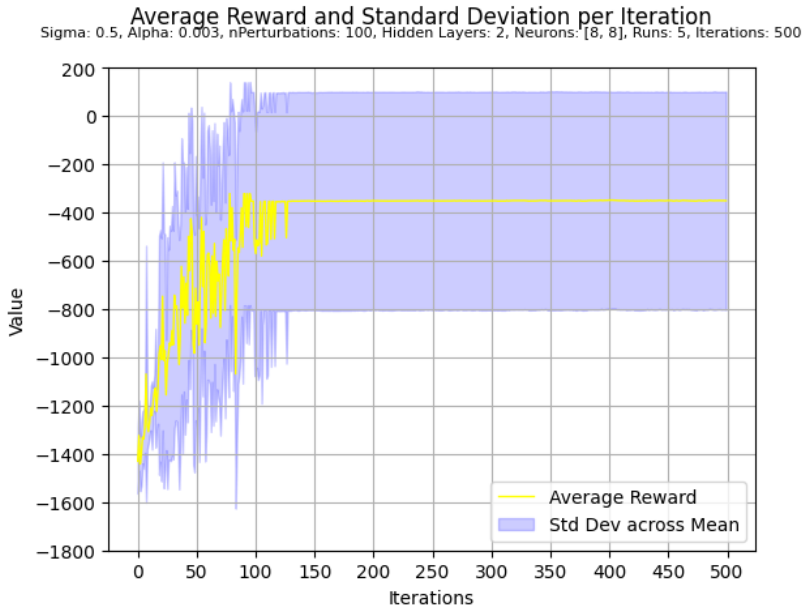

Figure 8

```
Average Reward after 5 run: -351.03057856780396
Standard Deviation after 5 run: 451.6479657921492
```

Figure 9

Here we get better results than all the previous results. Here we see that mean is -351, but still the standard deviation is still high. Here setting alpha to very small value and a moderate sigma helped. We tried making the neural net not too simple and not too moderate. Thats why it was able to learn better than all previous configs. It was able to fairly converge quickly too at almost 150 iterations, indicating that we do not necessarily need to run it for larger iterations.

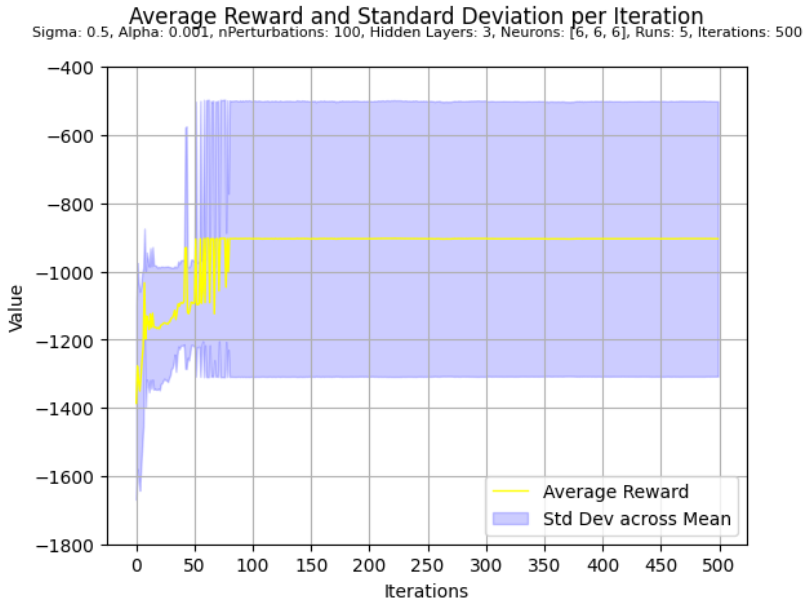

Figure 10

```
Average Reward after 5 run: -904.3112535724653
Standard Deviation after 5 run: 403.38938876063554
```

Figure 11

Here the performance dropped greatly than the previous setting, the mean dipped to -904 and the standard deviation was also high. This was potentially because neural net a little more complex and high sigma added more noise. Analyzing all the graphs at this point made me realize that we shouldnt set the exploration parameter too sigma high-keeping it on a low to moderate range and keeping alpha small along with keeping the neural net with 2-3 hidden layers can help us get good results. Also running to 200 iterations should be enough.

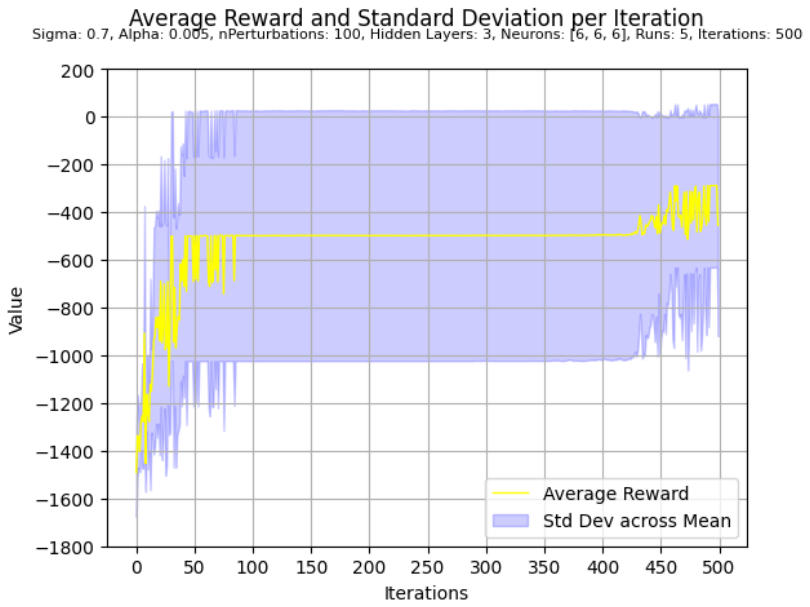

Figure 12

```
Average Reward after 5 run: -455.79468304654466
Standard Deviation after 5 run: 463.3986046202444
```

Figure 13

Here even though sigma is high, but increasing alpha with the same neural net settings as previous helped to get average reward of -455 but still high standar deviation potentially due to high sigma value which adds noise. however, increasing alpha indicated that i shouldnt be setting to less value of alpha so that it escapes the local minima and doesnt remain stuck on the sub optimal policies.

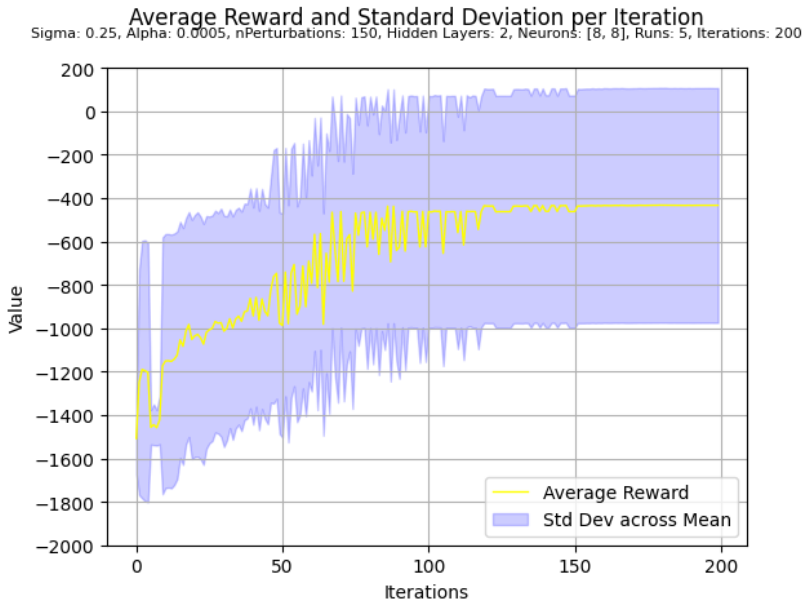

Figure 14

```
Average Reward after 5 run: -433.858167381455
Standard Deviation after 5 run: 540.8124490434205
```

Figure 15

Here we get a mean of -433 which is moderate but the standard deviation is the highest i.e. 540. This is potentially because we increased nPerturbations from 100 to 150 indicating more noise. However, this was able to perform fairly than previous indicating low to moderate sigma and alpha values, and 2-3 hidden layers rangind from 4-8 neurons. Here we see the mean eventially converging after approx 150 runs

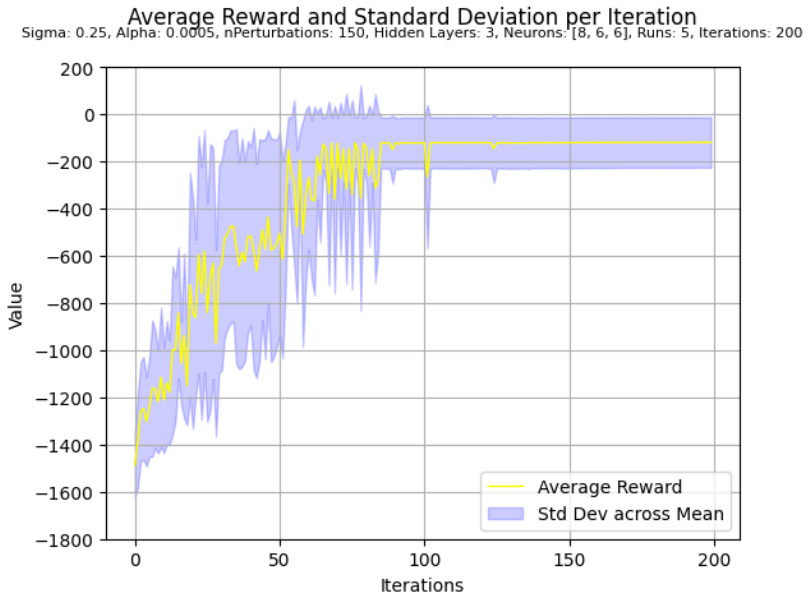

Figure 16

```
Average Reward after 5 run: -119.57069798223895
Standard Deviation after 5 run: 106.49773611850665
```

Figure 17

Here i got the best results with an average reward of -119, the closest reward to zero. And a standard deviation of 106. This suggested was better than previous which suggested that having 3 hidden layers with 6-8 neurons with low to moderate sigma and alpha values gave the best results. Even though the number of perturbations was high, the model was complex enough to get to the optimal policies.

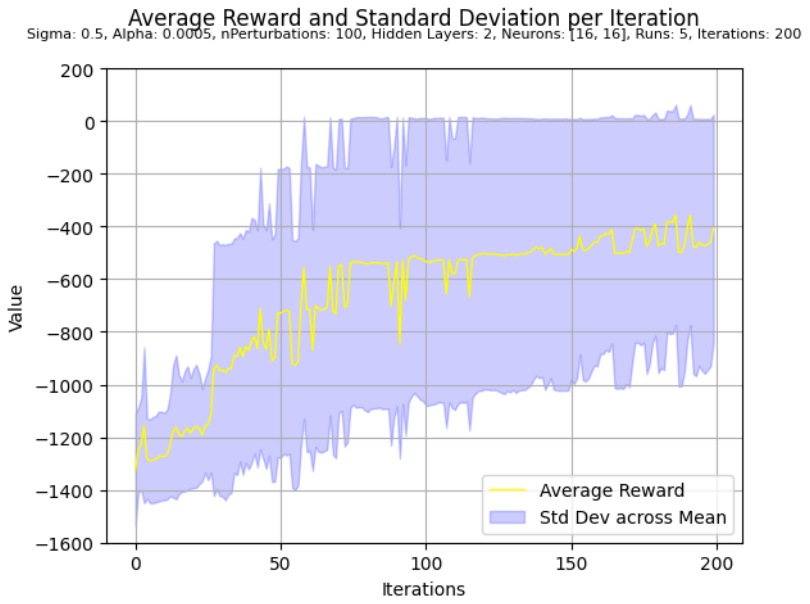

Figure 18

```
Average Reward after 5 run: -406.65779562438615
Standard Deviation after 5 run: 430.7710046755951
```

Figure 19

Here i wanted to try to use 2 hidden layer and add more neurons to see if i was getting comparable results. The resultant mean was -406 indicating adding too many neurons didnt help but rather declined the performance. Also we see that the standar deviation increases. I increased sigma slightly to see its effect and saw to higher sigma, perturbations cause the model to deviate highly from its mean reward.

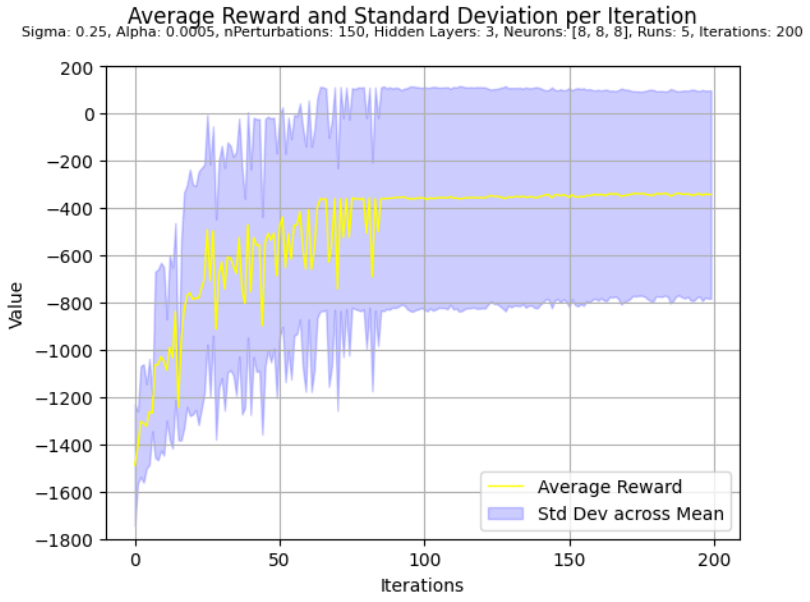

Figure 20

```
Average Reward after 5 run: -343.1982971054835
Standard Deviation after 5 run: 440.39290280037164
```

Figure 21

Here I get -343 mean reward slightly less than the best result potentially because we add more noise here, as nPerturbations is 150 indicating, the model leading to high std deviation as well. This suggested that we should keep the nPerturbations parameter moderate so that it neither overfits nor underfits

2. (**5 Points**) Consider the hyperparameters you identified, in the previous question, as hyparameters that work well. Present their values and show a learning curve plot of your algorithm when using these hyperparameters and when evaluated over **25 runs**.[2] In particular, you will run your algorithm 25 times and plot the mean/average return (computed over the 25 runs) after 1 iteration of ES, after 2 iterations of ES, and so on. *When creating this graph, you should also show the standard deviation of the return for each of these points.*

   The hyperparameters that worked the best and that gave maximum reward were sigma = 0.25, alpha = 0.0005, nPerturbations = 100, hidden layers = 3, neurons in hidden layers = [8, 6, 6], iterations = 200.

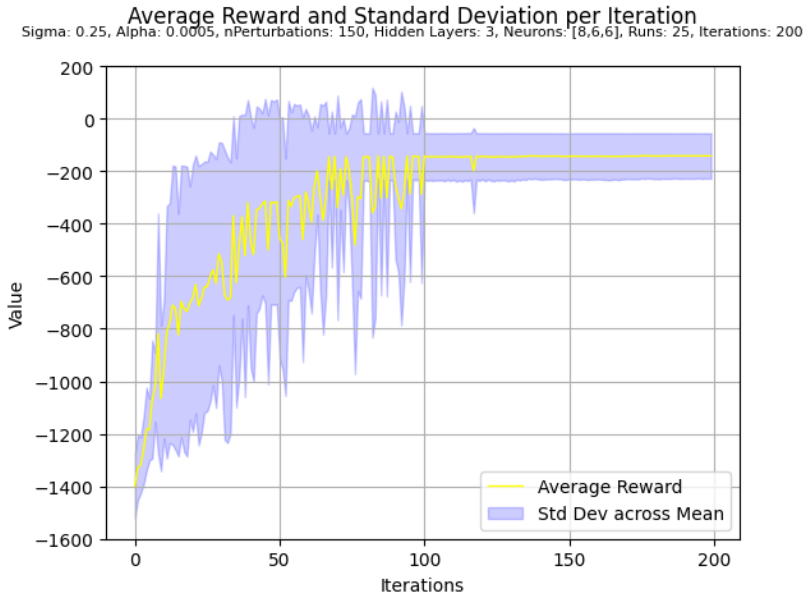

Figure 22

```
Average Reward after 25 run: -141.957442041462
Standard Deviation after 25 run: 86.41030232637438
```

Figure 23

3. (**7 Points**) Did your algorithm reach near-optimal performance/return? If not, why do you think that was the case? In general, reflect on this problem: was it easier or harder than you expected to get the method working? What were the main challenges?

   The algorithm did not reach zero reward, however, we got close to zero reward i.e. -141 for 25 runs as seen in question 2. We were able to get near to optimal results but it took alot of tries. In general, this was quite hard to get the expected method working. This is because of various reasons: as the equation of calculating the reward is complicated, it wasn't very intuitive to guess the hyper parameters and infer relationships between them to get optimal results. Some challenges involved, not knowing ways to find effective hyperparameters to get

[2] In real-life applications and research, results need to be averaged over many more runs. Here, using 25 runs is acceptable because the MDP has little stochasticity, the policy is deterministic, and ES's performance in this domain is not highly sensitive to its initial conditions.

good results, another major issue was the amount of processing time and computational resources. Even though I used numbas library and multiprocessing, it took alot of time to try out different combinations. Another problem was that I could not try more iterations than I already tried because of hardware limitations and computational resources limitation in general.

4. (**12 Points**) The type of analysis described above, where we check how different hyperparameters affect learning curves, can be used to fine-tune your algorithm to optimize different performance criteria. You could, for example, be interested in finding hyperparameters that *(a)* make the algorithm's average performance more stable over time (i.e., that prevent the mean return from fluctuating wildly as a function of the number of iterations or updates performed by ES); *(b)* allow the algorithm to find better policies, though possibly causing its runtime to increase significantly; or *(c)* cause the algorithm to find policies with a reasonable average performance and low variance.

   Discuss your findings on how different types of hyperparameters affect each of these three performance criteria. For example, are there sets of hyperparameters that lead the algorithm to converge to the same mean return, but where one set causes returns to fluctuate widely over time, while another results in less fluctuation? Are there hyperparameters that make the algorithm converge very slowly, yet consistently find high-performance policies? Do some hyperparameters allow the algorithm to find policies that perform well and have low-variance in their average return? Can you identify any patterns in how different hyperparameter values impact these performance criteria?

   (a) Making mean stable over time
   The stability can be affected by hyperparameters like the step size (alpha), noise scaling (sigma), and the number of perturbations.
   Step Size (alpha): If the step size is too large, the updates in weights may result in erratic behavior, leading to wide fluctuations in the mean return. A smaller alpha can help prevent these wild fluctuations, though it may slow down convergence. In my case, I got best result when alpha was small (0.0005 Fig 16). The mean even became constant after almost 100 iterations. However, when alpha was large such as alpha = 0.01 (Fig 6), I didnt that results. Iterations: I was however able to converge most of my results by running for iterations between 200-500. After 100-150 (Fig 10,12,14,16) iterations, my mean was able to converge.
   Noise Scaling (sigma): Large values of sigma increase the exploration during the policy search by adding more randomness to the updates, which can cause the returns to fluctuate more. In my case when sigma was high (0.7 Fig 12) in the middle of the iterations, it caused the mean to become stable however towards the end, it became erratic again leading to inconsistent results. A smaller sigma reduces exploration and can stabilize the learning curve but may also lead to premature convergence to sub-optimal policies. In most of my cases (e.g. sigma = 0.1, 0.3 Fig 2, 4), this happened, sigma was low however along with other parameters based on their values, it lead to less reward indicating converging to sub-optimal policies.
   Number of Perturbations (nPerturbations): More perturbations typically lead to a better estimation of gradients, stabilizing the returns over time. I was able to get good results with 100 perturbations (Fig 16, 18, 20) and was able to see mean converge eventually. A lower number may cause greater variability in returns such as in my case when it was set to 50 (Fig 6), i got extremely random results and didnt converge after almost 300 runs.

   (b) Finding better policies with increased runtime
   I was not able to find better policies, but i was able to converge the policies after increasing the runtime. Say when we ran the code for 200-500 iterations, the mean was able to converge. However, not always they resulted in the greatest of rewards. Sometimes for 200 iterations i got a reward of -144 and the other times i was not. Increasing the run time made the model more stable however.
   Smaller alpha values often result in slower, more controlled updates to the policy. This slower progression can allow the algorithm to eventually converge to a better policy, even though it might take more iterations. In my case i got best results when alpha was 0.0005 (Fig 16,12,20), it took at lot of time to run, approx 30 mins, but gave me the best result, other times when alpha was relatively large say 0.01 (Fig 6), though it ran very quickly but i didnt get optimal rewards

   (c) Finding policies with low variance and reasonable performance
   I found that when alpha and sigma are both relatively low, i got stable policies with low variance than others. The best hyperparams gave me a standard deviation of 106 for 5 runs (Fig 16) and 86 (Fig 22) for 25 runs. This indicated that if i ran my code for more number of runs, i was getting improved results in terms of both mean and standar deviation. But for the same alpha and sigma for which i got optimal result, I found that on chaging the number of hidden layers and neurons the model performed very errtically and gave me a very high standard deviation of 540 (Fig 14).

A general observation i found through my runs is that setting smaller to moderate values of sigma and alpha, setting higher values of iterations, setting moderate value of number of perturbations and keeping the neural net relatively simple with 2-3 layers with 4-8 neurons gave decent results. Over complicating them or setting high values of those, resulted in poor performance.