Fundamentals of Machine Learning

Aadim Nepal
an3854@nyu.edu

Spring 2024

Abstract

This document contains the report for the reinforcement learning project. My document can be roughly divided into two parts: implementation of the reinforce algorithm along with tuning the algorithm using actor-critic method for discrete action space and continous action space for the cartpole problem. I will mostly focus on two algorithms: firstly reinforce and secondly actor-critic method and compare these two for each instances of discrete as well as continous action spaces. This document also contains the challenges associated with continous action spaces and proposes potential ways to deal with it.

Contents

1 The Cartpole Environment

1.1 General Overview

The Cartpole Problem is a classic problem in the field of reinforcement learning. In this problem, you have a pole that is attached by an un-actuated joint to a cart, which moves along a frictionless track. The goal is to balance the pole upright by applying left or right forces to the cart. In our project, the environment for the cartpole is implemented in `"./src/envs/cartpole_v0.py"`. At the beginning of the project, the state space is continous and the action space is discrete taking in two actions, move right or move left denoted by +1 and -1 respectively

1.2 Objective

The goal of reinforcement learning for the cartpole problem is to learn to balance the pole by keeping it upright and preventing it from falling. Intuitively, this can be accomplished by moving the pole in the direction which its falling and the trick for the reinforcement agent is to learn the optimal movement in that direction that keeps the pole upright. The agent must learn this from trial and error, and each time it succeeds in balancing the pole, a rewards is earned and if it fails, it does not earn any reward. So it learns to balance by maximizing the long term accumulated reward of taking a certain action in the current state.

1.3 States

The Cartpole State consists of four variables:

1. The position of the cart along the track
2. The velocity of the cart
3. The angle of the pole with the vertical axis
4. The angular velocity of the pole

1.4 Actions

For the initial part of the project, there are two actions you can take

1. Apply force towards the left denoted by -1
2. Apply force towards the right denoted by 1

The force values can be thought to be in Newton but this is not important as the cartpole environment handles the equations well.

1.5 Rewards

The agent receives a reward of 1 everytime it successfully balances the pole and gets penalized if it does not balance the pole. The program terminates as soon as the pole falls down and a new iteration is started. The agent takes an action by being in a specific state and the long term reward of being in that state and taking that action is estimated using Monte Carlo method. Via trial and error, the agent learns to maximize the long term reward and is expected to learn the movements required to balance the pole.

1.6 Markov Decision Process

The Cartpole Problem is a MDP. What this means is that it can be represented by the tuple (S, A, P, R, γ) where S represents the State Space, A is the Action Space, P is the transition kernel, R is the immediate reward and γ is the discount factor. The probability of transitioning into next state only depends on the current state and is independent of past states - a fundamental assumption of a system that is considered a Markov Decision Process. In the context of the cartpole problem, the tuples denote the following:

1. S: This represents the entire combination of 4-dimensional space vector i.e. cart position, velocity, pole angle, and pole angular velocity.

2. A: Action space denotes two actions, either left force or right force.

3. P: Transition kernel that denotes the probability of moving into another state given the current state and the action taken after being in the current state. It is essential the probability distribution over state space.

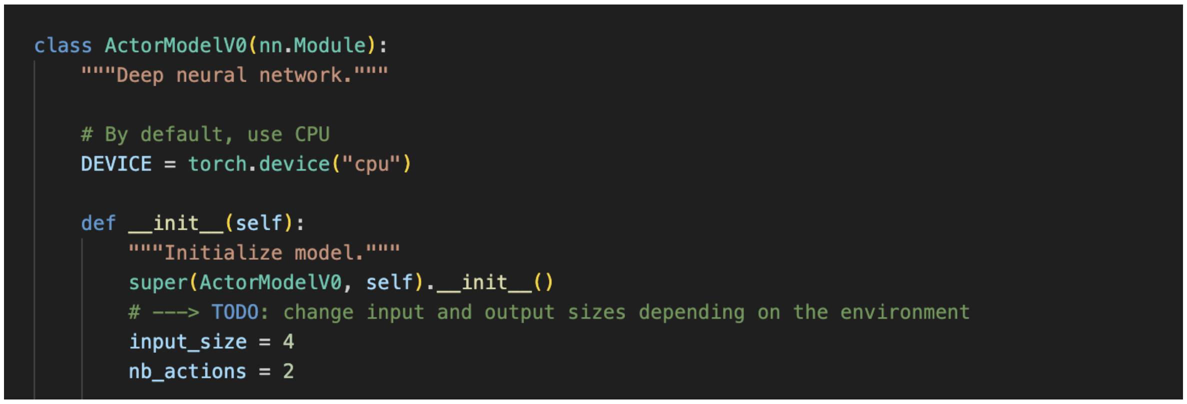
4. R: Denotes the immediate reward for being in the current state. In the context of the cartpole problem, it is 1 if the agent balances the pole. The transition PDF is determined by the physics of the cartpole problem which takes into gravity and other variables.

5. γ: Specifies how much to prioritize future reward. The value is always in the range 0 to 1 and it's important because it keeps the long-term reward bounded and convergent. Also, the Bellman operator is a contraction mapping because of the discount factor.

2 Policy Neural Network

2.1 actor_v0.py

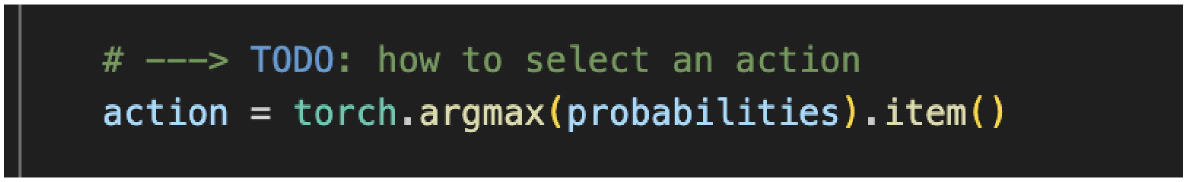
The first problem in the assignment consists of the problem of specifying the values for *nb_actions* and *input_size*. In the neural network, it is evident that *nb_actions* is the dimension of the action space and *input_size* is the dimension of the state space. In the cartpole problem, the dimension of the action space is 2 and state space is 4 respectively.



```
class ActorModelV0(nn.Module):
    """Deep neural network."""

    # By default, use CPU
    DEVICE = torch.device("cpu")

    def __init__(self):
        """Initialize model."""
        super(ActorModelV0, self).__init__()
        # ---> TODO: change input and output sizes depending on the environment
        input_size = 4
        nb_actions = 2
```

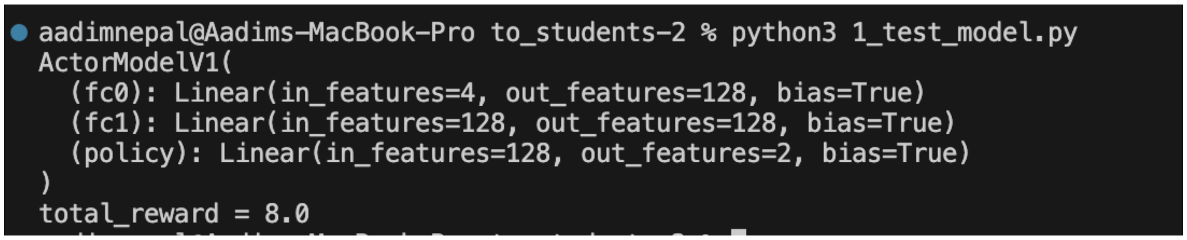
2.2 1_test_model.py

We will select the action with the maximum probability. In order to do this, we use the torch functionality *torch.argmax()*



```
# ---> TODO: how to select an action
action = torch.argmax(probabilities).item()
```

If we do this and run the code, the model performance is very poor because it is using a random policy. So the cart movement is pretty random and the animation in the pygame instantly closes the moment it starts because the pole immediately falls down. The random policy rendered a reward of only 8.0 as you can see in the figure below:



```
● aadimnepal@Aadims-MacBook-Pro to_students-2 % python3 1_test_model.py
  ActorModelV1(
    (fc0): Linear(in_features=4, out_features=128, bias=True)
    (fc1): Linear(in_features=128, out_features=128, bias=True)
    (policy): Linear(in_features=128, out_features=2, bias=True)
  )
  total_reward = 8.0
```

In order to improve the performance, we need the train the model to learn an optimal policy and we will see below that an optimal policy is one that maximizes the long term reward. If I am to be mathematically precise, optimal policy is the policy that maximizes the optimal value function denoted by the following equations:

$$\pi^*_\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V^*_\pi(s')] \tag{1}$$

$$V^*_\pi(s) = \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V^*_\pi(s')] \tag{2}$$

In a reinforcement learning context, s and s' represent the current and next states, respectively. The action a taken in state s leads to a transition to state s' with an associated reward r. The discount factor γ quantifies the importance of future rewards. The value function under policy π, denoted $V^*_\pi(s')$, evaluates the next state s'. The probability of transitioning from state s to state s' and receiving reward r given action a is represented by $p(s', r \mid s, a)$.

We will now see how we can adapt this into REINFORCE algorithm

3 REINFORCE: 2_reinforce.py

3.1 Discounted Sum of Rewards

The Discounted Sum of Rewards can be represented by

$$\sum_{t=0}^{\infty} \gamma^t R_t$$

where γ represents the discount factor and R_t represents the immediate reward of being in state t. I implemented exactly this in the code. Feel free to double check it but after accessing the rewards, I started backwards form the last time step and cumulatively computer rewards for the previous time steps.

3.2 Reinforce Loss

We want to find the parameters θ that maximize the performance measure $J(\theta) = \mathbb{E}_{\pi_\theta}[G_0]$ where $G_t = \sum_{k=0}^{\infty} \beta^k r_{t+k+1}$ and $\beta \in [0, 1]$ being a discount factor. To do so, we use the gradient ascent method: $\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta_k)$ with α being the learning rate.

The performance measure depends on both the action selection and the distribution of states. Both are affected by the policy parameters, which make the computation of the gradient challenging.

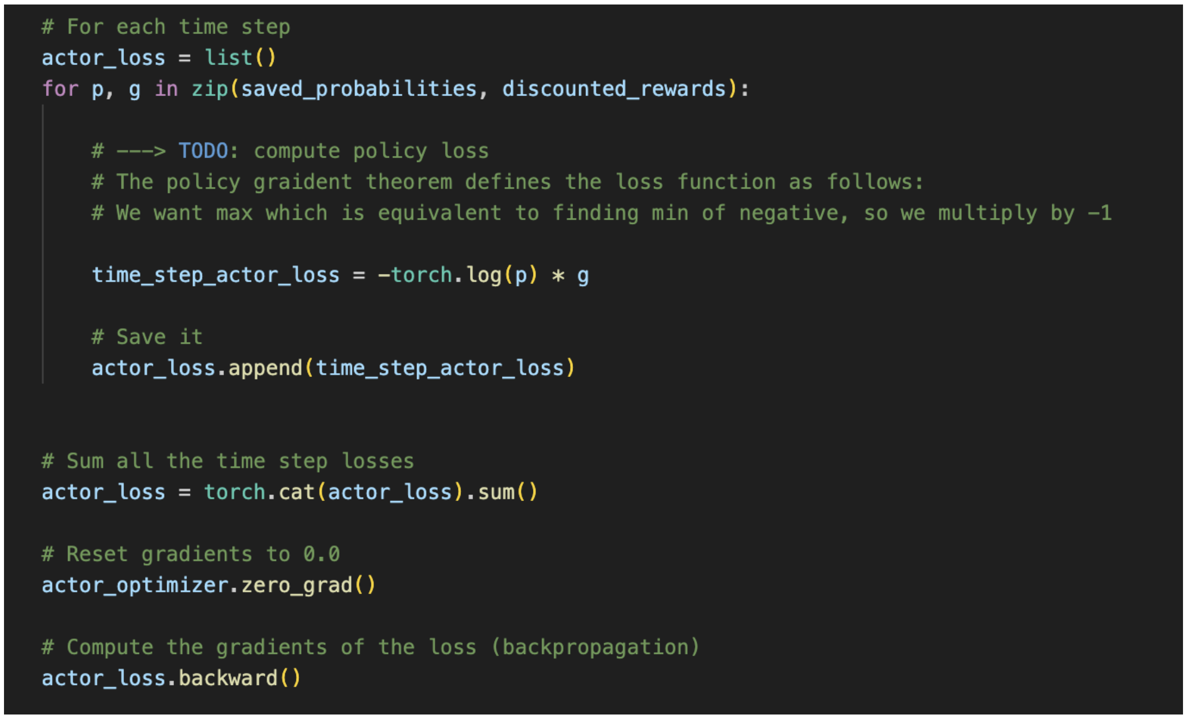
The policy gradient theorem gives an expression for $\nabla_\theta J(\theta)$ that does not involve the derivative of the state distribution. The expectation is over all possible state-action trajectories over the policy π_θ:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} G_t \nabla \log \pi_\theta(a_t|s_t)\right].$$

In the REINFORCE algorithm, we use a Monte-Carlo estimate over one episode, i.e., one trajectory:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{\infty} G_t \nabla \log \pi_\theta(a_t|s_t).$$

By following these equations, we write the code as shown below:



```
# For each time step
actor_loss = list()
for p, g in zip(saved_probabilities, discounted_rewards):

    # ---> TODO: compute policy loss
    # The policy graident theorem defines the loss function as follows:
    # We want max which is equivalent to finding min of negative, so we multiply by -1

    time_step_actor_loss = -torch.log(p) * g

    # Save it
    actor_loss.append(time_step_actor_loss)


# Sum all the time step losses
actor_loss = torch.cat(actor_loss).sum()

# Reset gradients to 0.0
actor_optimizer.zero_grad()

# Compute the gradients of the loss (backpropagation)
actor_loss.backward()
```

3.3 Learning Rate

The Learning rate is an important hyperparameter in the context of reinforcement learning. The default value for learning rate initially was 0.1, and discount factor 0.1. When I tried running with these values, the model did not give proper reward and showed no signs of convergence, it kept fluctuating between rewards 8 to 10. The theory behind the choice of learning rate is this: you want your model to find and skip local minimas optimally, something that the Adam optimizer helps in accomplishing as well. Using a high value for the learning rate might imply faster convergence but this might also mean that the model might overshoot and skip important saddle points. Small learning might take forever to converge and make the model stuck in unimportant local minimas. We want this to be optimal. So we choose this value optimally. In my setting, I found that the best choice was 0.001 because it achieved stable convergence as opposed to 0.1 and 0.01

3.4 Discount Factor

Discount factor is important for two reasons:

1. It makes the long term sum of accumulated rewards bounded and convergent
2. The bellman operator is a contraction due to the bounds of the discount factor

A higher value of discount factor (1) means that you are paying high attention to future rewards and a lower value (0) means that you are paying low attention to future rewards. Again, we want to find an optimal tradeoff for discounting future reward and find a strategy that works the best for our model in order to obtain convergence. 0.1 was not at all helpful, the algorithm did not converge, I kept increasing the discount factor until I reached 0.99 and found that the model converged when we valued the future reward the most. So, 0.99 is the optimal.

3.5 Convergence Metric

The convergence metric is this: if the agent obtains a reward of 500 successively, then the algorithm has converged. We can interpret this as: if there are any instances in the iteration step where the reward is 500 respectively for 10 consecutive iterations, then we conclude the algorithm has converged.

4 Markov Property

4.1 Acceleration removed, does it converge?

The algorithm does not converge even when it reaches an iteration of 10000. The rewards mostly fluctuate around 30-40. The reason this happens is because when you remove the the cart velocity and pole angular velocity from the cartpole environment, the process is no longer a markov decision process. Certainly, by excluding the cart velocity and pole angular velocity from the state, the problem's Markov property has been violated. This property asserts that the future system dynamics should be solely determined by the current state, independent of previous states. Without the velocities in the state, the policy lacks full insight into the system's dynamics, making it impossible to precisely forecast the future state using only the present state information (cart position and pole angle).

4.2 Modifications, does it converge?

I made the following addition. I added an information for previous state: I modified the step to include the position and angular velocity of the previous state. This way, I made the chain markovian again. The current state has information of the previous state (i.e. position and angular velocity of the previous state is now known). We will change back the nb actions to 4 again. In short, we modified the state to contain the information for previous state instead of its current state. The chain is now markovian. The algorithm now converges successfully. This highlights the importance of MDP in RL setting.

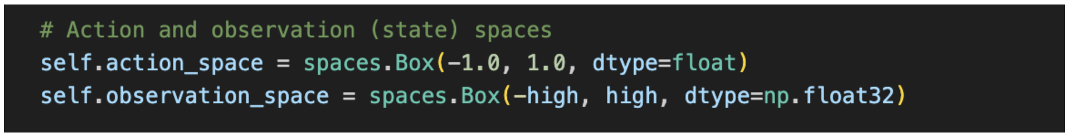
5 Open Questions

5.1 Overview of Continuous Action Spaces

For the Open Question, I chose to work on the continuous action space for the cartpole environment itself. The dimension of action space in our previous setting was 2, we could only apply force in two directions. For this setting, the dimension of action space is infinite which means I can apply force of any values in the range from positive 1 to negative 1. I first begin by modifying the cartpolev0 environment.

5.2 Making Cartpole Action Continuous

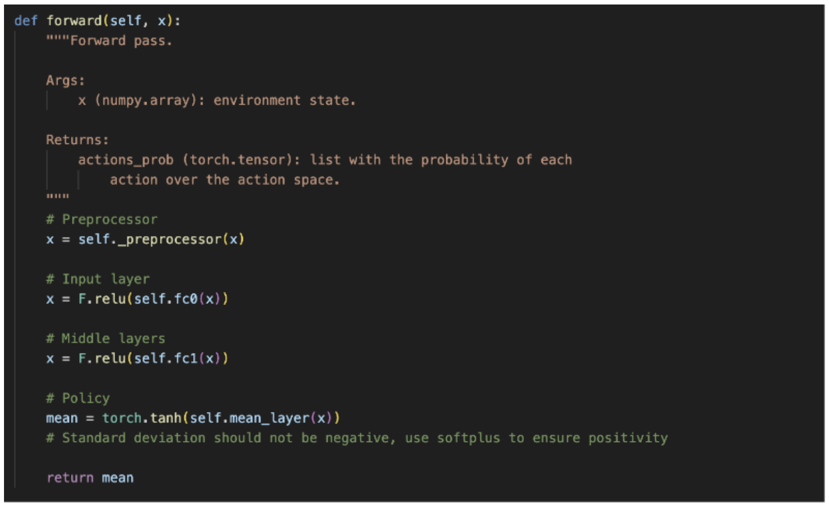
The CartPoleV0 environment had defined the action space to contain only two values -1, and 1. I changed this code and included a Box to define the action space from a range of -1 to 1 continuously. Here's the code that I modified:



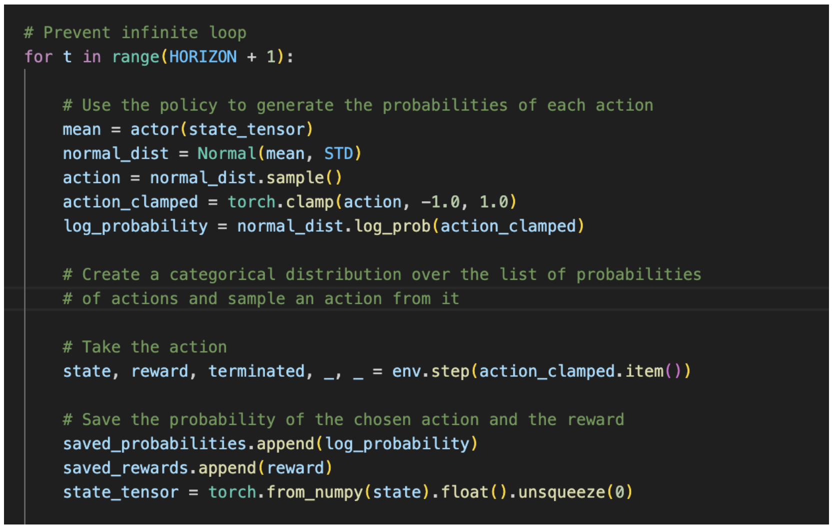
```
# Action and observation (state) spaces
self.action_space = spaces.Box(-1.0, 1.0, dtype=float)
self.observation_space = spaces.Box(-high, high, dtype=np.float32)
```

5.3 ActorV0 to output continuous actions

For discrete setting, ActorV0 defined a neural net that produced probability outputs for the two actions. My first instinct was to modify the neural net to ouput mean and standard deviation for action. This way the output would be a continuous value and the log probability could be calculated assuming that the action is distributed gaussian. But there were several issues, first the output of standard deviation was very weird and many times it led to high model complexity and NAN values (gradient explosion). I emailed professor Elie and he suggested me to keep standard deviation fixed, so its a hyperparameter. I set it to 0.1 throughout the project because I felt like this was the optimal value that reduced the divergence of the distribution while keeping the action output continuous. Hence, my actorV0 neural net contains only 1 output, assumed to be the mean continuous action output that is distributed gaussian with the standard deviation of 0.1 Note: You can see below that I used a Tan hyperbolic activation function at the end. This is because the range of tan hyperbolic is -1 to 1 which exactly defines the new continuous action space that we are employing in our setting.


```python
def forward(self, x):
    """Forward pass.

    Args:
        x (numpy.array): environment state.

    Returns:
        actions_prob (torch.tensor): list with the probability of each
            action over the action space.
    """
    # Preprocessor
    x = self._preprocessor(x)

    # Input layer
    x = F.relu(self.fc0(x))

    # Middle layers
    x = F.relu(self.fc1(x))

    # Policy
    mean = torch.tanh(self.mean_layer(x))
    # Standard deviation should not be negative, use softplus to ensure positivity

    return mean
```

5.4 REINFORCE for continuous action space


```python
# Prevent infinite loop
for t in range(HORIZON + 1):

    # Use the policy to generate the probabilities of each action
    mean = actor(state_tensor)
    normal_dist = Normal(mean, STD)
    action = normal_dist.sample()
    action_clamped = torch.clamp(action, -1.0, 1.0)
    log_probability = normal_dist.log_prob(action_clamped)

    # Create a categorical distribution over the list of probabilities
    # of actions and sample an action from it

    # Take the action
    state, reward, terminated, _, _ = env.step(action_clamped.item())

    # Save the probability of the chosen action and the reward
    saved_probabilities.append(log_probability)
    saved_rewards.append(reward)
    state_tensor = torch.from_numpy(state).float().unsqueeze(0)
```

I copied and pasted the REINFORCE algorithm for discrete case and made two significant changes to accomodate it for continuous action space:

1. Since my actor model outputs the mean value for the action, I extracted this information and defined a Normal distribution with this mean and standard deviation of 0.1

2. Then, I sampled a value from this gaussian. This value represents the action to be taken, and I computed the probability by computing the log probability of this sampled value from the defined gaussian distribution

3. Before computing the log probability, I made sure that the sampled action is in the range for the action space defined to be from -1 to 1. I utilized torch.clamp

to clamp the value from -1 to 1. It is very possible that the sampled value might be out of this range. One example is when the mean is 1. You can sample 1.1 with good probability and we dont want that.

The rest of the code is similar to the discrete setting. These are the only modifications required for continuous action space.

5.4.1 First set of Results

The results are extremely weird. When I tried to run this code, I used learning rate to 0.001 and discount factor to 0.99. It took around 60000 iterations but it converged successfully. I was happy, extremely elated but then I tried running it again but it did not converge. In short I faced two central problems:

1. Inconsistency with convergence, the algorithm rarely converged, and even when it converged, it was very inconsistent and took a significantly long time. With normal 500 reward, and convergence setting like in the discrete case, I tried to run this code for around 6 times and I was able to converge it just once. Other times, it was close but never actually converged

2. Gradient Explosion, and the presence of NaN. This was a big problem. This highlights the reinforce code is very sensitive to continuous setting and since there are so many iterations, implying many iterations of multiplicative gradients, the neural net just overshoot and led to gradient explosion.

5.4.2 Modifications

Hoping to improve the results, I tried making two changes:

1. I decreased the learning rate and decreased the convergence threshold. I made the inner loop run for 400 iterations and defined convergence as: if there are 10 respective successive iterations where the reward is larger than 400, then the algorithm should converge. I just relaxed the convergence threshold.

2. I added gradient clipping to handle gradient explosion.

5.4.3 Second set of Results

The results are as follows:

1. The algorithm converged with relaxed convergence setting. It was still inconsistent but not as inconsistent as when the convergence threshold was as tight as discrete setting. When I decreased the threshold to 10 successive rewards of 300, the algorithm seemed to converge frequently.

2. I ran the test model and the test model generally produced an output of 500 for the reward but was incosistent in that it sometimes produced an output of 200 to 300. This suggests that the current REINFORCE algorithm is very ill-suited for continuous action setting.

3. 0.001 was the best learning rate, but 0.0001 was worked well as well, timing was the only factor.

5.4.4 Next steps

I was fed up. I spent an entire day thinking that I would go over google's deepmind paper on Deep Deterministic Policy Gradient (DDPG) and implement it because I assumed this was the only way out. Even though I dont discuss the results above, I had tested actor-critic model in discrete setting on my own and found remarkable results so I decided to expand the actor-critic model to continuous setting and see how it performs. Motivated, I began.

5.5 Actor-Critic for Continuous Setting

5.5.1 Introduction

The Actor-Critic method is a staple approach in reinforcement learning, merging policy-based (actor) and value-based (critic) techniques to efficiently learn optimal policies. Herein, we define the structure and equations foundational to the Actor-Critic strategy.

5.5.2 Actor

The **actor** selects actions based on the current policy, parameterized by θ, with the policy function denoted by $\pi_\theta(a|s)$ indicating the probability of taking action a in state s.

5.5.3 Critic

The **critic** evaluates actions taken by the actor by computing a value function, which can be either a state-value $V^\pi(s)$ or an action-value $Q^\pi(s, a)$ function. These functions estimate the expected return from states or state-action pairs, respectively.

5.5.4 Critic Update

The critic updates its value function using temporal-difference (TD) learning:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

$$V(s_t) \leftarrow V(s_t) + \alpha_c \delta_t$$

where δ_t is the TD error, γ is the discount factor, r_t is the reward received, and α_c is the learning rate for the critic.

5.5.5 Actor Update

The actor updates its policy parameters guided by the critic's TD error:

$$\theta \leftarrow \theta + \alpha_a \delta_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

where α_a is the actor's learning rate.

5.5.6 Objective Function

The objective function in Actor-Critic methods aims to maximize the expected return from each state under the current policy. The policy's performance gradient is expressed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\delta_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

This equation takes the expectation over the trajectory distribution generated by π_θ, with the policy gradient scaled by the TD error δ_t.

5.6 Algorithm from Sultan and Andrew

Algorithm 1 One-step Actor-Critic (episodic)

```
 1: Input: a differentiable policy parameterization $\pi(a|s,\theta)$
 2: Input: a differentiable state-value parameterization $\hat{v}(s,w)$
 3: Parameters: step sizes $\alpha^\theta > 0, \alpha^w > 0$
 4: Initialize policy parameter $\theta \in \mathbb{R}^d$ and state-value weights $w \in \mathbb{R}^d$
 5: repeat
 6:     Initialize $S$ (first state of episode)
 7:     $I \leftarrow 1$
 8:     while $S$ is not terminal do
 9:         $A \sim \pi(\cdot|S,\theta)$
10:         Take action $A$, observe $S'$, $R$
11:         $\delta \leftarrow R + \gamma\hat{v}(S',w) - \hat{v}(S,w)$        ▷ if $S'$ is terminal, then $\hat{v}(S',w) = 0$
12:         $w \leftarrow w + \alpha^w \delta \nabla_w \hat{v}(S,w)$
13:         $\theta \leftarrow \theta + \alpha^\theta I \delta \nabla_\theta \log \pi(A|S,\theta)$
14:         $I \leftarrow \gamma I$
15:         $S \leftarrow S'$
16:     end while
17: until forever
```

I followed these steps and some tutorials from online and github notebook to produce the code below: The following Python script demonstrates the setup and execution of an Actor-Critic algorithm to train a policy to balance a pole on a cart. The implementation includes model definitions, training loops, and logging of performance metrics.
I want to mention that I adapted this for continuous setting similar to the way I adapted

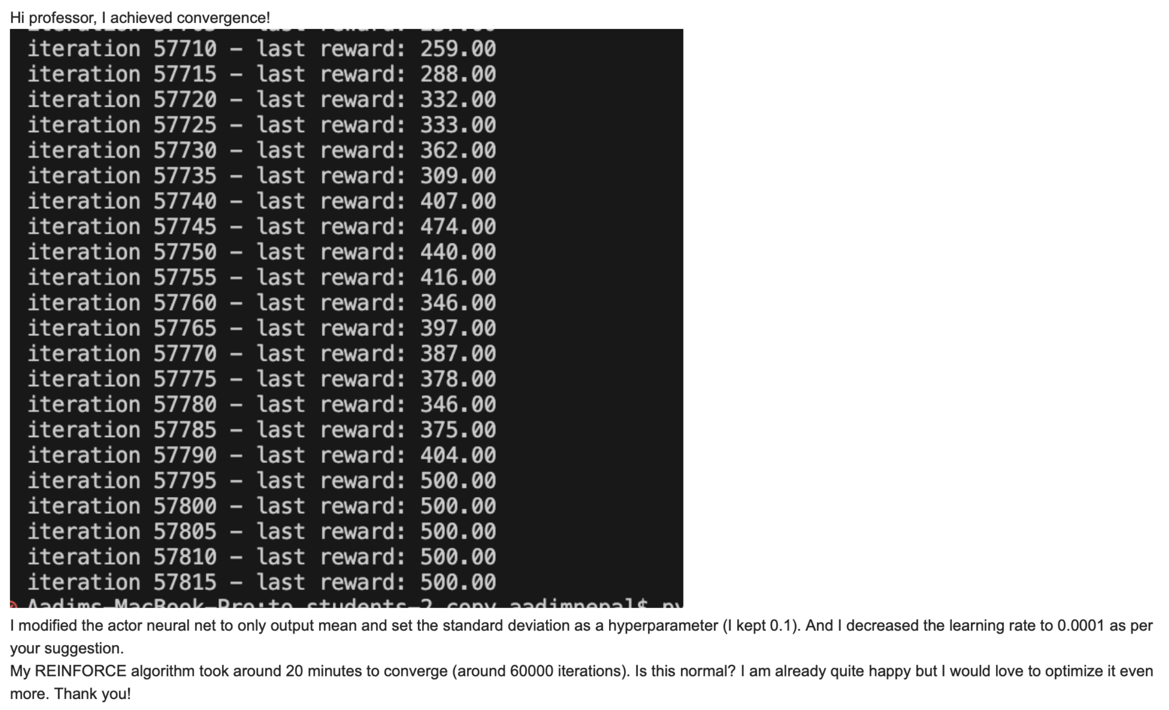
the REINFORCE code for continuous setting. I made a Critic Network, I didnt include code for this because I felt this was really simple, a typical neural net with one output assuming it outputs the value for that state. I extracted mean from the actor, defined a gaussian distribution, sampled an action, clamped it and computed the log probability of the clamped action. The action to be taken is the clamped action and the probability to be used is the log probability. Computation of delta, policy loss, value loss is intuitive from the algorithm from Sultan's book.

5.6.1 Results

The results are amazing. The algorithm always converges, averaging an iteration of about 6000. Expanding the actor critic algorithm to continuous setting is the optimal solution for continuous action space cartpole problem. Unlike the continuous version of the REINFORCE algorithm, the rewards are consistent and does not fluctuate from a very high value to low value. I kept discount factor to 0.99 and learning rate to 0.0001 because I found this value of learning rate to be the one which I could trust more because it always converged in around 6000 iteration. I ran the test model and the rewards were always 500. And just to note, the model converged in the tight convergence setting of discrete setting.

5.7 Miscallaneous Pictures

5.7.1 When Reinforce converged by mistake on continuous setting!

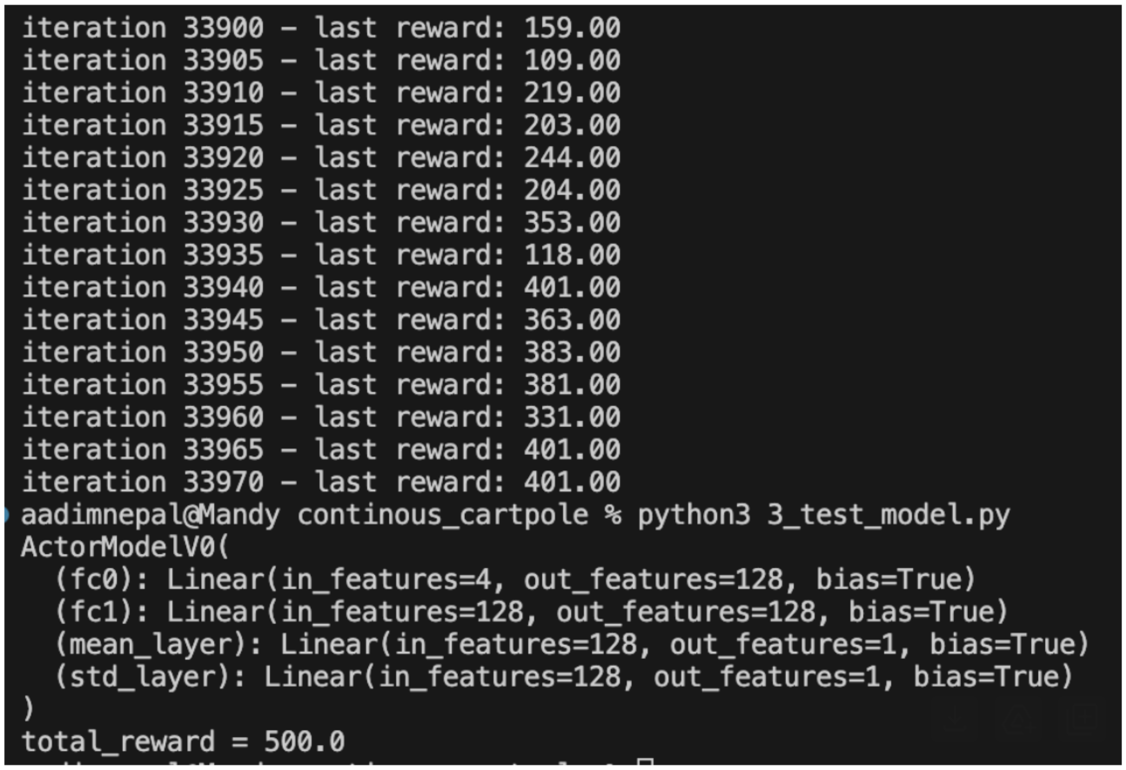


Hi professor, I achieved convergence!

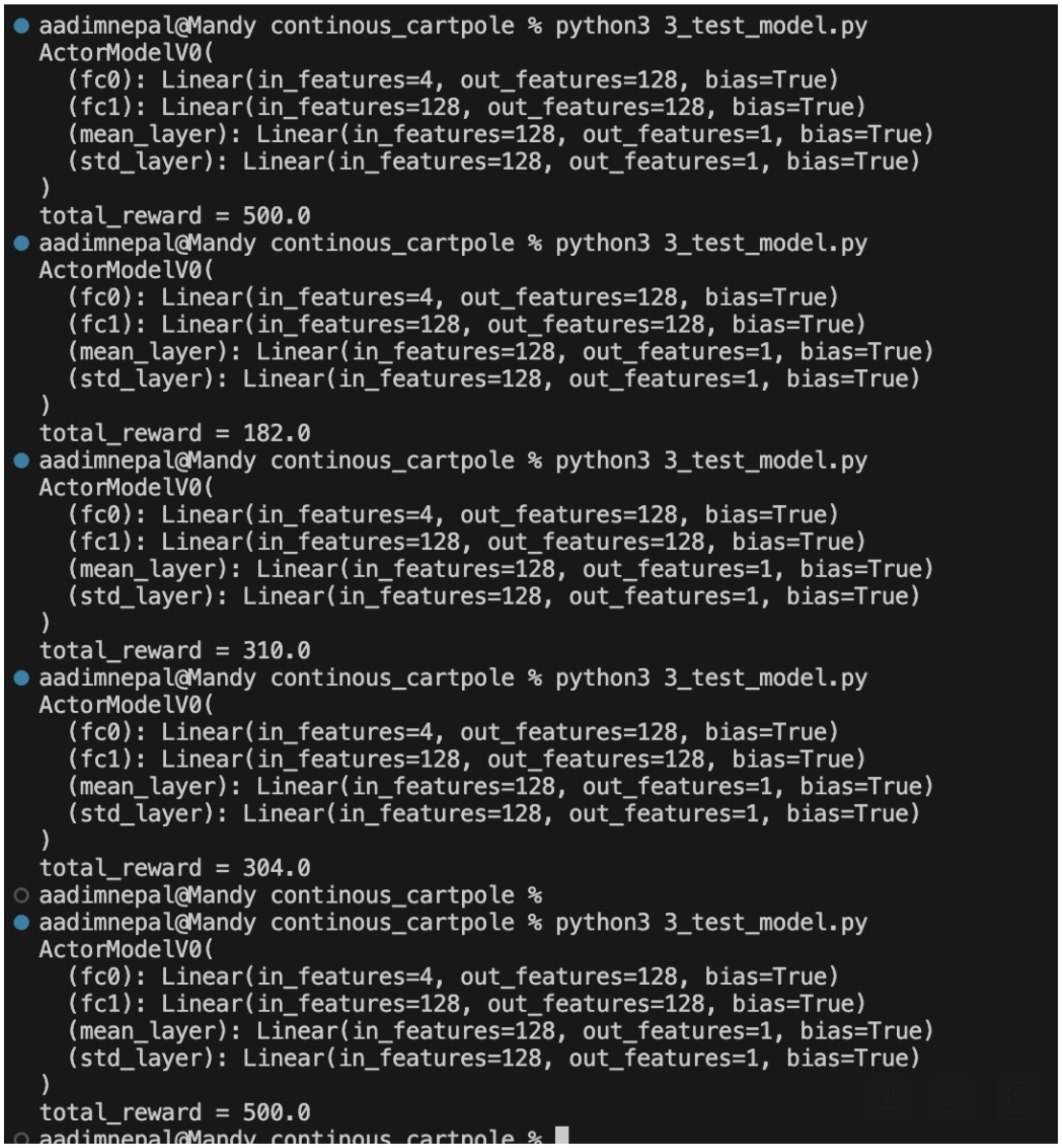
```
iteration 57710 - last reward: 259.00
iteration 57715 - last reward: 288.00
iteration 57720 - last reward: 332.00
iteration 57725 - last reward: 333.00
iteration 57730 - last reward: 362.00
iteration 57735 - last reward: 309.00
iteration 57740 - last reward: 407.00
iteration 57745 - last reward: 474.00
iteration 57750 - last reward: 440.00
iteration 57755 - last reward: 416.00
iteration 57760 - last reward: 346.00
iteration 57765 - last reward: 397.00
iteration 57770 - last reward: 387.00
iteration 57775 - last reward: 378.00
iteration 57780 - last reward: 346.00
iteration 57785 - last reward: 375.00
iteration 57790 - last reward: 404.00
iteration 57795 - last reward: 500.00
iteration 57800 - last reward: 500.00
iteration 57805 - last reward: 500.00
iteration 57810 - last reward: 500.00
iteration 57815 - last reward: 500.00
```

I modified the actor neural net to only output mean and set the standard deviation as a hyperparameter (I kept 0.1). And I decreased the learning rate to 0.0001 as per your suggestion.
My REINFORCE algorithm took around 20 minutes to converge (around 60000 iterations). Is this normal? I am already quite happy but I would love to optimize it even more. Thank you!

5.8 Relaxing the convergence for REINFORCE continuous setting

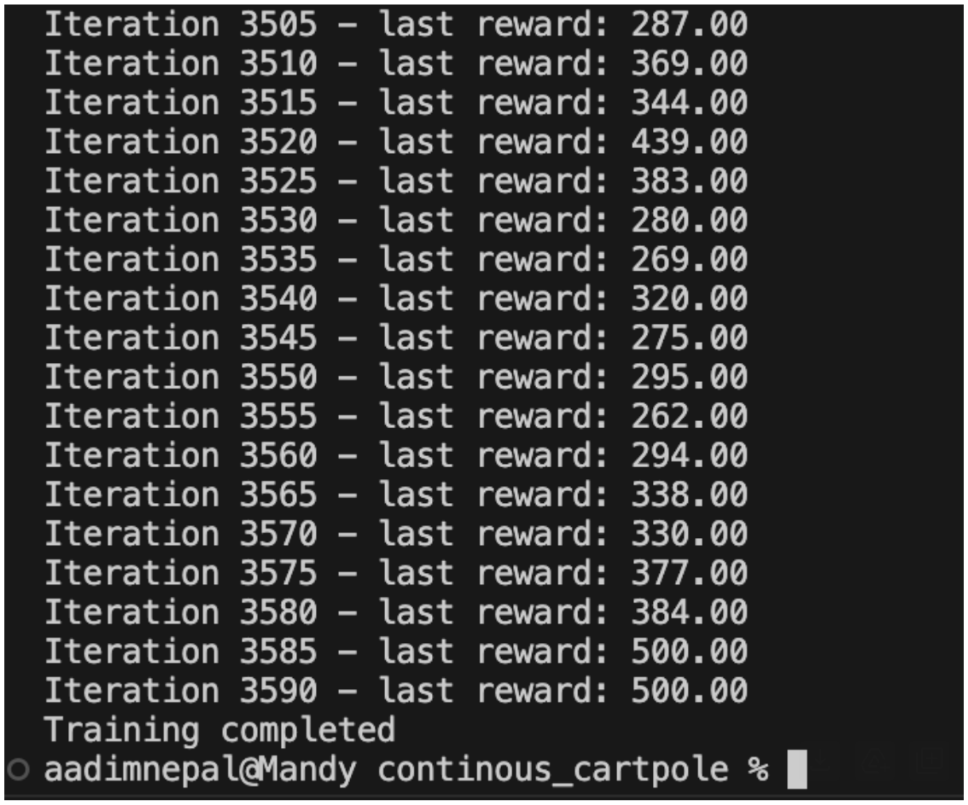


```
iteration 33900 - last reward: 159.00
iteration 33905 - last reward: 109.00
iteration 33910 - last reward: 219.00
iteration 33915 - last reward: 203.00
iteration 33920 - last reward: 244.00
iteration 33925 - last reward: 204.00
iteration 33930 - last reward: 353.00
iteration 33935 - last reward: 118.00
iteration 33940 - last reward: 401.00
iteration 33945 - last reward: 363.00
iteration 33950 - last reward: 383.00
iteration 33955 - last reward: 381.00
iteration 33960 - last reward: 331.00
iteration 33965 - last reward: 401.00
iteration 33970 - last reward: 401.00
aadimnepal@Mandy continous_cartpole % python3 3_test_model.py
ActorModelV0(
  (fc0): Linear(in_features=4, out_features=128, bias=True)
  (fc1): Linear(in_features=128, out_features=128, bias=True)
  (mean_layer): Linear(in_features=128, out_features=1, bias=True)
  (std_layer): Linear(in_features=128, out_features=1, bias=True)
)
total_reward = 500.0
```

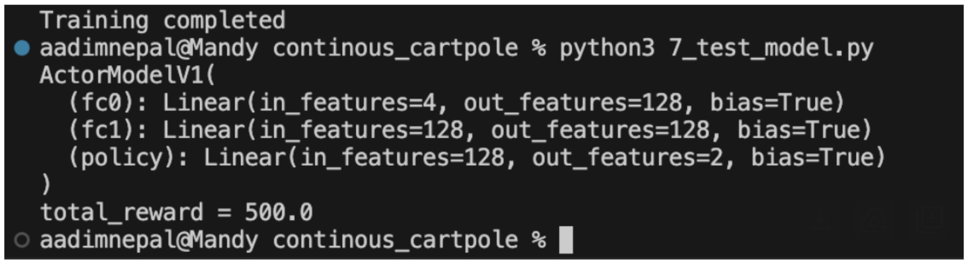


```
● aadimnepal@Mandy continous_cartpole % python3 3_test_model.py
  ActorModelV0(
    (fc0): Linear(in_features=4, out_features=128, bias=True)
    (fc1): Linear(in_features=128, out_features=128, bias=True)
    (mean_layer): Linear(in_features=128, out_features=1, bias=True)
    (std_layer): Linear(in_features=128, out_features=1, bias=True)
  )
  total_reward = 500.0
● aadimnepal@Mandy continous_cartpole % python3 3_test_model.py
  ActorModelV0(
    (fc0): Linear(in_features=4, out_features=128, bias=True)
    (fc1): Linear(in_features=128, out_features=128, bias=True)
    (mean_layer): Linear(in_features=128, out_features=1, bias=True)
    (std_layer): Linear(in_features=128, out_features=1, bias=True)
  )
  total_reward = 182.0
● aadimnepal@Mandy continous_cartpole % python3 3_test_model.py
  ActorModelV0(
    (fc0): Linear(in_features=4, out_features=128, bias=True)
    (fc1): Linear(in_features=128, out_features=128, bias=True)
    (mean_layer): Linear(in_features=128, out_features=1, bias=True)
    (std_layer): Linear(in_features=128, out_features=1, bias=True)
  )
  total_reward = 310.0
● aadimnepal@Mandy continous_cartpole % python3 3_test_model.py
  ActorModelV0(
    (fc0): Linear(in_features=4, out_features=128, bias=True)
    (fc1): Linear(in_features=128, out_features=128, bias=True)
    (mean_layer): Linear(in_features=128, out_features=1, bias=True)
    (std_layer): Linear(in_features=128, out_features=1, bias=True)
  )
  total_reward = 304.0
○ aadimnepal@Mandy continous_cartpole %
● aadimnepal@Mandy continous_cartpole % python3 3_test_model.py
  ActorModelV0(
    (fc0): Linear(in_features=4, out_features=128, bias=True)
    (fc1): Linear(in_features=128, out_features=128, bias=True)
    (mean_layer): Linear(in_features=128, out_features=1, bias=True)
    (std_layer): Linear(in_features=128, out_features=1, bias=True)
  )
  total_reward = 500.0
○ aadimnepal@Mandy continous_cartpole %
```

Figure 1: REINFORCE was not so good with continuous action



```
Iteration 3505 - last reward: 287.00
Iteration 3510 - last reward: 369.00
Iteration 3515 - last reward: 344.00
Iteration 3520 - last reward: 439.00
Iteration 3525 - last reward: 383.00
Iteration 3530 - last reward: 280.00
Iteration 3535 - last reward: 269.00
Iteration 3540 - last reward: 320.00
Iteration 3545 - last reward: 275.00
Iteration 3550 - last reward: 295.00
Iteration 3555 - last reward: 262.00
Iteration 3560 - last reward: 294.00
Iteration 3565 - last reward: 338.00
Iteration 3570 - last reward: 330.00
Iteration 3575 - last reward: 377.00
Iteration 3580 - last reward: 384.00
Iteration 3585 - last reward: 500.00
Iteration 3590 - last reward: 500.00
Training completed
aadimnepal@Mandy continous_cartpole %
```



```
Training completed
aadimnepal@Mandy continous_cartpole % python3 7_test_model.py
ActorModelV1(
  (fc0): Linear(in_features=4, out_features=128, bias=True)
  (fc1): Linear(in_features=128, out_features=128, bias=True)
  (policy): Linear(in_features=128, out_features=2, bias=True)
)
total_reward = 500.0
aadimnepal@Mandy continous_cartpole %
```

Figure 2: The Ultimate Convergence of Actor Critic in Continuous Action Space