CMPSCI 687 Final Report - Fall 2021

Hongyu Tu

1. Model Problem as MDP

The problem I decided to take a look at is the card game, blackjack. I think it will be a good fit as a problem that can be solved with reinforcement learning through coming up a control algorithm. To make this problem fit MDP setting better, I simplified it to be single player game as the goal is just to get as close as 21 as possible and avoid getting busted by going over 21. Rules like insurance and card calculation (analyze what card has been given out and what has not) are reduced as well as we have a single player game.

(a) Formal Definition of the MDP
The MDP that I come up for blackjack will include these parts: $(S, A, R, p, d_0, \gamma)$.

- **S (state)**, will be represented with two numbers, [not_a, ace]:
  1) sum of value of all cards except aces $\in [0, 30]$
  2) number of aces $\in [0, 4]$
  From the rule of blackjack, a player can have one or more cards in hand within the set of (1 to 10, J, Q, K). Card with number 2 to 10 will represent its original value; J, Q, K represents 10 as well; and ACE (1) is the special one that can represent 1 or 10, under different circumstances. From that, I reduces the set of possible cards to be 10 categories: 2 to 10, and ACE (1).
  Once a player gets a card, the card will stay and can not be discarded, therefore, we only care about the total value and not the exact type of cards that are in our hand (expect the ACEs). That's where the first number comes in. This MDP will only keep track of the total value of the non-ACE cards in hand. The maximum value a state can have is 30, if the original value in hand is 20 (not busted) and got a new card of value 10.
  Due to the special property of ACEs and its effect on the value, the second number will represent the number of ACEs in a player's hand. For example, if the sum of the value in hand expect ACE is 5, and the player has 1 ACE in hand. The total value can be either 6 or 15. More complexity can occur when we get more ACEs. How I implemented it will be discussed in detail in later part. The maximum value can occur is 4 as I defined to have 1 set of cards.

- **A (Action)**, will have three possible options:
  1) Not hit: keep the same cards and wait next round.
  2) Hit: Get a new card from the distribution of 52 cards. I looked at the distribution of cards for probability. For numbers 2 to 9, there are 4 each out of 52 cards, so for each of the 8 classes have a probability of $\frac{4}{52}$. For card 10, J, Q, K, all of them have value 10, and getting one of those cards that represent value 10 will have probability $4 \times \frac{4}{52} = \frac{16}{52}$. Lastly, for the special ACE that can take value 1 or 10, the probability of getting an ACE is again $\frac{4}{52}$. Just to check the probability here, for class 2 to 9 and ACE, all 9 of classes have probability $\frac{4}{52}$, which sums up to $9 \times \frac{4}{52} = \frac{36}{52}$, and adding the $\frac{16}{52}$ coming from value of 10s, we get $\frac{16}{52} + \frac{36}{52} = \frac{52}{52} = 1$. This makes sure that the probability is correctly distributed amoung all possible cases.
  3) End Game. This turns all states to terminal state and ends the episode.

- **p (Transition) and R (reward)**
  Here, the next state will be determined by the original state and the three actions above.
  1) Not hit: since no card was drawn, the sum of values of non-ACE card will not change, and the number of ACE will not change. Therefore, the new state remains the same as the original state, and no reward is generated (0). A limit of 15 is set for the steps within an episode, so the game will terminate after that.
  2) Hit: As mentioned above, a new card will be added to the deck. Based on the type of the card, if the new card is ACE, the number of ACE in next state will be added by 1, and the sum of value of non-ACE will remain the same: ((A,B) $\rightarrow$ (A,B+1)); if the new card is not ACE, the sum of value of non-ACE will be added with the new card's value, and the number of ACE will remain the same: ((A,B) $\rightarrow$ (A + val(new card),B)). If the total value of cards on hand exceeds 21, then we bust and get a reward of -5.
  3) End game: if the card on hand is not above 21, then the maximum possible value of the cards in hand will be counted as the reward, and we reach terminal state. For example, if we are in state (5, 1), it can have value 6 or 15 at the same time (one ACE count as 1 and 10 depending on situation), and here the reward will be 15 if the player decided to end game. To encourage the agent to try to hit exactly 21, an additional reward of 5 will be granted if the state has value 21 and decided to end game.

- **d_0 (Initial state)**
  During training, to explore more states, both numbers in the state will be randomly generated with in the range; during testing, it will simply be (0,0).

- γ **(Discount factor)**
  γ is defined to be 1 as here the goal here is just to collect as much value as possible within 15 steps, the value of the card does not change with the time.

(b) Implementation detail
For the setup, the MDP is fully in python and only needs PyTorch and Numpy as the external library. In order for this MDP to work, we will need a way to determine the values a state can represent, due to the special feature that ACE can represent either 1 or 10 under different circumstances. Therefore, besides the standard functions like init or step, I wrote a function called get value which takes in a state and returns a set of valid values (smaller or equal to 21) that it can take.
The way the function works is look at the number of ACEs in the state, as that is where the combination of values are from. More specifically, with number of ACE = 1, the list can take 2 value from the ACE: [1, 10]; with number of ACE = 2, we have [(1,1),(1,10),(10,1),(10,10)] $\rightarrow$ [2, 11, 11, 20].
The procedure is the same for 3 and 4, and we will get 2^3 and 2^4 combinations for each case. On top of that, the value from non-ACE value will be added to the list so we will get the list of possible values that all the card combined can take. And lastly, before return, duplicates and values larger than 21 will be removed as they are not valid, and the list will get sorted for convenience late.
With this function, we will be able to tell if a state has busted by looking at the list. If the list is empty, that means even the minimum value that it can represent is larger than 21. When the agent decided to end, looking at the last element of the list will give the largest possible value that the agent can take, which maximizes the effect of ACE.

(c) Solve the problem with algorithm tuning
I have tried to solve this problem with the three algorithms in part 2, find more detail below.

2. Implementation of uncovered algorithms

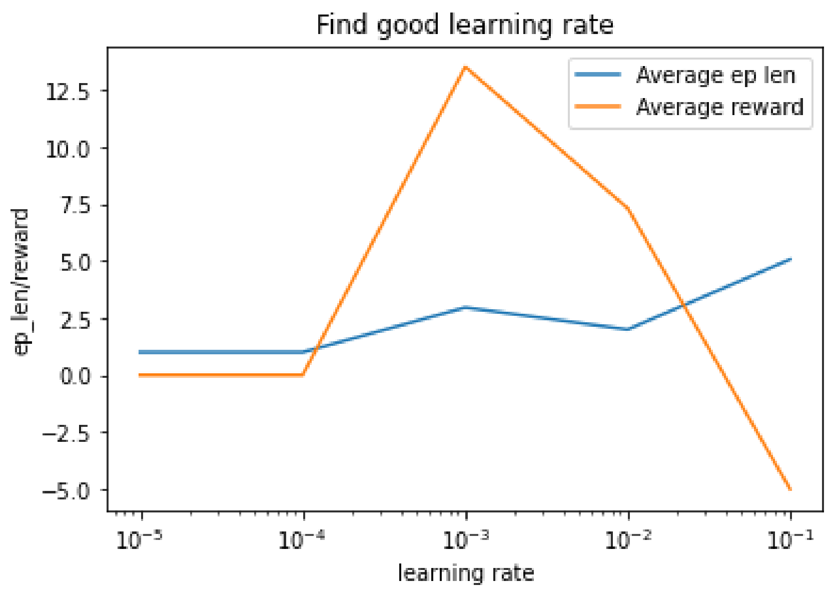
In this part, I looked at three uncovered algorithms: Reinforce with Baseline, One-Step Actor-Critic and Episodic Semi-Gradient n-step SARSA. All three of them were tested with 4 MDPs: 687 Grid World, Mountain Car, Blackjack (from previous part) and Cart pole (Benchmark, limit set to 200 steps). I will go over each of them separately below and then show the overall comparison at the end.

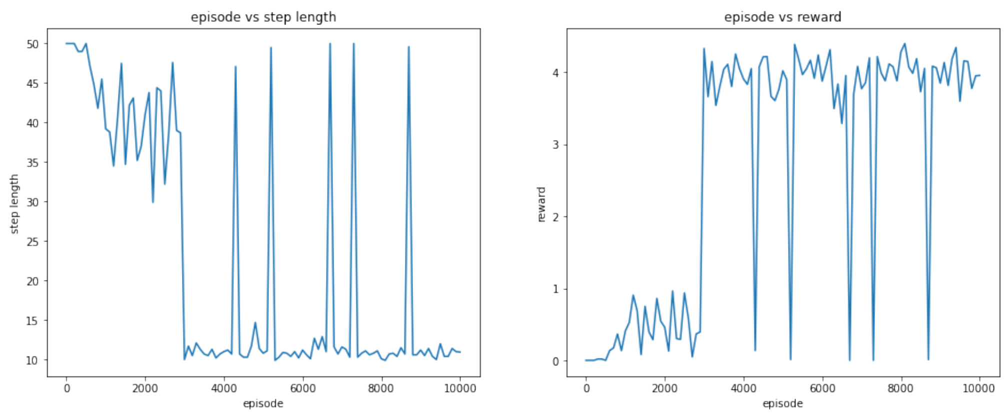
(a) Reinforce with Baseline

- **Brief Description**
  For this algorithm, the baseline is set to value function so that it converges faster. Here in the algorithm, the return is estimated with the observed sum of discounted reward, and it will be used to update the value function estimation, where the difference between the discounted return and value estimation will be used to update the log likelihood of the policy network.
- **Pseudo code**

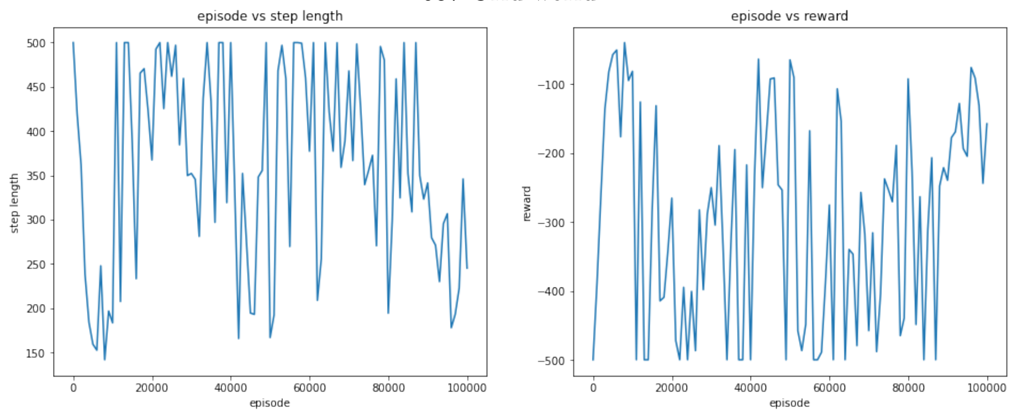
```
REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$
```

- **Hyper-parameter Tuning & Results**
  For this algorithm, the most import hyper-parameter to tune is the learning rate, to find the best one, I tried value within 1e-1 to 1e-5 and plot the return and episode length vs the learning rate. It turns out that 1e-3 is the best learning rate to use.

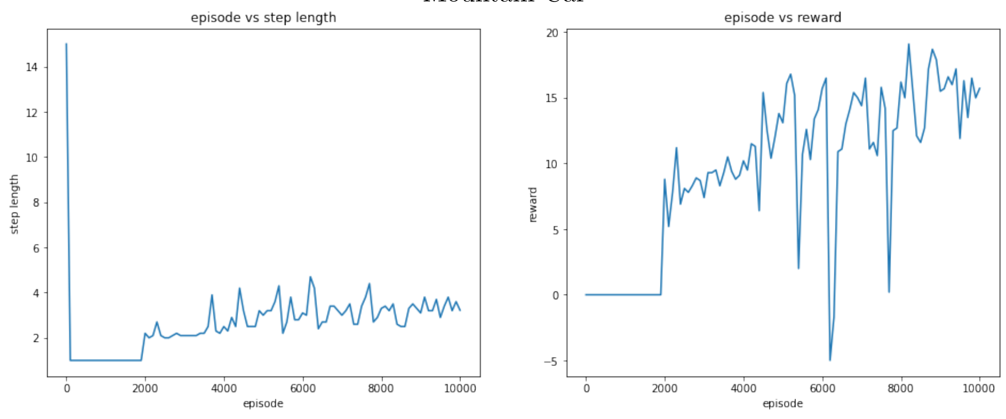




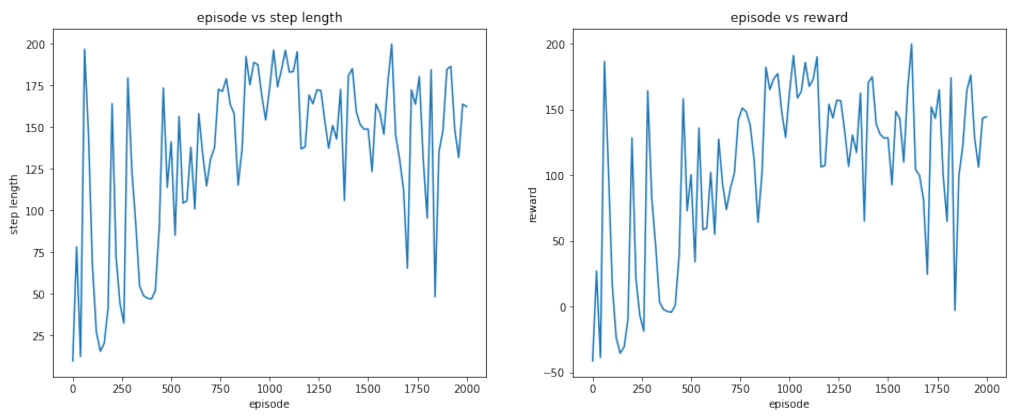
687 Grid world



Mountain Car



Blackjack



Cart pole

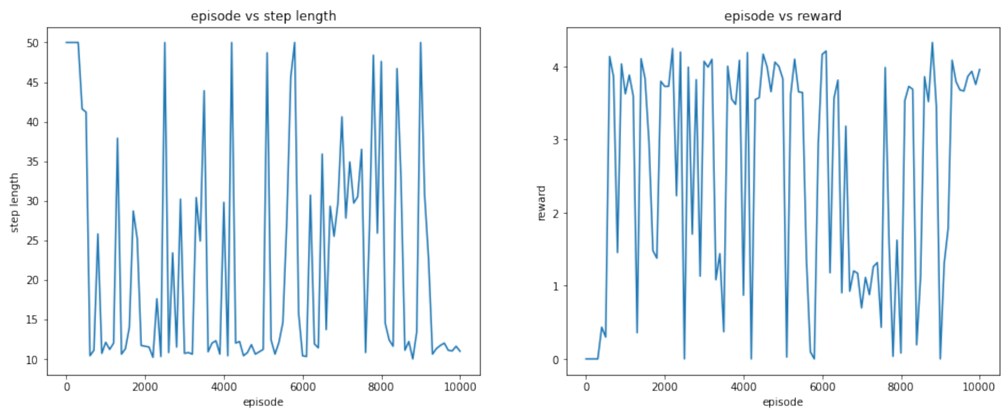
(b) One-Step Actor-Critic

- **Brief Description**
  Overall, this algorithm is really similar to the previous one, where the actor network is the policy network that will control the agent, and the critic network is basically the value function estimation network that evaluates how good the actor is performing. A major difference in this algorithm, however, is that bootstrapping is used on each step when estimating the return. Instead of summing up all later discounted reward, actor-critic uses only the current reward and an estimate of the value of the next state. Actor and critic is updated the same way as the reinforce when only one step is looked at. One thing that is important for this network to work well is that both network need to improve at the same time, because a bad critic will not give good score for the actor so the actor will not learn good action.

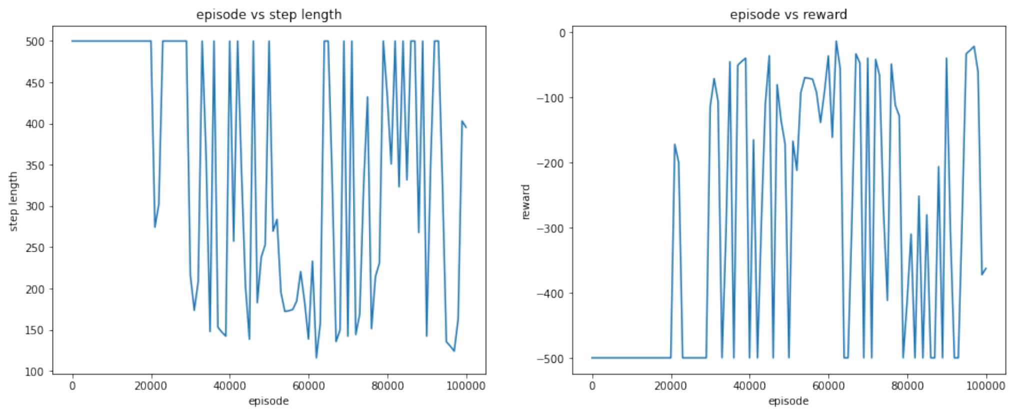
- **Pseudo code**

```
One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
    Initialize $S$ (first state of episode)
    $I \leftarrow 1$
    Loop while $S$ is not terminal (for each time step):
        $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
        Take action $A$, observe $S', R$
        $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$        (if $S'$ is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla\ln\pi(A|S,\boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
```

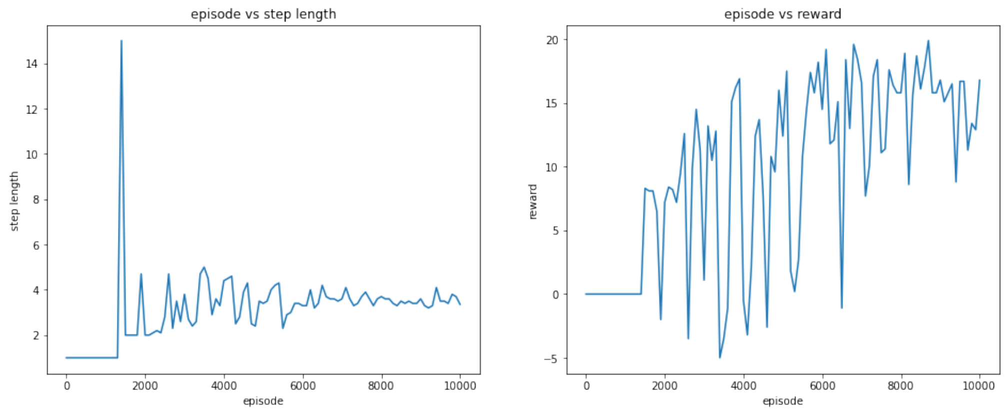
- **Hyper-parameter Tuning & Results**
  Through many trials, I found that it usually takes more episodes for this algorithm to converge or perform at a high level. To speed up the training, I tried to increase the learning rate for both the actor and critic from 0.001 to 0.1. However, that did not help at all as the learning rate is too height and it keeps getting stuck at local minimum. Therefore, in the end, I changed the learning rate back to 0.001 and simply run this algorithm longer in order to solve the problem. One advantage though, is that the critic in this algorithm does an excellent job at estimating the value function, which is a lot closer than the approximation than the one that was produced by reinforce with baseline.



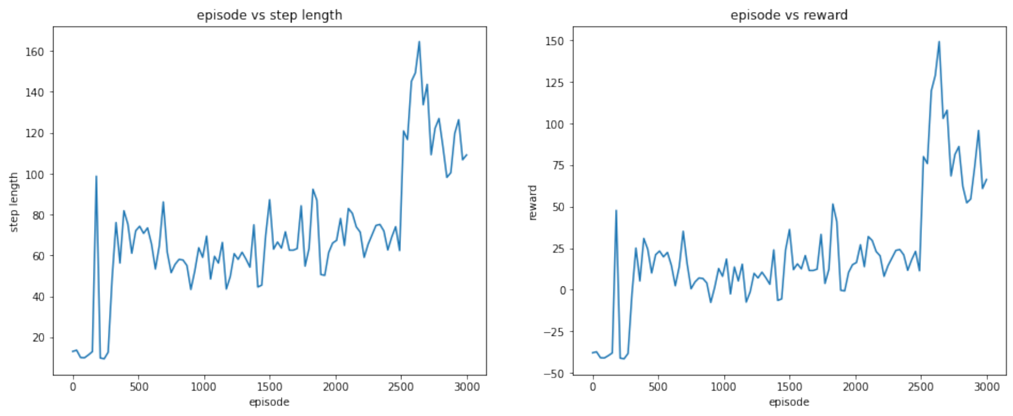
687 Grid world



Mountain Car



Blackjack



Cart pole

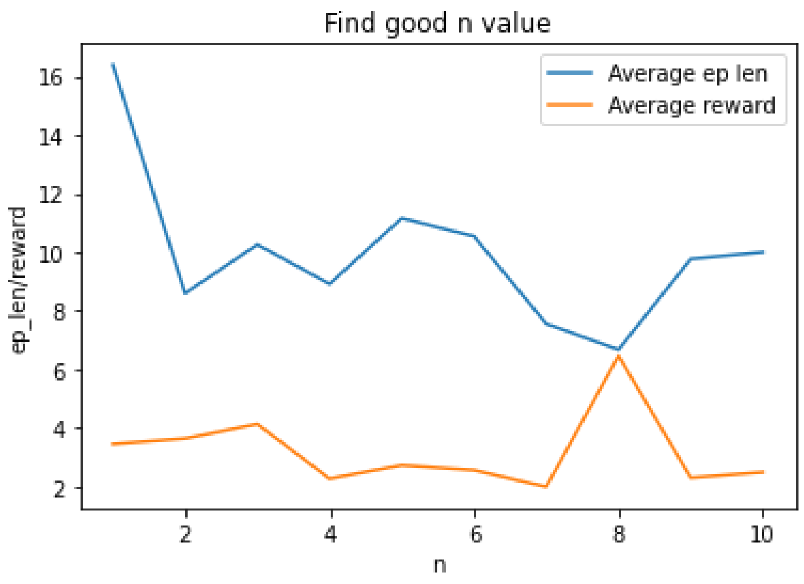
(c) Episodic Semi-Gradient n-step SARSA

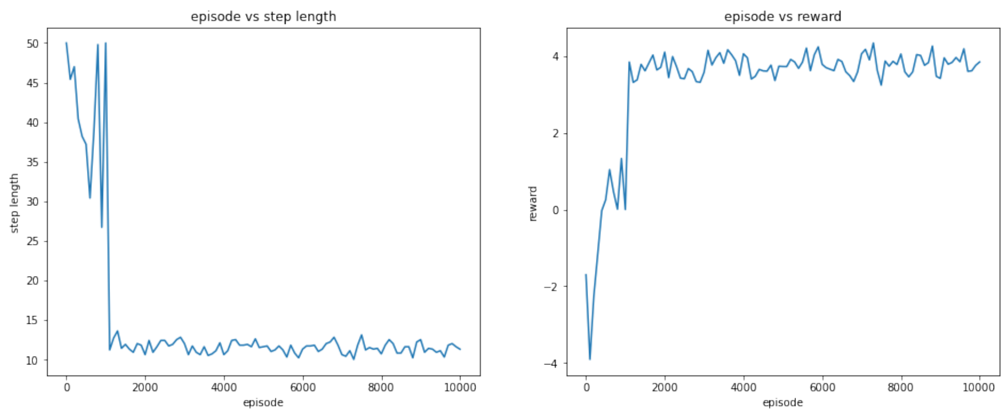
- **Brief Description**
  In this algorithm, it has a lot of difference when it comes to the network. It only uses one network to estimate the state-action value and the policy will be ϵ-greedy based on the state-action function. Unlike the one-step algorithm in previous part, the n in this algorithm will determine how much of the return is from exploration and how much is from bootstrapping, and that's where the name semi from. Using more n will give us more accurate value whereas less n will make the algorithm lean toward bootstrapping.
- **Pseudo code**

Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

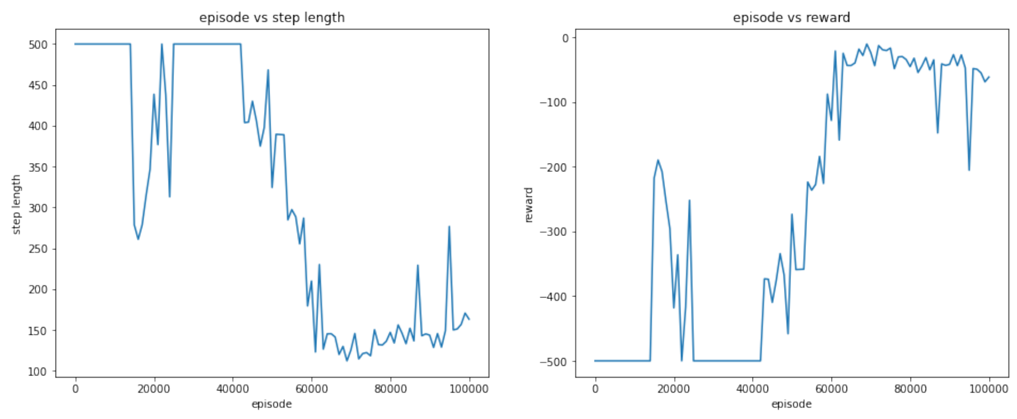
```
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy $\pi$ (if estimating $q_\pi$)
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer $n$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations ($S_t$, $A_t$, and $R_t$) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or $\varepsilon$-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$ :
  |  If $t < T$, then:
  |      Take action $A_t$
  |      Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
  |      If $S_{t+1}$ is terminal, then:
  |          $T \leftarrow t + 1$
  |      else:
  |          Select and store $A_{t+1} \sim \pi(\cdot|S_{t+1})$ or $\varepsilon$-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
  |  $\tau \leftarrow t - n + 1$   ($\tau$ is the time whose estimate is being updated)
  |  If $\tau \geq 0$:
  |      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  |      If $\tau + n < T$, then $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$          $(G_{\tau:\tau+n})$
  |      $\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{q}(S_\tau, A_\tau, \mathbf{w})] \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$
  Until $\tau = T - 1$
```

- **Hyper-parameter Tuning & Results**
  For this algorithm, the most import hyper-parameter to tune is the n, to find the best one, I tried value within 1 to 10 and plot the return and episode length vs the learning rate. It turns out that 5 is the best learning rate to use. This one is tested on the blackjack environment and the same process goes to the all other models.

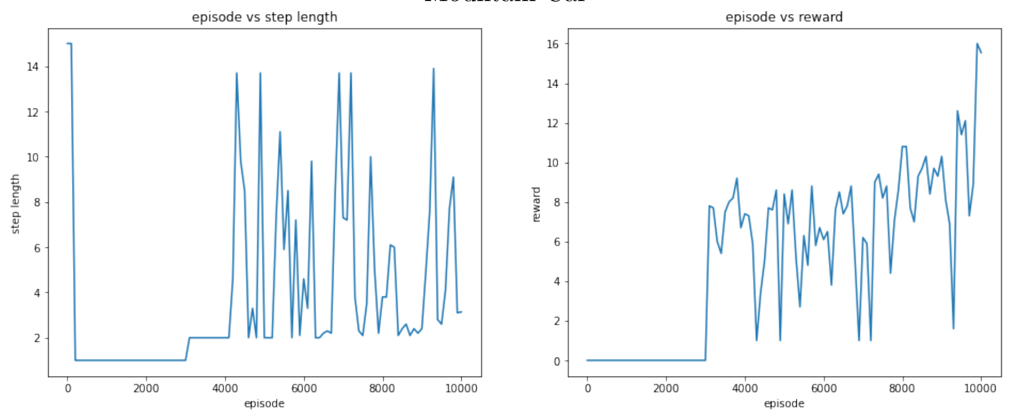




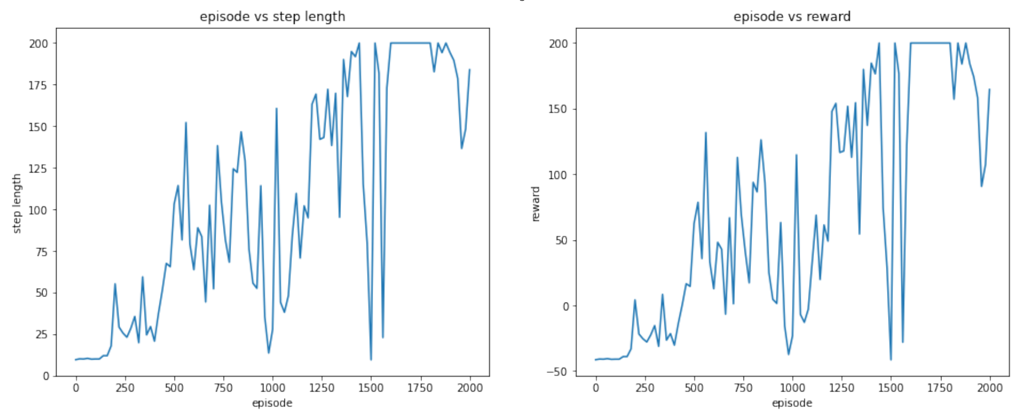
687 Grid world



Mountain Car



Blackjack



Cart pole

(d) Overall Performance Comparison

- **Average return over 100 runs**

Env vs. Algorithms	Reinforce with Baseline	One-Step Actor-Critic	Episodic Semi-Gradient n-step SARSA
687 Grid World	3.9528	3.9587	3.8514
Mountain Car	-157.36	-362.82	-362.82
Blackjack	15.72	16.79	15.54
Cart pole	144.61	94.93	164.58

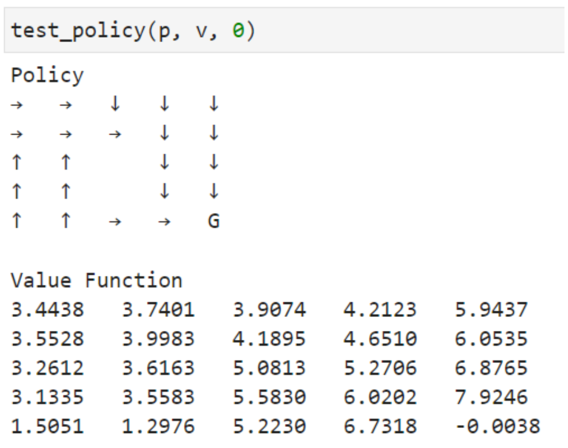
- **Average episode length over 100 runs**

Env vs. Algorithms	Reinforce with Baseline	One-Step Actor-Critic	Episodic Semi-Gradient n-step SARSA
687 Grid World	10.95	10.97	11.29
Mountain Car	245.08	395.46	395.46
Blackjack	3.22	3.36	3.14
Cart pole	162.46	130.12	183.96

Here for 687 grid world, the most optimal policy will get discounted return after 8 step, which is $0.9^8 \times 10 = 4.3$, here the policy was able to get a reward of near 4.0, given that steps have

changes to go in wrong direction, it's pretty good performance. The average steps also proved the point, all three algorithm found policy to solve the problem with around 10 steps.
For mountain car, reinforce with baseline performed really well whereas the other two are not as good. the best policy solved the problem within 250 steps and the other two taken about 400 steps. All lower comparing to the optiaml policy that can solve the problem with around 150 steps.
For blackjack, given the max point is 21, on average the agent under the policy produced by three algorithms are all around 15 to 17, which is good as well as higher value will lead to a higher probability of bust the card in hand. If we look at the steps per episode, it on average takes about 3 to 4 steps before ending the game, which means it learned not to wait, even when we have a discount factor of 1.
Lastly for cart pole, one step actor critic didn't perform well with holding on for only 90 steps. However, it's still an big improvement when compared to an untrained model, which has an average return of -50 (I changed the setting for cartpole to get a -50 penalty when end so that it will learn faster). On the other hand, reinforce with baseline and SARSA both got around 150 steps, which is an ideal output after only 2000 runs. Also, when looking at the episode length of actor-critic, it's 130. Its difference with the average reward much larger than the other two. I think it can be a sign of mixed situation of ending really soon and have control for 150 episodes.

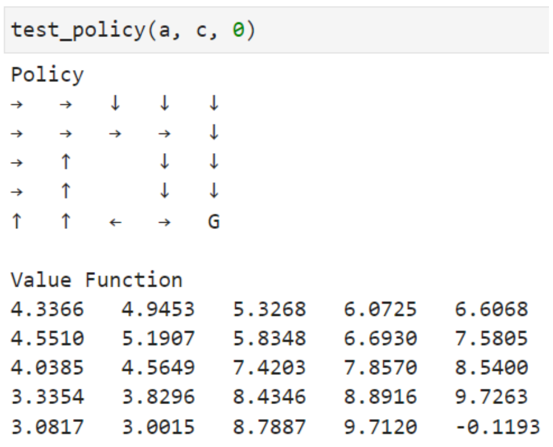
With the experiments above, it might seem that actor-critic algorithm doesn't have any advantage over the other two. However, when looked at the output model, specifically for 687 grid world, the value function is a lot better.



```
test_policy(p, v, 0)
```

```
Policy
→    →    ↓    ↓    ↓
→    →    →    ↓    ↓
↑    ↑         ↓    ↓
↑    ↑         ↓    ↓
↑    ↑    →    →    G

Value Function
3.4438   3.7401   3.9074   4.2123   5.9437
3.5528   3.9983   4.1895   4.6510   6.0535
3.2612   3.6163   5.0813   5.2706   6.8765
3.1335   3.5583   5.5830   6.0202   7.9246
1.5051   1.2976   5.2230   6.7318   -0.0038
```

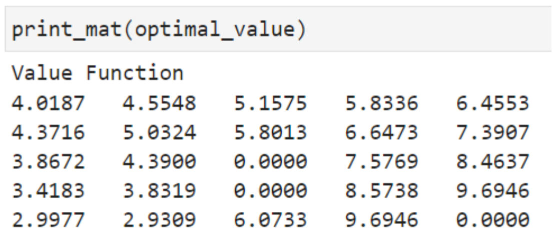
output of reinforce with baseline



```
test_policy(a, c, 0)
```

```
Policy
→    →    ↓    ↓    ↓
→    →    →    →    ↓
→    ↑         ↓    ↓
→    ↑         ↓    ↓
↑    ↑    ←    →    G

Value Function
4.3366   4.9453   5.3268   6.0725   6.6068
4.5510   5.1907   5.8348   6.6930   7.5805
4.0385   4.5649   7.4203   7.8570   8.5400
3.3354   3.8296   8.4346   8.8916   9.7263
3.0817   3.0015   8.7887   9.7120   -0.1193
```

output of actor critic



```
print_mat(optimal_value)
```

```
Value Function
4.0187   4.5548   5.1575   5.8336   6.4553
4.3716   5.0324   5.8013   6.6473   7.3907
3.8672   4.3900   0.0000   7.5769   8.4637
3.4183   3.8319   0.0000   8.5738   9.6946
2.9977   2.9309   6.0733   9.6946   0.0000
```

True optimal value function

Lastly, another thing that I noticed is that cartpole environment has a limit of 200 steps. To try out the best I can do, I changed the limit to 500 and trained a model that can run to 500+ steps before getting stopped. I saved the trained model as cartpole_500.pt and the driver to do ten tests are in file cartpole_demo.py.

(P.S. Thanks for this semester! I did a lot of extra things by myself for this final project because I true enjoyed this class and I have learned a lot. Cheers!)