**Part 2)**

The following plot shows the state-value function for the easy21-environment. It is an estimation based on 1 000 000 episodes with Monte Carlo learning:

Ein Bild, das Text, Display, Software, Screenshot enthält.

Automatisch generierte Beschreibung

**Ein Bild, das Text, Screenshot, Software, Display enthält.

Automatisch generierte BeschreibungPart 3)**  
1000 episodes is perhaps too little when we’re trying to find the optimal value for lambda, as the lambda-value which gets the lowest MSE changes from time to time.

Ein Bild, das Text, Screenshot, Software, Display enthält.

Automatisch generierte BeschreibungBy looking at comparison between lambda = 0 and lambda = 1 (MC learning), we can see that bootstrapping generally performs better than Monte Carlo. It learns faster at the start, and it even seems to give a better result over large timescales.

However, the estimates of the state-value function seem to be relatively bad at the very start of the training session:

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

**Part 4)**

We can see from the plots of MSE after 1000 episodes, that linear approximation of the value-function seems to perform better with fewer episodes:

Ein Bild, das Text, Screenshot, Diagramm, Reihe enthält.

Automatisch generierte Beschreibung

But when we look at larger timescales, we can see that this linear function approximation algorithm seems to have problems with convergence, and that is to be expected, given the rough picture of the state-space it has.

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

The conclusion seems to be that linear function approximation works well when you have few training episodes, but if you have enough episodes to train from, then Sarsa(lambda) seems to be the better algorithm.

**Part 5)**

**What are the pros and cons of bootstrapping in Easy21?**

From the graphs presented in part 3, it seems that bootstrapping leads to faster convergence to the optimal action-value function. We can see that for lambda = 1, which is Monte Carlo learning, we have the highest reported mean squared error after 1000 episodes, out of all of the lambda values. We can also see that regular TD(0) outperforms MC learning, even after 60 000 episodes.

The con of bootstrapping is as always that you get biased estimates of the action-value function but that you get less variance than Monte Carlo. With that said, both MC learning and TD learning should converge to the true value-function in the long run.

I don’t see any immediate disadvantage which comes with the use of bootstrapping, it just seems to be better than MC learning in the Easy21 environment. This observation is also in accordance with the heuristic that some bootstrapping is almost always better than no bootstrapping.

**Would you expect bootstrapping to help more in blackjack or Easy21? Why?**

Seeing as Easy21 is an environment where the episodes might be longer than in regular blackjack, you would expect bootstrapping to be more helpful in the Easy21-environment. The number of different trajectories which you can take from a specific state is just so much larger in Easy21, since you have the possibility of drawing a card which gives you a negative number added to the total.

Inherently, bootstrapping performs better than MC in environments where there is more variance, and with the assumption that the Easy21 environment has higher variance than Blackjack, I am able to conclude that bootstrapping will help more in Easy21 than in blackjack.

**What are the pros and cons of function approximation in Easy21?**

One advantage of using function approximation in Easy21 seems to be that it generalizes better when you have a small sample size. In our experiment, the sample size refers to the number of episodes you are allowed to train on. The tabular methods with Sarsa or MC-learning, are trying to give an action value to each of the different states in the environment. When we have few episodes, the action-values which are assigned to these states will have substantial variance, as there probably haven’t been many updates to them.

The linear function approximator on the other hand, will try to generalize for hands with sum in the range [2, 6] for example, and will have more samples to work out the action-value of that hand-range. This helps in the short term, when you don’t have many episodes to work with, but as we have seen, this leads to long term problems.

Tabular algorithms will outperform this linear approximation in the long term, as they simply have more accurate representations of the state and are able to provide more precise estimations of the action-value of a state, given enough compute and training data.

So the advantage of function approximation is that it quickly gets a decent picture of the action-value function with little data, but it has the disadvantage that it will never be able to find the true action-value function, since it doesn’t have full knowledge of the state space.

**How would you modify the function approximator suggested in this section to get better results in Easy21?**

The modifications which must be made depend on what you mean by better results. Do you want a faster convergence to some decent action-value function with few episodes, or do you want a function approximator which performs better in the long run, and which converges to something close to the true action-value function long term?

For example, if we wanted faster convergence to something decent, then we could look at q\* (the optimal action-value function), and we could look at which intervals which have almost the same actions which is picked. For example, if we know that the optimal policy always prefers to stick if the dealer shows a card in range [1, 3], then you could have dealer sum in range [1, 3] as a feature. The thing I am discussing now basically comes down to finding better features, which isn’t always as easy. If you don’t have the oracle q\*, then it is hard to know which features are good, and in all practical problems you don’t have q\*, since if that is the case, then there is no problem to solve.

Apart from adjusting the features, you could have implemented a decaying epsilon, such that the policy gets greedier over time, and you could do the same thing with the learning rate.

But other than that, the only way to improve the algorithm is to change your features somehow. And to do that, you just need to follow some good heuristics. For example, I would assume that if you have some states where the difference in action value between the best and the next best action is small, then you would probably like to add some extra features which would apply for those states.

As an example, if it turns out that there is just a small difference between the expected future reward for hit and stick when the dealer shows a 5, then you can add that as a separate feature.

If your goal is to get a good approximation of q\* long term, then I think you have chosen the wrong algorithm, as tabular methods will be better when we work with environments as small as Easy21. But you could perhaps get faster convergence if you in addition to having one feature for each state, have extra features like: player having card sum in range [0, 5]. I don’t know if it is a good idea, but it is something.

And at the end we can mention that a non-linear neural network would probably work better as a function approximator, but seeing as the state-action space in Easy21 is so small, it looks like overkill to implement a neural network here. The tabular methods should work just fine for this problem, and we have the guarantee that they will converge to the optimal policy. There is no such guarantee for non-linear neural networks, although they would probably be able to get a much better estimate of q(s, a) after a small amount of episodes.