An Implementation of Uncertainty-Aware Action Advising for Deep Reinforcement Learning Agents Applied to DDQN

Simon Gibson, Russell Harter - Learning and Advanced Game AI

***Abstract*— Human learning is not a solitary action. Despite our best efforts, we often cannot resolve the answer to a question by ourselves. In times such as this, passing the torch, even momentarily, to another, more knowledgeable, person is beneficial. In the same vein, we look to apply this type of learning to classical reinforcement learning techniques. Uncertainty Aware Action Advising for Bootstrapped Double Deep Q-Learning enables the ability for the network to ask for help in times of uncertainty. This paper describes our application of this algorithm to classic OpenAI gym environments, our successes and failures, and key insights into the world of Reinforcement Learning.**

I. INTRODUCTION

In the field of Artificial Intelligence, techniques are being constantly improved upon and reworked. There is a lot of contemporary research being done on a multitude of different algorithms, each having advantages and disadvantages. The paper that we focused on in this implementation offers a way to augment these algorithms by adding uncertainty-aware action advising. As we'll get into more detail with later, this is a technique that allows the agent to ask for advice from a demonstrator while training, allowing them to train faster. In this paper, we focus on this concept as applied to Double Deep Q-Learning Networks.

II. BACKGROUND

To understand the algorithm behind Uncertainty-Aware Action Advising, first a clear understanding of DQN, DDQN, DDQN with Experience Replay, and Bootstrapped DDQN is necessary. This paper will only briefly cover the aforementioned topics and assumes knowledge of Q-learning.

Deep Q-Learning Networks are an application of Q-Learning, but instead of a table to store and modify Q values, a neural network is used. This neural network has an input vector size equal to the number of observations in a given state of the environment and an output vector equal to the size of the action space of the environment. This output vector will act as the Q values for each potential action for the given state. Note that you can also have multiple vectors with the same input space but with an output dimension of 1, each network outputting a single Q value. We opted for one network that outputs all the Q values. Every step of the environment, the network is trained using gradient descent on the value it predicted for the timestep of the environment (predicted value) compared to the actual value (target value).

Knowing DQN, we move on to Double Deep Q-Learning Networks, or DDQN. These networks follow the same concepts as DQN but provide a more accurate estimation of Q values, as DQN is known to overestimate in certain scenarios. In DDQN, as the name suggests we have two networks which we'll call a predictor network and a target network. The predictor network is the network which will provide the action to take and it will be the network that trains every step. The target network is the network that will be used in the computation of the target value. Every constant number of timesteps, the weights of the target network are set equal to the weights of the predicted network.

The next addition is Experience Replay. Every transition (State, Action, Reward, Next Action) from the environment is saved, up to a certain limit past which we remove the oldest transition. Then, instead of training on the transition that we just took, we instead take a minibatch (sample) from the memory to train on. This decontextualizes the training, allowing for a more consistent policy, but more importantly it increases the efficiency of the algorithm as each transition can be trained on multiple times.

The final concept required is Bootstrapped DDQN. In this case, there are multiple head networks and sometimes a core network. Each head is essentially it's own DDQN network, having a target network and a predictor network. The core network's role is to give the heads input that they can work with. A typical example is the core network being a convolutional neural network to interpret an image into a more realistic input dimension for the head networks. Each head is initialized with unique random weights separate from one another. To know what action to take, a head is randomly selected and the Q values for that head are used. For training, experience replay works largely the same. A single minibatch is selected for all the heads. Then, for each head, you randomly remove samples from the minibatch and then train the head as you do in normal DDQN. This results in the heads being trained on very similar, but not exactly the same, data. Eventually, the heads should converge and start to output very similar Q values. The major benefit of this is that this results in a more advanced exploration of the environment and can be used with ϵ-greedy or without it.

III. ALGORITHM OVERVIEW

In this section we describe a high-level analysis of the original algorithm proposed by the authors. Throughout this explanation, we will be performing all mathematical operations in a tensor form, unless otherwise noted.

A. Calculating uncertainty

Previous papers such as Agents Advising Agents in Reinforcement Learning (*Torrey and Taylor 2013*) use differing metrics for which to rate a models confidence at some state.

In this paper, the authors choose to represent confidence with a simpler is better attitude. Confidence is achieved by calculating the average variance over all possible actions and all head in the action space of an environment. For the sake of simplicity, it is assumed that action spaces are discrete, as continuous actions provide much strain for value-based learning.

$$\mu(s) = \frac{\sum_{a \in A} \text{var}(\mathbf{Q}(s,a))}{|A|} \tag{1}$$

where $\mathbf{Q} = \begin{bmatrix} Q_(s,a) \\ \cdots \\ Q_h(s,a) \end{bmatrix}$, $Q_i(s,a)$ is the Q-value given by head *i*, action *a* and state *s*. As described by the authors "The final value prediction (used, for example, for extracting a policy from the value function) is the average of the predictions given by each head". However, in practice we found this not to work and used a different method as described in the following sections.

B. Training Episode scheme

According to the original authors, the training episode scheme starts calculating the uncertainty as shown in equation 1 and used in the below algorithm.

Algorithm 1: Requesting Confidence Moderated Policy Advice (RCMP)

1 **Require** *Value function approximator* $\hat{Q}$*, agent policy* π*, uncertainty estimator* μ*, demonstrator* π_Δ*, availability function* $\mathcal{A}_t$
2 **for** $\forall$ *learning step* t **do**
3 *Observes*
4 $\mu \leftarrow \mu(s)$
5 **if** μ *is high and* $\mathcal{A}_t$ **then**
6 a $\leftarrow \pi_\Delta$ (s)
7 **else**
8 a $\leftarrow \pi(s)$
9 **end**
10 Apply action a and observe s', r
11 Update $\hat{Q}$ with (s, a, r, s')
12 **end**

This algorithm is quite digestible. In line 5, we calculate uncertainty according to equation 1. In the following lines, we either ask for the demonstrator to give an action, or we compute an action by the default policy (such as ϵ-greedy).

The authors provide another algorithm, which describes the training method referenced in line 11 of RCMP.

Algorithm 2: Implementation of DDQN with heads

1 **Require** *minibatch* $\mathbf{D}$*, Q-network* Q*, target network* Q^t*, sample selecting function* d*, number of heads* h
 $(\mathbf{s}, \mathbf{a}, \mathbf{r}, \mathbf{s}') = \mathbf{D}$
2 $\mathbf{act} \leftarrow one\_hot(\mathbf{a})$
3 $\mathbf{q^t} \leftarrow \max_a Q^t(\mathbf{s}')$
4 $\mathbf{q} \leftarrow Q(\mathbf{s})\mathbf{act}$
5 $\mathbf{p} \sim d(\mathbf{D}, h)$
6 **for** $\forall i \in \{1, \ldots, h\}$ **do**
7 $\mathbf{target} \leftarrow \mathbf{r} + \gamma \mathbf{q}^t[i] \odot \mathbf{p}[i]$
8 $\mathbf{pred} \leftarrow \mathbf{q}[i] \odot \mathbf{p}[i]$
9 $loss[i] \leftarrow \frac{1}{|\mathbf{D}|} \sum(\mathbf{target} - \mathbf{pred})^2$
10 **end**

This algorithm is much less digestible than RCMP. However, if we convert it into its scalar form it is actually quite simple. In overview, the algorithm trains each head on the minibatch. However, in order to prevent the heads converging to the same network too quickly, the authors train each head on a random sample from the minibatch. That is, for a minibatch of size 4, head 1 might train on transitions 1,3,4 and head 2 could train on transitions 2,3.

For the transitions a head doesn't train on, we instead train them against their own Q-values, so that nothing happens when passed to a stochastic optimizer. We do this in pursuit of a simplistic implementation, as every head trains on a constant batch size. Every other aspect of training is identical to regular DDQN with Experience Replay, including calculations of target and predicted values. The loss for each head is regular mean squared error.

IV. Our Implementation

We attempted to follow the original authors as closely as possibly, but due to the nature of the environments available to us versus the authors, we made select changes to the authors proposed algorithm. These changes were not spurred by happenchance, but based on other papers we had read such as [2] and [3].

A. Network Structure

For this paper, we aimed to avoid hard-coding every aspect of implementation; from network creating, to training, and environment setup. Every hyperparameter was accessible and easily set in the model creation stage. Our model was applicable to any type of observation space, as long as their action space was discrete. For each environment, our models had two hidden layers, each of which had a length of $2\times$ the number of actions.

In the original implementation, and other Bootstrapped DDQN implementations found online, there was a Core network present. We reasoned, however, that we did not need this. Each of these implementations was designed to be applied to environments where the observation space is not easily accessible, such as Atari games. For those environments, the core network functioned as a method to extract the game state from the screen. Because we focused

solely on OpenAI gym environments, we forgoed the core network altogether.

Next, each head network had a corresponding target network to which ER-DDQN could be used to train with. We tied these network pairs into an ensemble of tuples (head, target) so that we could easily access them in our implementation.

B. Training

For each epoch of training, we implement a training method highly reminiscent of traditional ER-DDQN. Our training method can be described in three main "blocks": Action Advising, Transition Collection, and Experience Replay.

1) Action Advising: Our implementation of RCMP is very closely related to the pseudocode described in Algorithm 1. At the suggestion of the authors, our final availability function was a simple budget. If you have budget left, you can ask for help, if not, you are out of luck. However, this is not the only function we tried. For example, you can run an environment for your demonstrator and learning model, and if the demonstrator and model are too far physically, you cannot ask for help, modeling real world scenarios. Another method is too implement a budget with cooldown; Say you ask for help, then you cant ask for n steps.

Once uncertainty is calculated, we have two options for an action. The first comes from the demonstrator, a StableBaselines-3 Proximal Policy Optimization (PPO) model for the same environment. The second, was our custom policy, based on [3], where we choose a single head to be the core network for an entire epoch. This core network decides our action in classic ϵ-greedy form. There are, of course, alternate ways to assign the default policy such as averaging the Q-values of the heads and selecting an action from that average. We tried this method as well and found no merit in it.

2) Transition Collection: The authors proposed transition collection to be identical to regular Experience Replay. Each experience is added to a replay buffer and a minibatch is sampled each training step. Unfortunately, as we will elaborate on later in this report, this method was one of many which resulted in critical failure of the algorithm and results nearly opposite to that of the authors.

Through many cycles of trial and error, we came upon our approach to transition collection. As a happy compromise between training every head on an advised action every step, and simple experience replay, we sample a minibatch with size one less than desired, and append our advised transition to this minibatch. Thus, since each head trains on a subset of this minibatch, some train on the advice and others don't. This approach gave us the most stable results.

3) Experience Replay: As described earlier in this report, we implemented a scalar form of the experience replay described by Algorithm 2. Our approach is best described in the below pseudocode:

Algorithm 3: Our Bootstrapped DDQN Experience Replay

```
Require miniBatch D, ensemble of h head-target pairs, head chance β
for i ∈ {1,...,num_heads} do
    targ ← zeros(len(minibatch),num_actions)
    states ← zeros(len(minibatch,num_obs)
    for (s,a,r,s',done) ∈ D do
        states[i] ← s
        targ[i] ← heads[i].predict(s)
        if random number > β then
            targ[i][a] ← r
            if not done then
                targ[i][a] += γ MaxQ(s,a)
            end
        end
    end
    Train heads[i] on states, targ batch
    if timestep mod stridelength = 0 then
        Copy heads[i] weights to targets[i]
    end
end
```

In the algorithm, each head decides whether to train on a particular transition purely by random numbers reaching or missing a threshold.

While algorithm 3 is portrayed in its final form, it is not the original design we had in mind. In fact, through numerous "learning moments", we arrived at this form. We will briefly explain the root cause of the algorithms problems. In the original paper [1], we were unable to determine the size, shape, and depth of the authors networks. Taking from our previous homeworks, we started this implementation on CartPole-v1, and thus decided our networks to have two hidden layers of size 25. Assuming the rest of the environments to be of similar complexity, we kept this size throughout our project.

C. Flaws

DQN is known to have several flaws; one such flaw presents itself early in the training cycle. DQN networks often choose a single action regardless of state, and this behavior needs to train out with time. DDQN usually solves this issue, but it seems that bootstrapped DDQN brings it back. We assume this is because not every head is trained on the full minibatch, so it is not as consistent as DDQN. Now, since advised actions are assumed to bring the agent closer to the goal and the actions which maximize reward, these actions should boost the Q-values of this state action pair significantly. Environments such as CartPole have minimal difference between two consecutive states. This presents quite the issue going forward: One head network acting with this single action prioritization behavior screws up the entire uncertainty calculation.

The Q-values for a 4-head ensemble on action 0 could look like this: $[0.0, 6.8, 7.2, 7.01]$ which has var$(\mathbf{Q}(s,a)) =$

12.28, whereas if we drop the first Q-value from this list, we get var($\mathbf{Q}(s, a)$) = 0.04003. It is clear to see that the Q-values are quite uniform except for the misbehaving network. Thus when you ask for help, the "good" networks raise their Q-values, and the "bad" networks would go from Q-values of $[0.0, 3.4]$ to $[0.0, 3.1]$. You can see how this raises the uncertainty when asking for help; the antithesis of this whole paper. This is the root cause of our pain. Asking for advice thus presents itself in the following cycle:

1) Ask for help
2) Bad networks cause uncertainty to rise
3) High uncertainty means ask for help
4) Go back to step 1

The moment our agent is uncertain, it will greedily use up all the advice it is allowed, then immediately forget everything it was advised on. For the sake of brevity we will not mention the many strategies we employed to mitigate this behavior. Instead, know that our implementation as according to Algorithm 3 stems from multiple discussions with the professor and is the best we found.

V. Results

Here we will speak to our implementation's results compared to normal DDQN, our changes that we attempted to do, and a discussion of these results and changes.

A. Frozen Lake

To start, we tested our algorithm on frozen lake. We chose frozen lake because it's a more advanced version of the basic Deep Q-Learning test, cliff walking. In frozen lake the agent is on a 4x4 grid, starting in the upper left. It's goal is to get to the bottom right while avoiding holes in the lake. The agent can take one of four actions, either up, down, left, right. The observation space is a number representing the index of a flattened 2D vector, the vector representing the 4x4 grid. The minimum value possible is 0, which is the upper left most location (the starting location). 15 is the lower right most location (the goal). Note that the agent has no knowledge of the holes, instead having to work it out itself.

For each timestep, a reward of 1 is given if the agent reaches the goal, else a reward of 0 is given. Finally, there is only a 1/3 chance that the agent actually goes to the location it was trying to go to. If it doesn't, it instead moves in a direction perpendicular to the one it was trying to go in. The idea is that the agent has to learn to take the safest route possible so even if it "slips" and moves in a wrong direction it's still fine.

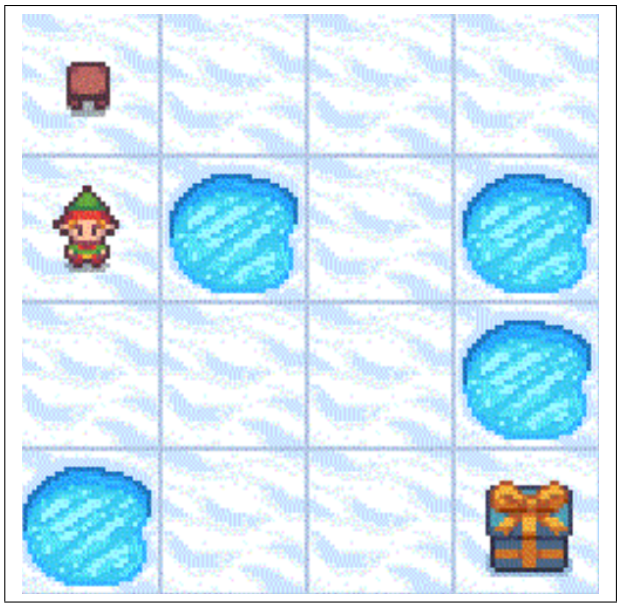

Fig. 1. The frozen lake environment. You can see the starting location at location 0 and the goal at location 15. There are holes at locations 5, 7, 11, and 12. The state of this would be 4, as that is where the agent is.

First, we did 3 runs with normal DDQN with experience replay so we can get a baseline. The parameters used are as follows: $\alpha = 0.01$, $\gamma = 0.8$, $\epsilon = 0.2$, stride length (the number of timesteps in between target network updates) = 50, memory limit = 2500, minibatch size = 8, number of epochs = 150.

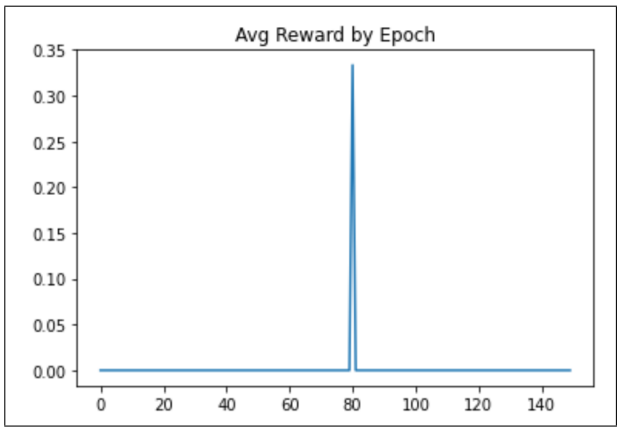

Fig. 2. The average reward per epoch over 3 runs of normal DDQN on the frozen lake environment.

As you can see in figure 2, the algorithm did not perform very well. Across all runs, it only ever had a single victory. Clearly, normal DDQN is not a very effective algorithm for this environment. Moving onto our algorithm, we used the same parameters as above, with the following added: We had 8 total heads, a help limit of 700 (although it was never fully used), and a 0.3 chance to remove a transition from the minibatch. Again, the demonstrator used is a successfully trained PPO (Proximal Policy Optimization) agent.

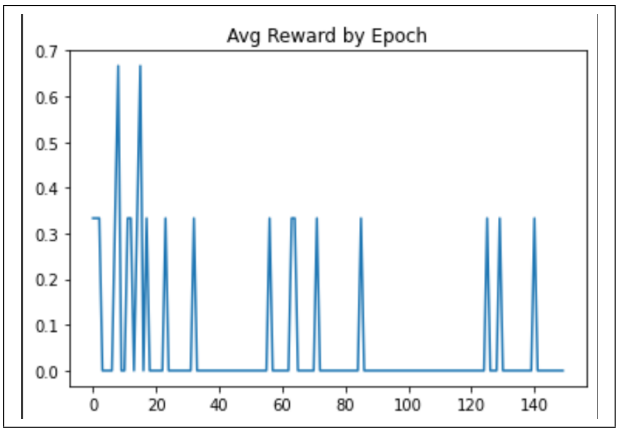

Fig. 3. The average reward per epoch over 3 runs of our algorithm on the frozen lake environment.

As you can see in figure 3, our algorithm significantly outperforms the normal DDQN algorithm. Ignoring the victories at the start, as these were likely caused by help from the demonstrator, we were able to get victories significantly more often. It's still not acting as a perfect policy would, however it's clearly significantly better.

In addition, looking at the uncertainty of one of the runs gives us our expected pattern:

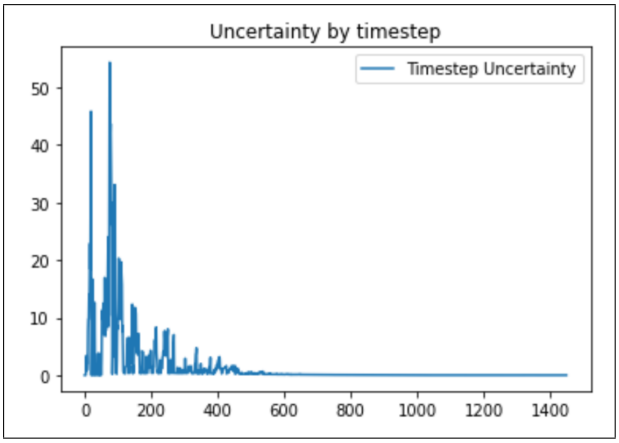

Fig. 4. The calculated uncertainty per timestep of the first run of our algorithm on the frozen lake environment. It follows the expected uncertainty pattern.

These results are in line with what we would expect. The normal DDQN was not very effective in frozen lake. While it's a similar environment to cliff walking, the optimal solution is incredibly hard to find due to the fact that the agent will only go where it selects 1/3 of the time. The only exploration policy that normal DDQN has is ϵ-greedy, which isn't very effective, so it has a hard time finding the solution.

One of the issues with this environment that we haven't spoken on yet is because the agent sees reward only once it's beaten the game and it's very unlikely to randomly stumble into the solution, algorithms tend to have a very hard time training on this environment. However, because our algorithm can ask for help, it's able to actually see reward and train on it correctly. Additionally, due to it's bootstrapped nature, it's exploration is better. These two factors allow it to beat the environment more often. As we see in figure 4, it starts to get very confident pretty quickly. However, our implementation clearly does not solve the environment. Our stable baselines 3 PPO agent was able to solve it more than 60% of the time. This goes to show that while the uncertainty-aware action advising helps, it's by no means an effective solution compared to the best algorithms we have at our disposal.

B. Cart Pole

Although our algorithm was largely inspired by previous homeworks of Cart Pole, we tested on Frozen Lake first because it did not have marginally different consecutive states. For us, Cart Pole was seen as the "final challenge" of sorts. If we were to overcome this hurdle, then we believed it to be easily translateable to other continuous and discrete observation space environments. To start, we show a graph of uncertainty that is representative of all non-final iterations of our algorithm.

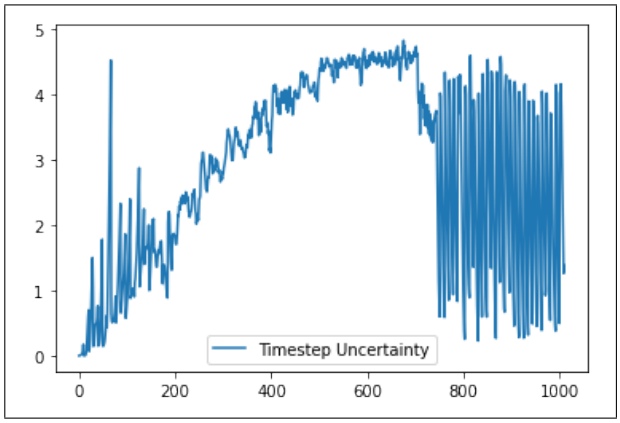

Fig. 5. Uncertainty: Generic Bad Implementation for CartPole. Advice Budget of 700.

Note in figure 5 that the agent asks for help in the first steps, and then uses it all in the next 700 timesteps. This coincides with the massive rise in uncertainty, peaking at 5. After 700 steps, the uncertainty slowly recedes back down as normal training ensues.

Comparing this to a graph for our final implementation, you see much more stable uncertainty.

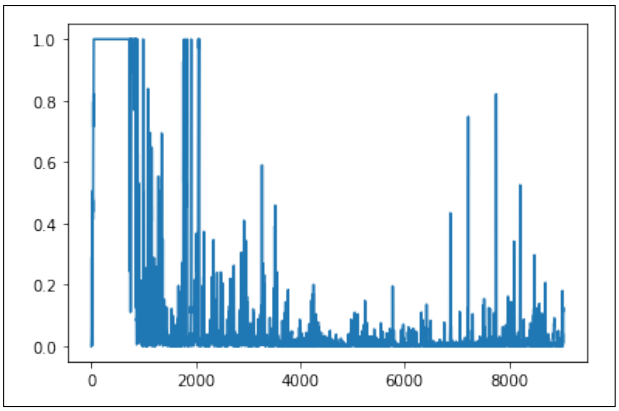

Fig. 6. Uncertainty: Final implementation for CartPole. Advice Budget of 700 and uncertainty capped at 1.

In figure 6, we capped uncertainty to 1.0, and you see that all 700 budget was used up from the beginning. Contrary to results from 5, uncertainty quickly dropped below 1.0, and then the threshold for advice of 0.5. Of course, these results mean nothing if we have no reward data to compare with. Beware that our reward data is particularly un-motivating. The above figure was cut off at 8000 timesteps; in actuality, we continued on for another 30,000 or so timesteps keeping the pattern you see in the figure.

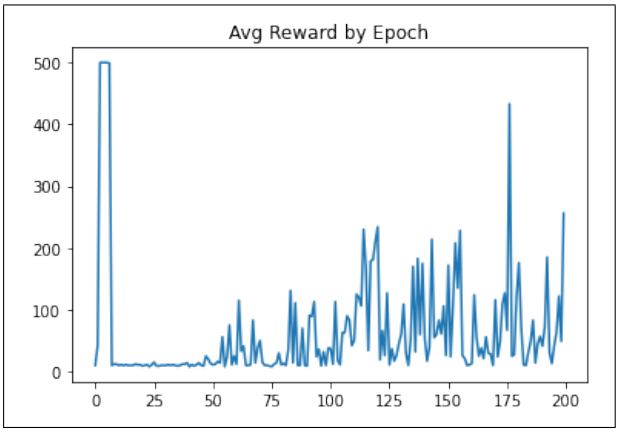

Fig. 7. Reward: Same run as Figure 6

As expected of a head with bad behavior, you can see that the advice budget was wholly consumed in the first 5 epochs. As described earlier in this report, the agent completely disregards the advice it was given for a period of time, and then proceeds training in a manner indistinguishable from normal Bootstrapped-DDQN. However, We can observe one upside. Both DQN and its derivatives struggle immensely with CartPole, training to a epoch reward average of ≈ 200.0 then falling victim to over-training and regressing in epoch reward. Both our implementation and Bootstrapped-DDQN minimize this attribute of DDQN. The max average epoch reward is higher, ranging in the $250.0 - 300.0$ range instead of 200.0, and regression is much delayed. This allows for more forgiveness in training times, which can be beneficial to those who do not have infinite time to optimize over their hyperparameters.

It is inevitable to conclude our algorithm performs worse than what we had hoped for for CartPole. That being said, through numerous revisions we have managed to bring it a level which is equal, if not better, than its predecessors such as DDQN and Bootstrapped-DDQN. In its current state, we can apply our algorithm confidently; that it will meet baselines testings, and sometimes surpass them. Thus we are content with the level it has reached, even if it has fallen short of the bar set by the original paper.

VI. Conclusion

In this section, we present a recap of our most important discoveries and the implementation, as well as a discussion on the potential applications to other learning architectures.

A. Summary

As we've seen, the uncertainty-aware action advising model for DDQN does outperform normal DDQN. This is likely because the model has a few notable advantages. For example, the bootstrapped nature leads to a better exploration policy. Additionally, being able to ask for advice leads to more reward being given at the start of training, making training more effective. However, the difference in results is unfortunately relatively marginal.

Unfortunately, one of the major downsides of adding uncertainty-aware action advising is the computational time loss. This algorithm takes significantly more time, primarily due to the bootstrapping. Each head gives two more networks that need to be trained each timestep. This means that for an implementation with 8 heads, that's 16 more networks. For example, in the frozen lake environment this lead to just under a 500% time increase.

The other notable downside is the limited effective application of uncertainty aware action advising to environments. Any environment comprised of marginally different consecutive states will be difficult to train on, and will often require fine-tuning or even complete modification to the original algorithm. Due to these downsides, we unfortunately don't believe that uncertainty-aware action advising will be helpful for DDQN-based algorithms.

B. Future Work

Taking Q-Learning at its base level, a value function approximator, the uncertainty-based architecture could be added to any other model which uses value functions for their approximators. It would be interesting to see this applied to Actor-Critic methods. For example, Proximal Policy Optimisation (PPO), which is widely regarded as one of the current best policy gradient methods, is often implemented in Actor-Critic form. In this form, the critic gives a critique of the Actor networks choice of action. Thus it could be quite useful to know how confident the critic in its critique of the Actor. This was the main implementation we thought to apply bootstrapping and network uncertainty towards. Yet, we be live there to be many applications which we have not even thought to consider. We hope that more research happens in this area in the future, as this method of learning more closely imitates human learning.

C. Post Scriptum

Our complete implementation can be found as a Google Colab Notebook. Thank you to our Professor, Dr. Mei Si for her continual advice and suggestions throughout the progression of our project.

REFERENCES

[1] F. L. Da Silva, P. Hernandez-Leal, B. Kartal, and M. E. Taylor, "Uncertainty-aware action advising for deep reinforcement learning agents," Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, no. 04, pp. 5792–5799, 2020.

[2] T. Tan, T. Chu and J. Wang, "Multi-Agent Bootstrapped Deep Q-Network for Large-Scale Traffic Signal Control," 2020 IEEE Conference on Control Technology and Applications (CCTA), 2020, pp. 358-365, doi: 10.1109/CCTA41146.2020.9206275.

[3] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. 2016. Deep exploration via bootstrapped DQN. In Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16). Curran Associates Inc., Red Hook, NY, USA, 4033–4041.