Blackjackapprenticeship.com. (2019). [online] Available at: https://www.blackjackapprenticeship.com/blackjack-strategy-charts/ [Accessed 20 Oct. 2019].

Deep Q-Learning for Blackjack

ID 58, Student ID: 27824438

1 Deep Q-Learning

1.1 Q-Learning Algorithm

Q-Learning is a type of reinforcement learning algorithm based upon the Markov decision process (MDP). The MDP "provides a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the control of a decision maker." [1] The basis for MDP is that within an environment (blackjack), an agent (the model) can take actions (hit or stand) based on the current state (the player's hand and the dealer's showing card). The MDP can have an optimal policy, being the best action to take for every given state to obtain the highest reward (winning the round).

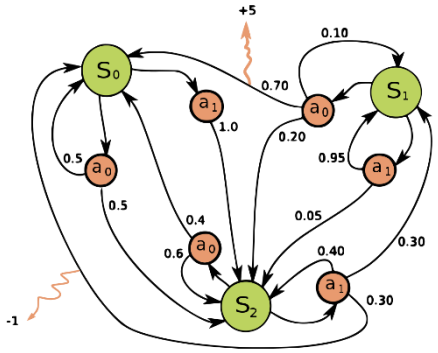


Figure 1 Markov Decision Process [1]

Q-learning is a reinforcement learning (RL) algorithm that aims to discover the optimal policy for a given environment. As the outcome for an action is partly random (if we stand when holding a 20, there's still a chance the dealer could hit 21 and win), this randomness must be incorporated to calculate a probability of reaching another state (and so reward) for each action. As a reward may only come after a series of actions, the importance of a higher delayed reward signal must also be taken into consideration to avoid local optima. The Q-Value is defined as the current reward for an action *a,* taken at state *s*, plus gamma (discount factor for future rewards) of the highest reward for the next state.

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

[2]

As the reward of the next start depends on the reward of the following state recursively, the gamma value is to the power of the amount of actions taken so that the importance of the future rewards decay the further they are in the future.

$$Q(s,a) \rightarrow \gamma Q(s',a) + \gamma^2 Q(s'',a) \dots \dots \dots \gamma^n Q(s''^{\dots n},a)$$

The full Q-learning equation is therefore:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\right]$$

Where random Q-values are initially chosen for all state/actions. As the agent explores the environment, it updates the Q-value for each state using the difference between its original Q-value and the realised Q-value. Alpha is the learning rate for how much of this difference is applied when updating. This algorithm continue until it has a confident Q-table (a Q-value for each action for all states).

The weakness to this algorithm arises when applying it to complicated environments (e.g. playing many arcade games) as it would have to encounter every single state. If your state is based on image data, there can be a very

large state space. Additionally, the algorith is not able to generalise so it will make a random decision for any states it has not previously encountered.

1.2 Deep Q-Learning Algorithm

Many of these weaknesses are improved by Deep Q-Networks (DQN). Instead of generating a Q-Table, a linear deep network is used to approximate the Q-value function. This then becomes a regression problem where the model learns to approximate the Q-value function. This is done using the the mean squared error loss between the models predicition for a Q-value of each state, compared to the Q-value that is realised when a predicted action is taken. DQNs are therefore able to recognise patterns and generalise significantly better than Q-learning for complex enviornments.

2 Blackjack Environment

From Open AI's Gym Blackjack environment class docstring:

"Blackjack is a card game where the goal is to obtain cards that sum to as near as possible to 21 without going over. They're playing against a fixed dealer.

Face cards (Jack, Queen, King) have point value 10. An Aces can either count as 11 or 1, and it's called 'usable' at 11. This game is placed with an infinite deck (or with replacement). The game starts with each (player and dealer) having one face up and one face down card.

The player can request additional cards (hit=1) until they decide to stop (stick=0) or exceed 21 (bust). After the player sticks, the dealer reveals their facedown card, and draws until their sum is 17 or greater. If the dealer goes bust the player wins. If neither player nor dealer busts, the outcome (win, lose, draw) is decided by whose sum is closer to 21. The reward for winning is +1, drawing is 0, and losing is -1.

The observation of a 3-tuple of: the players current sum, the dealer's one showing card (1-10 where 1 is ace), and whether or not the player holds a usable ace (0 or 1)."

3 IMPLEMENTATION

Experience Replay:

```
class Memory():
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, cur_state, action, next_state, reward, done):
        """Add the state, action and consequential state to memory as Tensors"""
        state_trans = [cur_state_t, action_t, rewart_t, next_state_t, done_t]
        if len(self.memory) < self.capacity:
            self.memory.append(state_trans)
        else:
            self.memory[self.position] = state_trans
            self.position = (self.position + 1) % self.capacity

    def sample(self, batchsize=10):
        """Return tensors needed to train model"""
        minibatch = random.sample(self.memory, batchsize)
        ...
        return state_batch, action_batch, reward_batch, state_1_batch, done_batch
```

A replay buffer is used to store current state, action, future state, reward and if the action resulted in the round finishing. After completing an episode by playing a set number of rounds with the model remaining constant, the model is updated based on a random sample of the memory. This allows mini batching to be utilised instead of updating each step which would make it difficult for the model to converge. This random sampling also avoids learning from subsequent experiences which highly correlate.

Updating the network:

```
current_state_qs = model(state_batch)
for i in range(MINIBATCH_SIZE):
    # If the Action didn't cause it to finish, the max q is given from the next q values * gamma discount
    if not done_batch[i]:
        max_future_q = torch.max(next_state_qs[i])
        new_q = reward_batch[i] + GAMMA * max_future_q
    # If final state reached, the max q of the action taken is the reward value received
    else:
        new_q = reward_batch[i]

    # Update the target qs for with the new_q value for the action taken
    action_taken = int(action_batch[i].item())
    current_qs = current_state_qs[i].clone()
    current_qs[action_taken] = new_q
    target_qs_list.append(current_qs)

target_q_values = torch.stack(target_qs_list)
loss = loss_func(current_state_qs, target_q_values)
```

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a)$$

The loss for our model is MSE between the q values the model predicts, and a target set of q values for the same states. The target qs are based on the realised or future reward, depending if the action resulted in a final state (standing or busting). As we can only update the q value for the action that was taken, the target q value for the action not taken is left as the same value as the models prediction so there is no loss.

Epsilon-Greedy exploration:

```
def get_action(model, cur_state, epsilon):
    """Output an action from either the recomendation from the NN
or a random choice, depending on epsilon."""
    if np.random.rand() < epsilon:
        return np.random.randint(0, NUMBER_OF_ACTIONS)
    else:
        predicted_action = model(cur_state)
        values, index = predicted_action.max(0)
        return index.item()
```

To ensure that the model initially explores the state space, the epsilon-greedy policy is used. Initially, epsilon is high and so the agent takes predominately random actions allowing the model to encounter a variety of the state space. As training progresses, epsilon decreases so that action taken are predominantly the models learnt predictions which can then be verified and improved. The intent is that it will decrease the chance of the model learning a local optimum early in the training.

4 RESULTS

Blackjack was chosen for this project as there is an optimal action for each state [4], which makes it easy to compare the output of the model to this ideal policy and evaluate how effectively the model is learning. A soft hand is defined as containing an Ace that can either be a one or eleven.

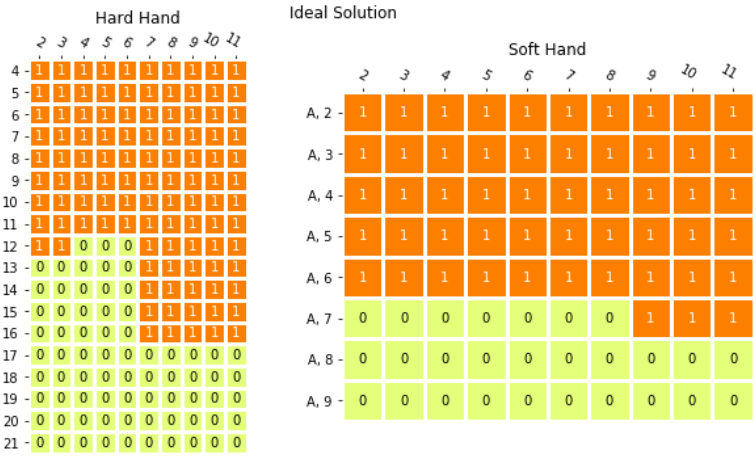


Figure 2 Ideal Policy

A properly trained DQN was effective in learning the ideal policy, with the model's policy matching the ideal by 91.11% for hard hands and 82.5% for soft hands. After allowing the trained model to play 100,000 rounds of blackjack, the model had the following round outcomes:

```
Win: 42.21%, Lose: 49.14%, Draw: 8.65%
```

If the optimal policy is used in a cut card game, the ideal event probability is **Win: 42.43%, Lose: 49.09% and Draw: 8.48%** [5]. This shows that the model was able to approximate a Q function resulting in an outcome that's almost identical to the ideal policy. This shows that for the states where it couldn't match the ideal policy, the difference between the ideal and the model's policy has a marginal impact.

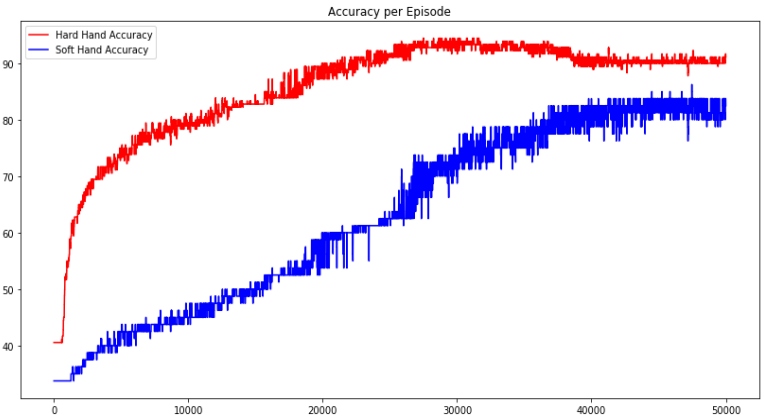


Figure 4 Model Accuracy Compared to Ideal Policy

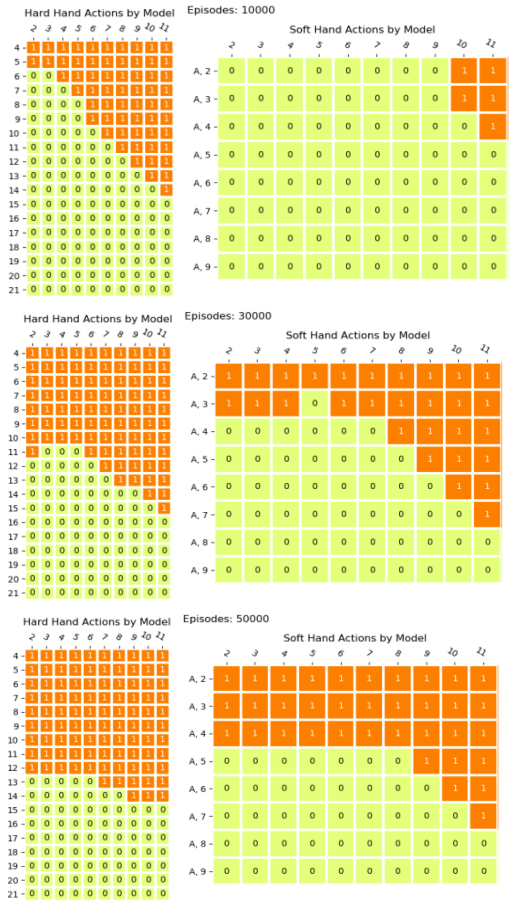


Figure 3 Model's Actions while Training

5 OPTIMISATIONS

5.1 WEIGHT NORMALIZATION

Some form of normalization can often be employed to assists a networks ability learn. From the paper that introduced weight normalization, DQNs are "an application for which batch normalization is not well suited: the noise introduced by estimating the minibatch statistics destabilizes the learning process" [6]. Weight normalization is the process of splitting a layer's weights into learnable scalar and vector component such that:

$$weight = \frac{scalar * vector}{\|vector\|}$$

This change is supposed to assist with optimization for certain networks (DQNs included). It also has the advantage of being less computationally expensive than batch normalization. The results show that it does indeed have an impact on both training speed and accuracy, particular for soft hand accuracy which networks find harder to learn.

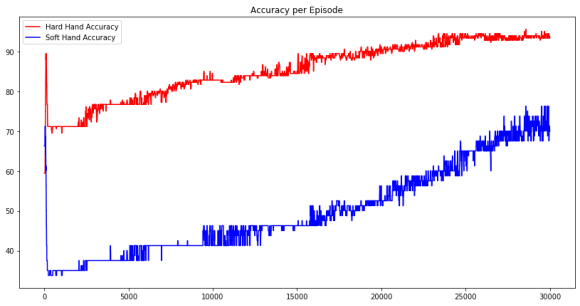


Figure 5 No Weight Normalization

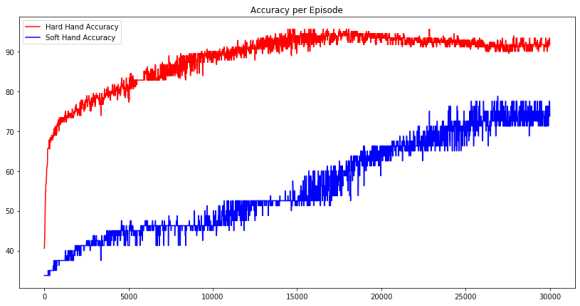


Figure 6 Weight Normalization on all Linear Layers

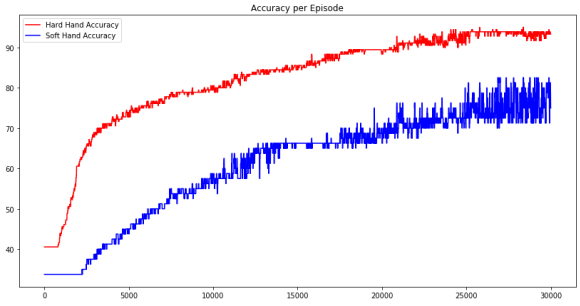


Figure 7 Weight Normalization on all but Last Layer

5.2 Network Size

Several network sizes were experimented with to try and maximise accuracy. It seems that hard hand accuracy is unsurprisingly easy to learn, even for a simple network. This may be due to the stronger correlation between the action and reward with a hard hand. For example, if the player has a 17 hand, there is an extremely high probability the player will lose if they hit, regardless of the dealer's hand. With a soft hand, the detrimental effects of hitting are dampened. If a soft hand hits at 17, there is no instant negative reward as the hand cannot bust (the ace reverts to a one). This means training must rely on the discounted future reward which increases difficulty. The final architecture as demonstrated in section 5.1 was (3 > 96), (96 > 48), (48 > 24), (24 > 2).

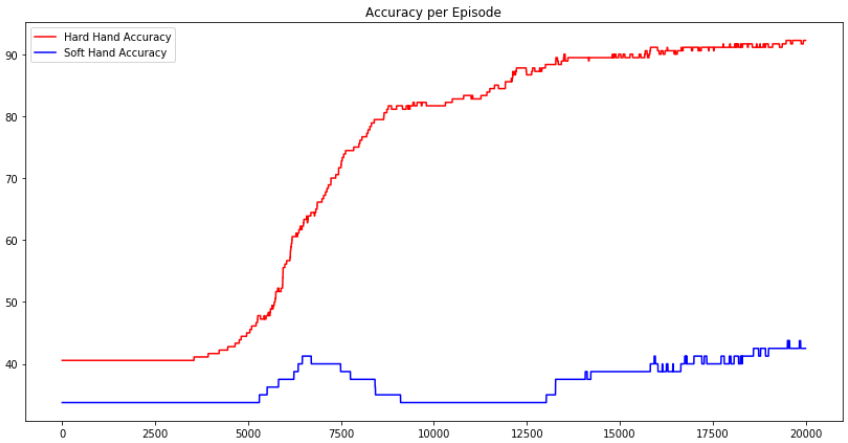


Figure 8 Linear (3 > 12), (12 > 2)

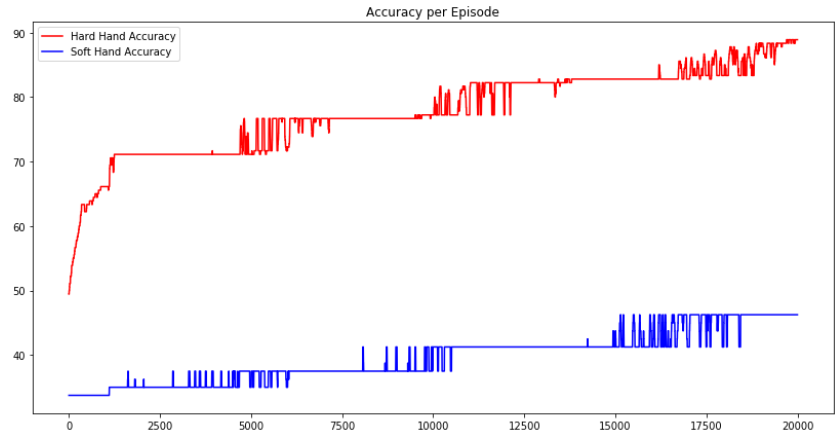


Figure 9 Linear (3 > 30), (30 > 30), (30 > 2)

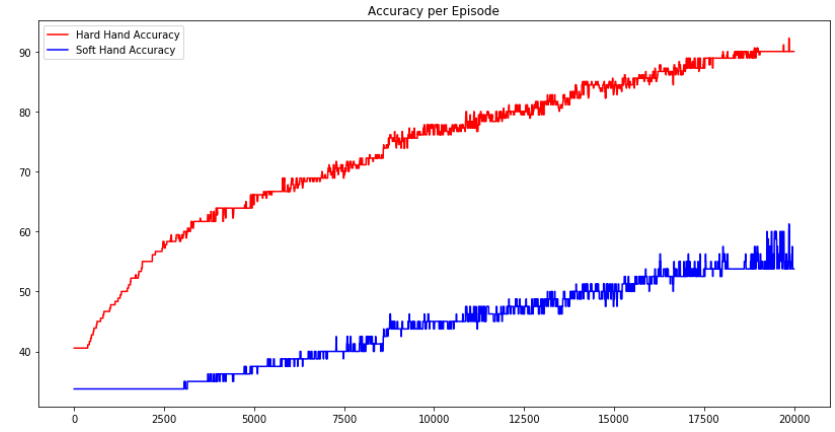


Figure 10 Linear (3 > 60), (60 > 20), (20 > 2)

5.3 EPSILON DECAY

For all networks explored so far, epsilon has decayed from 1.00 to 0.01 linearly over the duration of episodes the model was trained. Typically, greedy-epsilon is implemented with an exponential decay rate so that the state space is explored initially before converging towards a policy that deems the highest reward. Interestingly as shown below, an exponential decay improves the speed at which hard hand accuracy initially learns. However, this comes at the large cost of severely reducing the ability of soft hand accuracy to learn. The importance of a reasonable epsilon at later episodes is highlighted by Figure 13, where Epsilon remains constant at 0.05 and beats the exponential decay networks by still having soft accuracy trending upwards at 20,000 episodes.

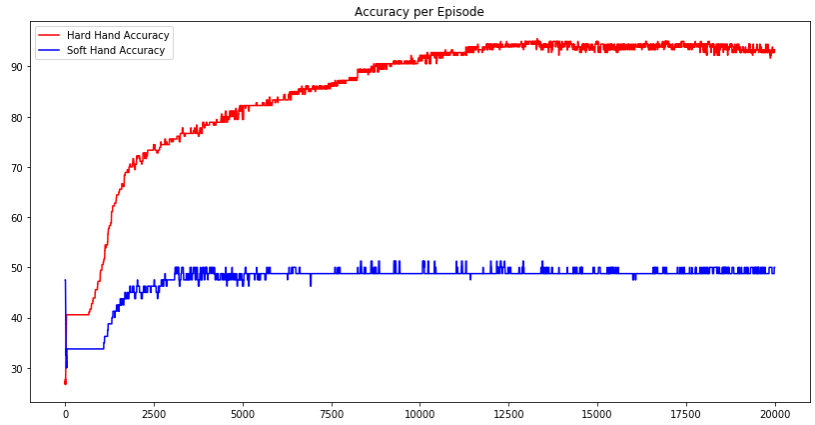


Figure 11 Exponential Epsilon Decay at 0.999 per Episode

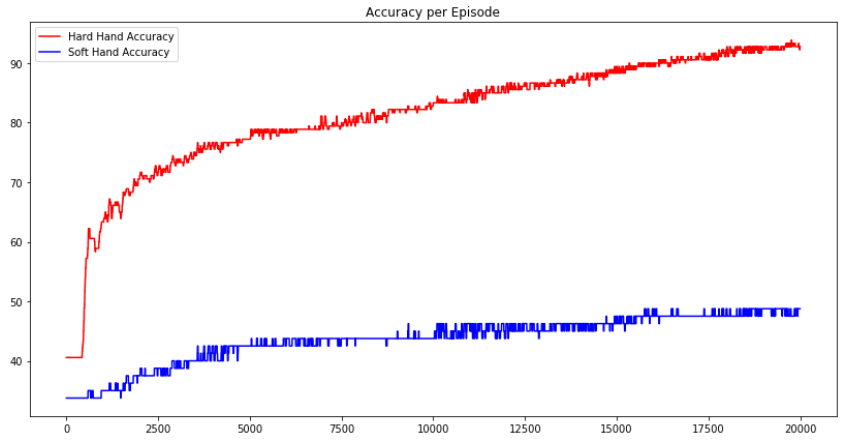


Figure 12 Exponential Epsilon Decaying lower to 0.001

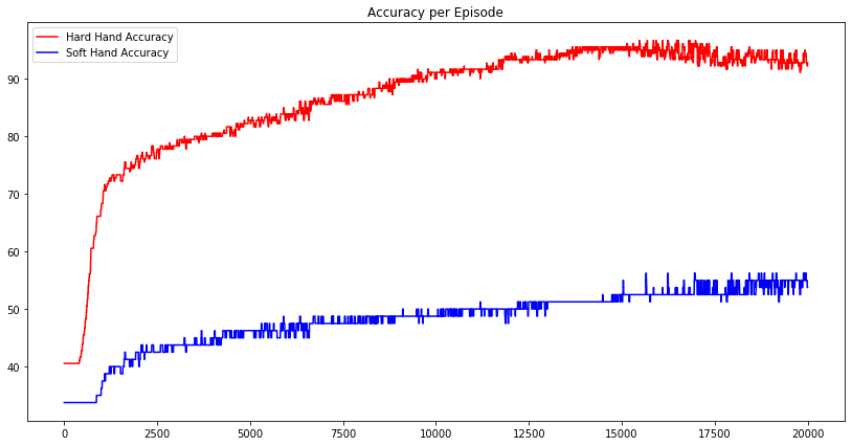


Figure 13 Epsilon Constant at 0.05

5.4 MINIBATCH SIZE

All other networks are trained with a minibatch size of 256. This is the number of samples randomly chosen form the experience memory and then fed through the network so that the network can be improved from the loss function. Halving the size of the minibatch severely reduces the time it takes to train the network (~30% slower to reach the same hard accuracy). Even letting the model train for an extremely large number of episodes, it would seem the network converges to a less accurate result.

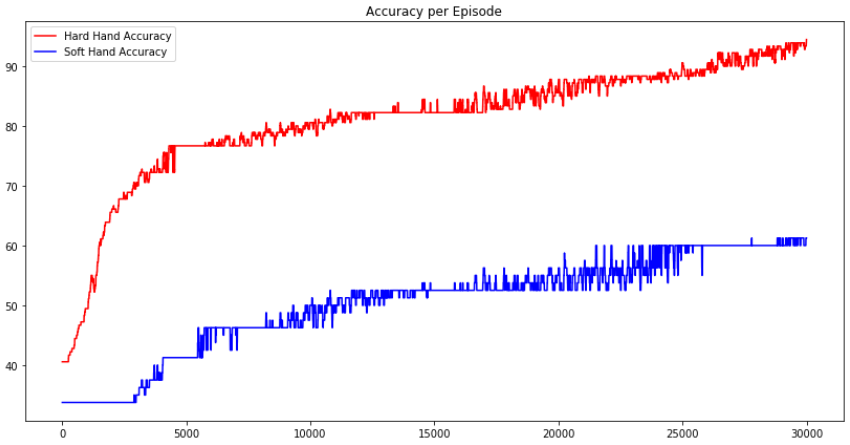


Figure 14 Minibatch of 128

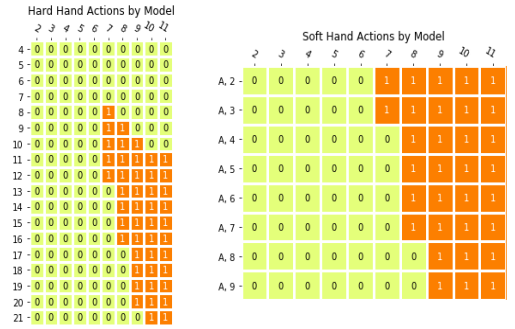
EPISODE NUMBER	HARD HAND ACCURACY	SOFT HAND ACCURACY
EPISODE: 1000	Hard Acc: 52.22	Soft Acc: 33.75
EPISODE: 2000	Hard Acc: 63.89	Soft Acc: 33.75
EPISODE: 10000	Hard Acc: 75.00	Soft Acc: 40.00
EPISODE: 20000	Hard Acc: 83.89	Soft Acc: 51.25
EPISODE: 30000	Hard Acc: 90.00	Soft Acc: 60.00
EPISODE: 40000	Hard Acc: 92.78	Soft Acc: 73.75
EPISODE: 50000	Hard Acc: 91.67	Soft Acc: 78.75
EPISODE: 60000	Hard Acc: 90.00	Soft Acc: 80.00
EPISODE: 70000	Hard Acc: 89.44	Soft Acc: 80.00

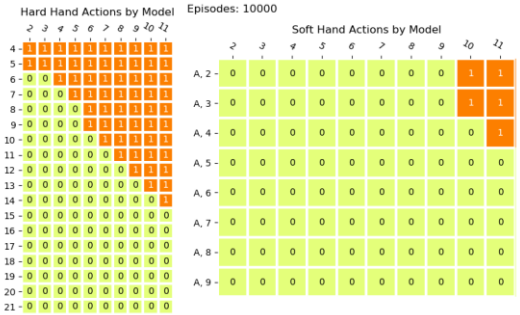
6 CONCLUSION

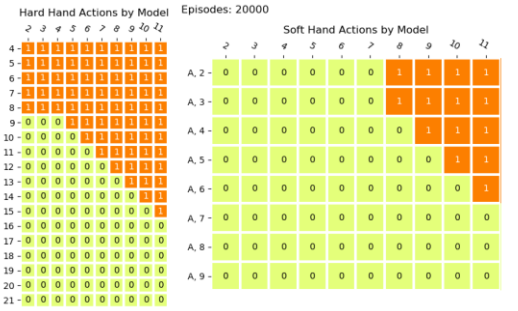
A DQN was successfully able to learn an almost optimal strategy for blackjack. The best results were using reasonably large network with weight normalization, linear epsilon decay to 0.01, a 256 minibatch size and with around 50,000 training episodes. It would be difficult to further improve the accuracy performance, though further hyperparameter tunning would highly likely yield even better results for the same model.

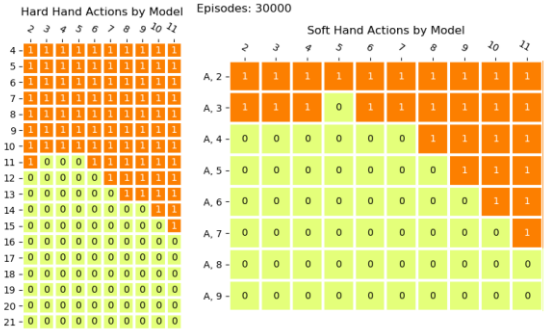
7 APPENDIX

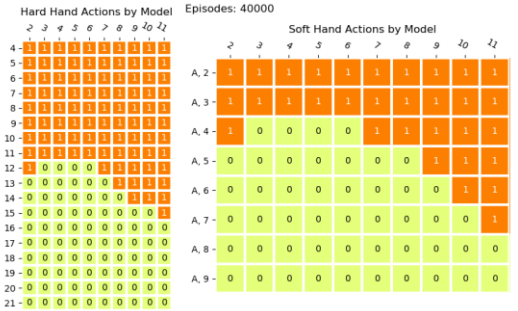
Visualization of the model training using the network as discussed in section 5.4.

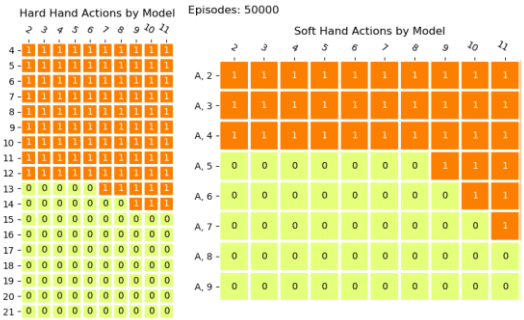


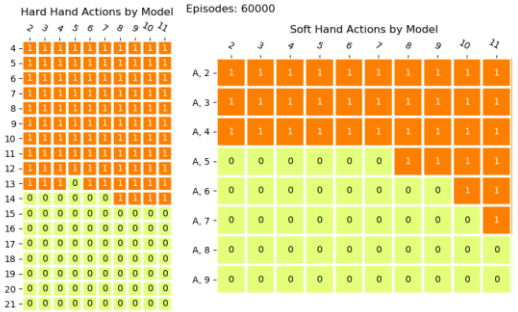


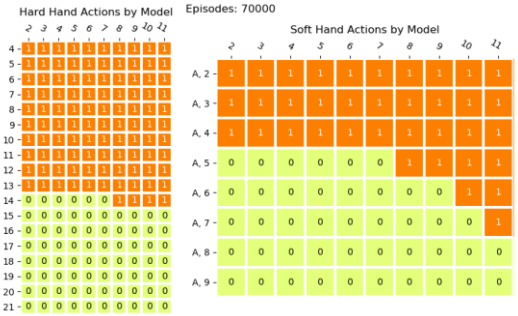












8 REFERENCES

[1] En.wikipedia.org. (2019). Markov decision process. [online] Available at: https://en.wikipedia.org/wiki/Markov_decision_process [Accessed 20 Oct. 2019].

[2] Python, A. (2019). Introduction to Deep Q-Learning for Reinforcement Learning (in Python). [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/ [Accessed 20 Oct. 2019].

[3] Mao, H. (2019). Resource Management with Deep Reinforcement Learning. Massachusetts Institute of Technology? , Microsoft Research. Available at: http://people.csail.mit.edu/hongzi/content/publications/DeepRM-HotNets16.pdf [Accessed 20 Oct. 2019].

[4] Blackjackapprenticeship.com. (2019). [online] Available at: https://www.blackjackapprenticeship.com/blackjack-strategy-charts/ [Accessed 20 Oct. 2019].

[5] Blackjackapprenticeship.com. (2019). [online] Available at: https://www.blackjackapprenticeship.com/blackjack-strategy-charts/ [Accessed 20 Oct. 2019].

[6] Blackjackapprenticeship.com. (2019). [online] Available at: https://www.blackjackapprenticeship.com/blackjack-strategy-charts/ [Accessed 20 Oct. 2019].