Advanced Topics in Reinforcement Learning (048716) Wet Exercise

Eyal Ben-David
Ron Dorfman

December 2018

1 Solving the Taxi Environment

1.1 Environment Description

The taxi environment is comprised of a 5×5 grid world with 4 designated locations. When an episode starts, the taxi starts off at a random square and the passenger is at a random location (one of the designated four). The taxi drive to the passenger's location, pick up the passenger, drive to the passenger's destination (another one of the four specified locations), and then drop off the passenger. Once the passenger is dropped off, the episode ends.

In this environment, there are 6 possible actions:

$$\mathcal{A} = \{\textit{'south'}, \textit{'north'}, \textit{'east'}, \textit{'west'}, \textit{'pickup'}, \textit{'dropoff'}\}.$$

Thus, the action space is $|\mathcal{A}| = 6$.

There is a reward of -1 for each action and an additional reward of +20 for delievering the passenger. There is a reward of -10 for executing actions "pickup" and "dropoff" illegally. We limit each episode to be 200 transitions at most, meaning, if the passenger doesn't dropped off in its destination in 200 environment transitions, the episode ends.

There are 500 discrete states since there are 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is the taxi), and 4 destination locations $\Rightarrow$ The state space is $|\mathcal{S}| = 500$.

1.2 Fully Connected DQN Architecture

1.2.1 Q-learning

The goal of *Q-learning* (Watkins, 1989) algorithms is to learn the action-value function, Q, of some policy π, which is defined as the expected return starting from state s,

taking the action a, and thereafter following policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[R_t|s_t = s, a_t = a] = \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a]$$

We know that for deterministic MDP's, the following Bellman optimality equation holds:

$$Q^*(s_t, a_t) = r(s_t, a_t) + \gamma \max_{a' \in \mathcal{A}} Q^*(s_{t+1}, a')$$

where Q^* is the optimal action-value function.

1.2.2 Deep Q-learning

In *Deep Q-learning* we aim to learn the Q function using a deep neural network. In the taxi environment, the action space is discrete, thus we will train a network that gets the state s of the environment as input, and outputs the Q values for each action in the action space.

We estimate the Q function using a fully connected network with two layers:

$$Q_{\theta}(s,:) = \sigma_2(W_1\sigma_1(W_2 s + b_2) + b_1)$$

where θ is the set of all the network's parameters (weights and biases), the inner activation is $\sigma_1(\boldsymbol{x}) = ReLU(\boldsymbol{x})$ and the outer activation is linear, $\sigma_2(\boldsymbol{x}) = \boldsymbol{x}$. Moreover, s is some vector encoding of the environment's state. We use a *one-hot* encoding for the states, meaning, we encode the i-th state as $\mathbf{e}_i \in \mathbb{R}^{500}$ where

$$(\mathbf{e}_i)_j = \begin{cases} 1, & j = i \\ 0, & j \neq i \end{cases}$$

Thus, the input size of the network is 500. Note that $Q_{\theta}(s,:)$ is the estimated vector of Q values for every action, therefore the output size is 6. The size of the middle layer is not specified and we chose it to be 32. The justification for this specific choice is presented below.

We choose the first

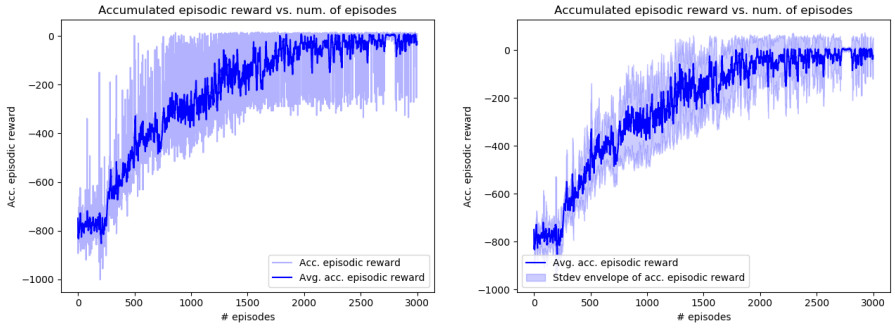
1.2.3 Optimizing the Network

Since we know that the optimal action-value function satisfies Bellman optimality equation, we would like to solve:

$$\min_{\theta} (r(s_t, a_t) + \gamma \max_{a' \in \mathcal{A}} Q_{\theta}(s_{t+1}, a') - Q_{\theta}(s_t, a_t))^2$$

To stabilize the learning procedure, we use a target network that is updated in much slower time-scale, resulting in the following minimization problem:

$$\min_{\theta} (r(s_t, a_t) + \gamma \max_{a' \in \mathcal{A}} Q_{\theta_{target}}(s_{t+1}, a') - Q_{\theta}(s_t, a_t))^2 \tag{1}$$



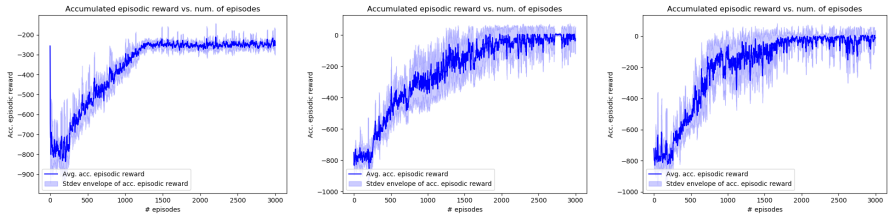
(a) (b)

Figure 1: Convergence plots of the DQN algorithm with hidden layer size of 32. (a) The accumulated episodic reward and its average over the most recent 10 episodes vs. number of episodes, (b) Average accumulated episodic reward over the most recent 10 episodes and its standard deviation vs. number of episodes.

Furthermore, we clip the Bellman error (the term we square in the minimization objective) to $[-1, 1]$, meaning every error that is above/below the value $1/-1$ is set to $1/-1$, respectively, to prevent a gradient explosion during training.
In online *Q-learning*, sequential states are strongly correlated. In order to break those correlations we use an *Experience Replay* as proposed in the original DQN paper (Mnih *et al.*, 2013): we collect transitions (s_i, a_i, r_i, s_i') by running an ϵ-greedy policy on the environment, store them in a buffer and then sample batches of transitions with which we learn the Q function by minimizing (1) (or a version of it) with some gradient-based method.

We used the RMSProp minimizer with minibatches of size 32. The behavior policy during training was ϵ-greedy with ϵ annealed linearly from 1 to 0.1 over the first 350000 transitions, and fixed at 0.1 thereafter. The replay buffer memory was set to 200000 most recent states. We don't use any special regularizations. The learning starts after 50000 transitions and the target network is updated every 10000 transitions. We limit the number of episodes to 3000 and then the algorithm halts.

The convergence results are presented in Fig. 1. In Fig. 1a we can see the accumulated reward in an episode vs. the episode number, and on its top the average episodic reward over the last 10 episodes is presented. We can see that the reward increases over time, starting at around -800 and begin increasing after approximately 250 episodes (each episode at the beginning is 200 transitions and we start learning after 50000 transitions), till it reaches also positive average episodic rewards. Fig. 1b illustrates the standard deviation of the learning process, where the average episodic reward over the most recent 10 episodes is displayed along with its standard deviation as its "sleeve".



(a) (b) (c)

Figure 2: Convergence plots of the DQN algorithm for different middle neuron layer dimension. (a), (b) and (c) presents the average accumulated reward and its standard deviation for hidden layer size of 4, 32 and 256, respectively.

1.2.4 Middle Neuron Layer Size Selection

In the following section we justify our choice of middle neuron layer dimension by trying multiple different dimensions and comparing their convergence results. We tried the following 3 different dimensions: 4, 32 and 256. The convergence results for all the 3 architectures are displayed in Fig. 2, and we display the average accumulated episodic reward signal of all the 3 architectures on the same figure in Fig. 3. We can conclude that the first architecture, which has 4 hidden neurons is too small since it doesn't converge to a policy that achieves positive accumulated rewards, and this is due too the network being not rich enough. In other words, the network is not expressive enough to estimate a good Q function. Another conclusion we can derive is that the two other architecture converges to some good policies (alternatively, learn some good estimation of the Q function) with small difference in learning curves. As a result, we choose the smaller network since it has a faster training procedure and it saves memory.

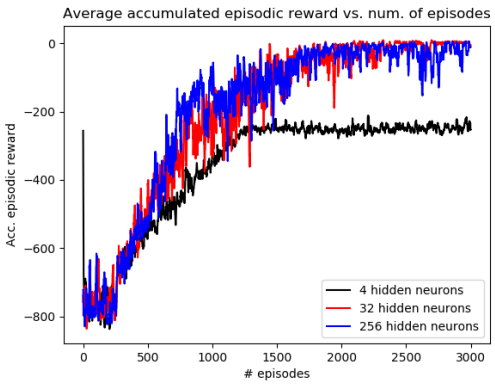


Figure 3: Average accumulated episodic reward against number of episodes for different hidden layer dimensions.

In Table 1 we can see the total learning time, which is the time took to finish 3000 episodes and best average accumulated episodic reward. The smallest architecture takes longer to train because it doesn't reach to a good policy, thus each episode takes longer. The largest architecture takes longer to train compared to the 32 hidden neurons architecture because it has more weights, therefore the backpropagation is slower. The difference in the best average accumulated episodic reward is probably due to randomness and is not significant.

Num. hidden neurons	Total learning time [s]	Best average acc. reward
4	417.51	-211.2
32	**260.05**	**8.1**
256	338.82	6.6

Table 1: Comparison of learning time and best accumulated episodic reward between the 3 different hidden layer dimensions

1.2.5 Regularization

In this section, we check how regularizion of the network's weights affects the results. We use two regularizations: *dropout* and L_1 regularizarion. The learning curves are presented in Fig. 4, where we compare the results for using *dropout* or L_1 and without using any regularization. We can learn from the empirical results that using a regularization is not helping and in fact is harmful. As we can see, after 3000 episodes the regularized models are significantly worse. Whether the convergence is slower or results in some bad local minimum, not applying regularization performs better. The reason for that is we are trying to learn the Q function and unlike different classification tasks (e.g. ImageNet), we are trying to get an overfit to the training data in order that our estimation would be exact and the derived policy would be optimal.

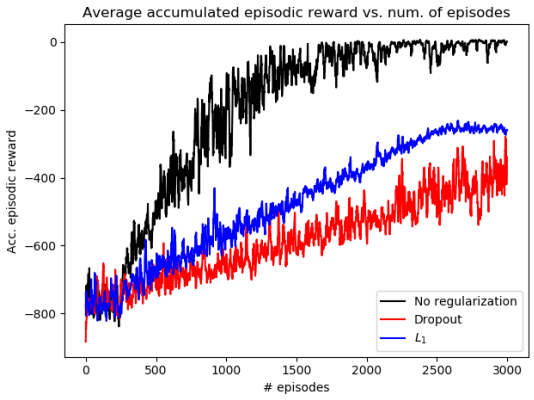
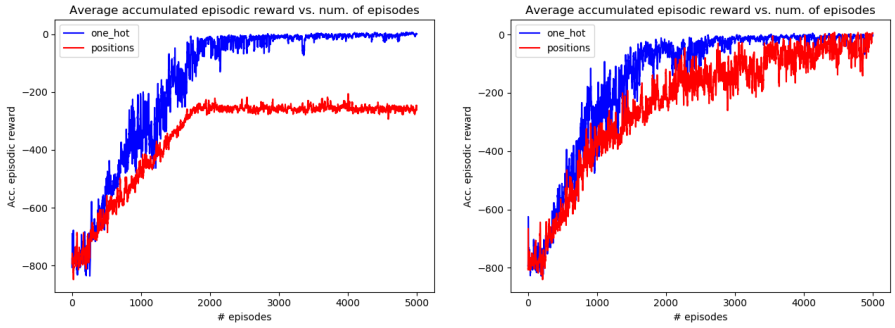


Figure 4: Convergence comparison between different regularizations: Dropout, L_1 and without any regularization.



(a) (b)

Figure 5: Learning curves comparison between two encoding methods: 'one-hot' and 'position' for two different middle neuron layers 32 and 128 in (a) and (b), respectively.

1.2.6 Different Input Encoding

So far, we used *one-hot* encoding of the states. Now, we present another state encoding and compare the later method with the previous one. We suggest the following encoding scheme, based on the understanding of what is the intrinsic data in the environment's states. We know that the state is fully determined by the taxi's position, the passenger's position and the destination location. We represent the taxi's position by its row and column in the 5×5 grid, there are 5 options to every one of them. We represent the passenger's position as one out of five options, each of the possible locations in case that the taxi didn't pick him up yet and the fifth option is that the passenger is in the taxi. Finally, we represent the destination location as one out of four possible locations. Each of the taxi's row, taxi's column, passenger's position and the destination location we encode as a *one-hot* vector. We need 5-dimensional vectors for the taxi's row, the taxi's column and the passenger's position and another 4-dimensional vector for the destination location. We then concatenate them for a total of 19-dimensional vector to represent the environment's state and term this encoding by *'position'* encoding.

In Fig. 5a, we present the learning curves for both the *one-hot* and the 'positions' encoding methods for middle layer size of 32. We learn that using the suggested new encoding, the network converges to a local minimum and the network trained using the *one-hot* performs better. We can relate this result to the fact that now the state encoding is more complex, thus with the new encoding for the network to learn the Q function, it needs to also to decode the state representation, meaning, it should 'understand' the state encoding which is harder. In this case, perhaps the network is not expressive enough and it can't learn a good estimate of the Q function. To support our last argument, we compare again between the two encoding methods, this time with a middle neuron layer of size 128. The convergence results are presented in Fig. 5b. As we can see, the two encoding methods converge to a good result. We summarize interesting learning features in Table 2. The total learning time is presented and it

Num. hidden neurons	Encoding method	Total learning time [s]	Best average acc. reward
32	*one-hot*	306.58	7.2
	positions	691.98	−205.6
128	*one-hot*	434.55	4.1
	positions	498.96	8.0

Table 2: Comparison of learning time and best accumulated episodic reward between two encoding methods: *one-hot* and *positions* for two middle layer sizes: 32 and 128.

is the time took the network to complete 5000 episodes. Focusing on the case where the middle layer size is 128, we can see that there is a small learning time difference between the encoding methods, but both achieves high average accumulated reward.

In terms of memory consumption, obviously the suggested encoding is better, since each state is represented only by 19 values vector instead of 500 values, which is more than 25 times smaller. Even with the fact that the more complex encoding forces us to use a larger hidden layer (but still smaller number of network's parameters), we still get good results (with the cost of more longer learning process).

As a final say, we want to note that in this 'toy problem' of 5×5 grid world, the memory saving might look small, but this is due to the fact that there are only 500 states in this environment. In real world problems, the state space is typically tremendous and using a *one-hot* encoding is not possible, e.g. chess has approximately 10^{50} states. Thus, a smarter encoding is necessary.

1.2.7 Different Optimizers

We discuss the following 3 optimizers: stochastic gradient descent (SGD), Adam and RMSProp. We briefly describe each of those optimizers: Suppose that the loss we aim to minimize is some $L(\theta)$, and we use its estimate based on the data points, $\hat{L}(\theta) = \frac{1}{n}\sum_{k=1}^{n} L_k(\theta)$, where $L_k(\theta)$ is the loss for the k-th data point.

'SGD' - stochastic gradient descent is gradient descent method where the gradient is simply estimated over a different batch at every step. The update rule is given by:

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta \hat{L}(\theta_k)$$

where η is the learning rate parameter.

'RMSProp' - In 'RMSProp', the algorithm scales each parameter's learning rate by a value representing the volatility of the gradient w.r.t it, but to prevent the learning rate from decreasing too much over time, the algorithm decays this value. The update rule is given by:

$$g_{k+1} = \alpha g_k + (1-\alpha)(\nabla_\theta \hat{L}(\theta_k))^2$$

$$\theta_{k+1} = \theta_k - \frac{\eta}{\sqrt{g_{k+1} + \epsilon}} \odot \nabla_\theta \hat{L}(\theta_k)$$

where α and η are parameters.

'Adam' - The idea of 'Adam' is to incorporate both momentum and adaptive learning rate into a single algorithm. The update rule is given by:

$$m_{k+1} = \beta_1 m_k + (1-\beta_1)\nabla_\theta \hat{L}(\theta_k)$$

$$\tilde{m}_{k+1} = \frac{m_{k+1}}{1-\beta_1^k}$$

$$v_{k+1} = \beta_2 v_k + (1-\beta_2)(\nabla_\theta \hat{L}(\theta_k))^2$$

$$\tilde{v}_{k+1} = \frac{v_{k+1}}{1-\beta_2^k}$$

$$\theta_{k+1} = \theta_k - \frac{\eta}{\sqrt{\tilde{v}_{k+1}} + \epsilon} \odot \tilde{m}_{k+1}$$

where β_1, β_2 and η are parameters.

We tested the 3 optimizers above, and the results are presented in Fig. 6, where the average accumulated episodic reward is displayed for all the 3 optimizers over 5000 episodes.

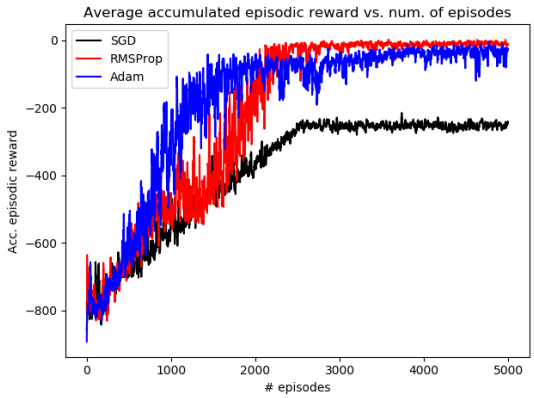


Figure 6: Convergence comparison between 3 different optimizers: 'SGD', 'RMSProp' and 'Adam'.

While the 'SGD' performs the worst of all 3, since it converges to some bad local minimum, both the 'RMSProp' and 'Adam' optimizers converges to a Q function that induces a good policy (with positive average episodic reward). Notice the difference between the later two, where using 'Adam' results in earlier significant average reward increase, but slower and noisier overall convergence, i.e. 'RMSProp' convergence after smaller number of episodes and with lower variance, but 'Adam' improves significantly faster at the beginning.

1.3 Policy Gradient Architecture - Actor-Critic

In this section, we choose to use an Actor-Critic method for solving the taxi environment. We suggest a fully connected network with two output head, one to learn directly the policy $\pi(a|s)$ and the other to estimate the value function $\hat{V}(s)$, as illustrated in Fig. 7. The input dimension is 500 (*one-hot* encoding), and is followed by a

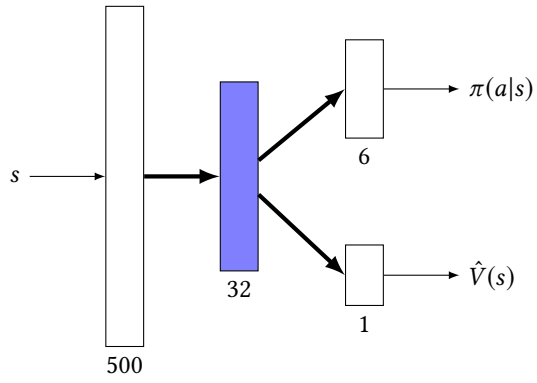


Figure 7: Actor Critic network architecture. The input is a *one-hot* encoding of the environment's state and the hidden layer dimension is 32. Afterwards, the network splits to output both the policy $\pi_\theta(a|s_t)$ and the value function $\hat{V}(s)$.

layer of size 32. Afterwards, the network splits into two different layers, the first is the policy output and the second is the value function output.

We implement an online Actor-Critic algorithm, where the objective of the network is to minimize the following loss function:

$$L(\theta) = -J(\theta) + (r(s_t, a_t) + \gamma \hat{V}_\theta(s_{t+1}) - \hat{V}_\theta(s_t))^2$$

where the first term is minus the expected return

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_t \gamma^t r(s_t, a_t) \right]$$

and the second term is the squared *TD error*.

Practically, we estimate the gradient of this loss as follows. We take an action $a_t \sim \pi_\theta(a|s_t)$ and get the transition (s_t, a_t, r_t, s_{t+1}). Then, the gradient of the first term is estimated by:

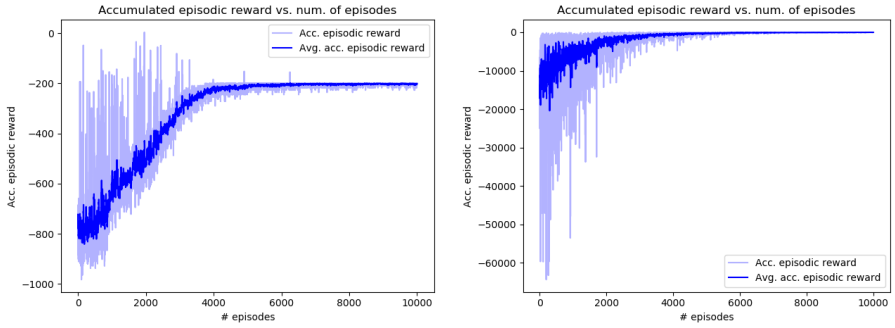
$$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}(s_t, a_t)$$

where $\hat{A}(s_t, a_t) = r(s_t, a_t) + \gamma \hat{V}_\theta(s_{t+1}) - \hat{V}_\theta(s_t)$. For the second term we use a *Huber* loss instead of a pure quadratic loss to reduce the cost for large deviations and stabilize the learning. Recall:

$$Huber(x) = \begin{cases} \frac{1}{2}x^2, & |x| \leq 1 \\ |x| - \frac{1}{2}, & |x| > 1 \end{cases}$$

Furthermore, we normalize the reward for every batch to stabilize the learning process.

Applying the algorithm to the originally described taxi environment gets stuck on



(a) (b)

Figure 8: The accumulated episodic reward and its average over the most recent 10 episodes for the actor-critic model for both limiting and not limiting the episode's length to 200 transitions in (a) and (b), respectively. Notice the scale difference.

an average accumulated episodic reward of approximately 200. We can assume it is because the learned policy is to never try *'pickup'* and *'dropoff'* which results in a reward of -10 when is done illegally, thus the algorithm doesn't manage to learn to pickup and deliver the passenger to its destination. To avoid the algorithm from getting stuck in a bad policy, we change the stopping rule of an episode. Instead of trying 200 transitions at most, an episode ends only when the passenger is delivered to its destination. In this way, we assure that every episode ends with a 'good' action. The difference from the *Q learning* we saw before is that this algorithm is online, whereas the previous one is a batch algorithm where we use a *Replay Buffer* to decorrelate consecutive states. In Fig. 8a we can see that the model converges to a bad policy, where in Fig. 8b, the model converges to a policy that achieves a positive average accumulated episodic reward. Note that there is a scale difference between the two learning curves due to the fact that not limiting the episode's length, results in longer that 200 transitions episodes.

2 Solving a Control Task from Visual Input

2.1 Environment Description

The acrobot environment includes two joints and two links, where the joint between the two links is actuated. Initially, the links are hanging downwards, and the goal is to swing the end of the lower link up to a given height, once the lower link passes the given height, the episode end and the agent is rewarded with +1, at each other time step the agent is punished with a -1 reward.

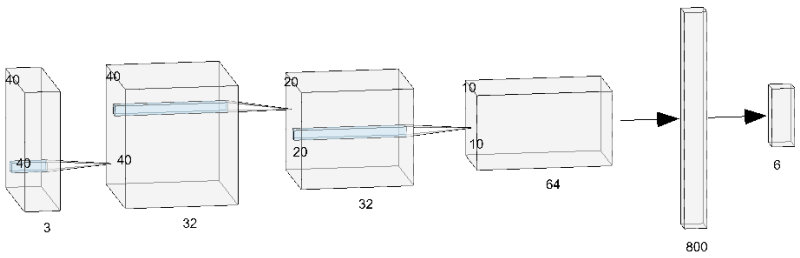


Figure 9: Three layers Convolutional neural network followed by two fully connected layers. At each convolutional step there is a max pooling layer followed by batch normalization.

2.2 Convolutional Neural Network DQN Architecture

In the acrobot environment, there are 3 possible actions:

$$\mathcal{A} = \{\textit{'left'}, \textit{'stay'}, \textit{'right'}\}.$$

Thus, the action space is $|\mathcal{A}| = 3$.

The states of the environment are comprised of 6 parameters:

$$\mathcal{S} = \{\cos\theta_1, \cos\theta_2, \sin\theta_1, \sin\theta_2, \omega_1, \omega_2\}$$

where θ_1 and θ_2 are the rotational joint angle of upper and the lower joints respectively, and ω_1, ω_2 are the joints angular velocities. At each step we construct a screen image corresponds with the agent position in the game. For state representation we subtract the current screen with the last screen in order to represent the velocity and movement direction of the agent. Our objective is to the same as in section 1.2. For this task we build a CNN with 3 convolutional layers: 32, 32, 64 filters accordingly, followed by two fully connected layer, as depicted in Fig. 9. The reason we use 3 wide layers is to construct a rich network which could express the environment and refine important features out of the current state using max pooling. The activation function we used per CNN layer is ReLU.

2.2.1 Input Extraction And Preprocessing

As mentioned earlier, the original state representation the environment supply consist of 6 parameters. We propose using a visual input for the solution of the task. Hence we construct out of the environment '.render()' command an image with 500×500 pixels RGB, which is a representation of the game board for each stage of the game. The picture at each stage is mostly comprised of white pixels and a very few important objects. Hence we down sample it to a size of 40×40. This also gives us a much

more efficient training duration, since we use only a CPU for the task. As a single image provides a good representation for the state, it doesn't supply any knowledge about the joints velocities and directions of movement. As a result we provide as input in each time step a difference image, which is a subtraction between the current board and the last step board. Another way simulating joints movement is using a concatenation of a few images, we tried using this method, but because the lack of GPU our training sessions went to long.

2.2.2 Regularization and Robustness methods

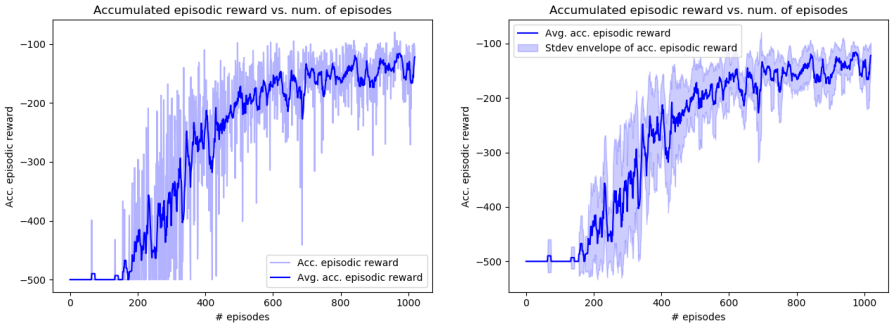
Since the acrobot environment introduce a very wide space dimension, it is very important to construct a good, reliable and robust agent. We would like our agent to match a wide policy, study from a wide self experience and accumulate a knowledge for dealing with new scenarios, which it haven't seen yet. For accumulating a wide experience we use both epsilon greedy action choosing, that promises exploration through all training session, and *Prioritized Experience Replay* (Schaul *et al.*, 2015), which is biased to sample more experience from a transition that is less common in the memory buffer, without over-fitting to these transitions. At architecture level, we use batch normalization and max pooling methods, where the previous scales the input for each layer and the latter filters only dominant features from each layer, combined together they extract both faster training and robustness of the agent for a new input state. Finally, we regularize at backpropagation level by clipping gradients to prevent large update values for the network weights.

2.3 Training

For optimizing the network, based on previous section results, we used 'RMSprop' optimizer. In order to avoid large updates resulting with high variance accumulated rewards we choose a small learning rate the size of $lr = 24 \cdot 10^{-4}$. At beginning of training we force our agent to high exploration rate by using epsilon greedy, thus in first period of training the agent sample actions randomly, with ϵ descending during training such that it reaches its minimum value of 0.03 after 850 episodes. Every 500 transition we update the target network using the agent weights, yielding lower gradients which derived from lower loss values, because the target network is relatively updated. In fig. 10 we demonstrate training curve, noticing the speedy convergence to good results for less than thousand episodes. This can be credited to the use of PER, high target update frequency and using batch size of 32 for each back propagation. We define a stopping criterion for a trained model such that it reaches an average accumulated reward higher than -150 for the last 20 episodes, which is being reached after thousand episodes. We justify this choice by aiming for a stable network, instead of stopping at an instantaneous minimum accumulated reward.

2.4 Prioritized Experience Replay

Experience replay lets online reinforcement learning agents remember and reuse experiences from the past, while other methods simply sample transitions at the fre-



(a) (b)

Figure 10: The accumulated episodic reward and its average over the most recent 10 episodes for the DQN model for the acrobot task presenting the episodic accumulated reward and its average over lat 10 episodes for each step in (a) and the average episodic accumulated reward together with its std over 10 episodes in (b).

quency they were originally experienced by the agent, this method replay important transitions more frequently, and therefore learn more efficiently. We define the importance of the sampling in correlation with the TD error it generates, that is:

$$p_t = |TD| = |r(s_t, a_t) + \gamma \max_{a' \in \mathcal{A}} Q_{\theta_{target}}(s_{t+1}, a') - Q_\theta(s_t, a_t)| + \epsilon$$

where ϵ assures that p_t is above 0. This way we encourage both a robust net, experiencing also rare transitions, and specifically sample at beginning of the training session the transitions where the reward was higher, as they are rare for start. We define the sampling probability to be:

$$P_t = \frac{p_t}{\sum_k p_k^\alpha}, \ \alpha \in [0, 1] \tag{2}$$

as α is is a randomness factor such that for its minimum value it forces a random sampling. Prioritized replay introduces bias because it changes the sampling distribution, and therefore changes the solution that the estimates will converge to. Its is possible to correct this bias by using importance-sampling (IS) weights

$$W_{IS_i} = \left(\frac{1}{N} \cdot \frac{1}{P_i}\right)^\beta \tag{3}$$

that fully compensate the non uniform probabilities if $\beta = 1$. N is the total number of samples in the buffer. As mentioned by (Schaul et al, 2015), for stability reasons, we always normalize weights by $\frac{1}{\max_i W_i}$ so that they only scale the updates downwards. For implementation of PER we use a binary sum tree as we further discuss below.

2.4.1 Sum-Tree

A naive implementation might use a list of all memory transition sorted according to their probabilities, and for each batch sample transition using these probabilities. We choose much more efficient implementation using a binary sum tree. For a binary sum tree (BST), each nodes has maximum of two child's and its value is the sum of their values. Each leaf of the tree represents a single transition sample, as it hold both a pointer to the transition (s_t, a_t, r_t, s_{t+1}) and its value represents the transition sample probability. Note that tree root holds the entire transition probabilities sum. While sampling a batch, we divide the accumulated probabilities sum by the batch size, and at each range we sample a value. Afterwards we search the tree for the appropriate transition as further explained below. At the end of the tree search we attend the transition sample using the pointer at the leaf. Update of a leaf node involves propagating a value difference up the tree, obtaining $O(\log n)$. Sampling follows the thought process of the array case, but achieves $O(\log n)$ and Inserting a new leaf achieves $O(1)$.

Algorithm 1 Sample a transition from memory buffer implemented as BST

Data: a scalar t uniformly sampled from [0, Root Value]
Result: A transition (s_t, a_t, r_t, s_{t+1}) which the final leaf points at
Initialize node = root
while node *is not a leaf* **do**
  if $t \leftarrow Value_{left\,t}$ **then**
    node = left child
  else
    $t = t - \text{Value}_L$
    node = right child
  end
end