# Advanced Subjects in RL - Wet Exercise

Orr Krupnik - 302629027
Shunit Haviv - 305147456

December 11, 2018

## 1 Solving the Taxi environment

### 1.1 Representation of the Taxi Environment

The taxi environment state is encoded as a 1-hot vector of size 500:

- 25 taxi positions
- 5 possible locations of the passenger
- 4 destination locations

The action space is encoded as a 1-hot vector of size 6 indicating the action:

- 0: move south
- 1: move north
- 2: move east
- 3: move west
- 4: pickup passenger
- 5: dropoff passenger

### 1.2 DQN Architecture

We built a DQN architecture using a fully-connected neural network with the following parameters:

- Two fully connected layers, with 64 neurons in the hidden layer
- Initial learning rate: 0.002
- Experience replay memory with 20,000 slots

| Hidden Size | Run Time (2000 episodes) |
|---|---|
| 8 | 933.62 sec |
| 16 | 556.44 sec |
| 32 | 146.14 sec |
| 64 | 160.34 sec |
| 128 | 266.21 sec |
| 256 | 398.78 sec |
| 512 | 832.70 sec |

Table 1: Comparison of run time for different hidden layer sizes for the Taxi-v2 task.

- Learning mini-batches of 256 samples
- Discount factor of $\gamma = 0.99$
- Updates to the target network every 600 steps
- We used an $\epsilon$-greedy exploration policy, with an initial value of $\epsilon = 1$ (fully random actions) decaying exponentially with a rate of 0.999, to a minimum value of $\epsilon = 0.1$ during training. For evaluation, we used an $\epsilon$-greedy policy with $\epsilon = 0.01$.
- We trained the network for 2000 episodes, where each episode continues until the pickup task is completed successfully, or until a timeout of 200 steps.

### 1.2.1 Hidden Layer Size

We tried different hidden layer sizes, and found that 16 was the most fit one. The results are described in figure *

When choosing bigger hidden layer size, our complete model is bigger and as a result we need more memory and higher training time. Furthermore, a bigger model can express better a variety of cases in comparison to a smaller model. Allowing a bigger model can potentially lead to over-fitting, and so hurt the generalization. Smaller layer size is faster in training and smaller in memory consumption, but a too small model may not be able to generalize and converge. The overall trade-off is between generalization and over-fitting. In the specific problem of Taxi-v2, it is important to state that since it is a deterministic problem, there is no problem in over-fitting.

### 1.2.2 Optimization

First, we optimized the system and used the parameters described above. The results are shown in figure 2. Each evaluation point is the average of evaluation over 10 episodes, using an $\epsilon$-greedy strategy with $\epsilon = 0.01$. Evaluation is run once every 50 training episodes (approx. 10,000 training steps at first, and much less later).

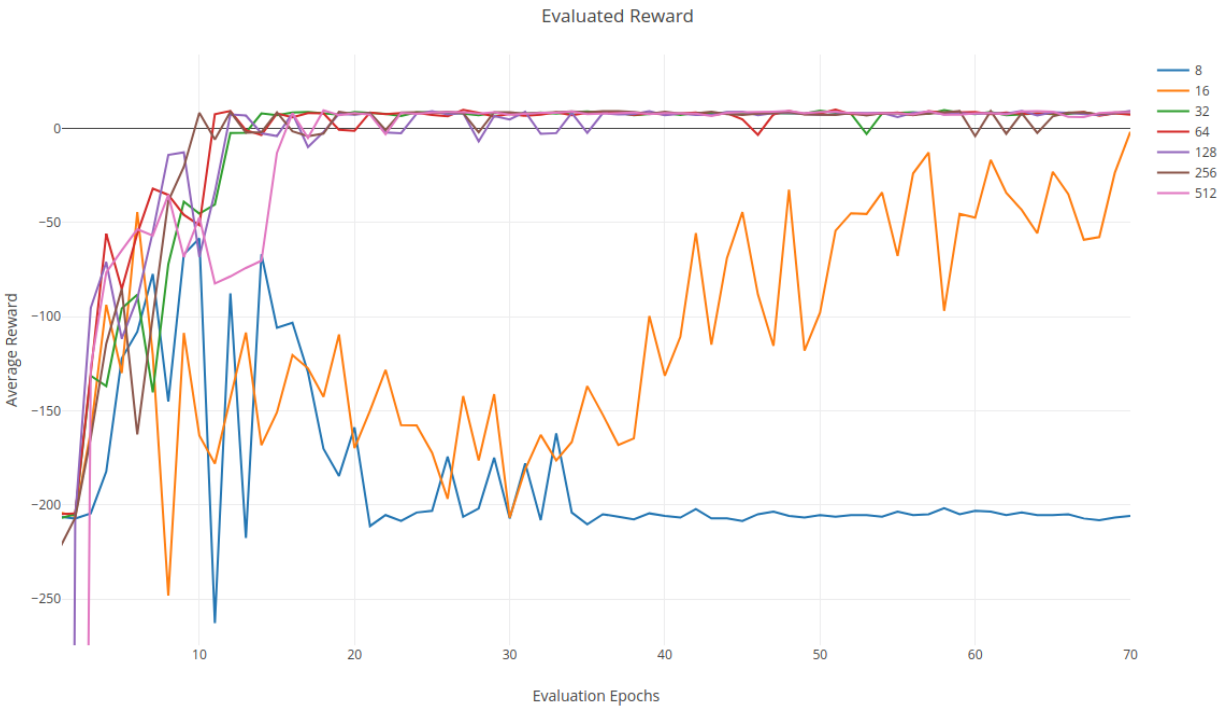


Figure 1: Comparison of different hidden layer sizes for the Taxi-v2 task.

### 1.2.3 Regularization

We ran multiple test with different dropout and $l1$ parameters. The results are displayed in Figure 3.

### 1.2.4 Alternative Representation

We chose the following representation: for the state encoding we use a 4-hot vector of size 19:

- 1-hot of size 5 for taxi location in the horizontal axis
- 1-hot of size 5 for taxi location in the vertical axis
- 1-hot of size 5 for possible locations of the passenger
- 1-hot of size 4 for destination locations

The dense representation we chose did give some memory usage benefit, but since the representation is now 4-hot instead of 1-hot, training the network took more time since there is a bigger amount of calculations. Furthermore, the model managed to converge slower than the initial representation, as can be seen in figure 4.

### 1.2.5 Optimizer Comparison

We chose to compare the following optimizers:


Figure 2: Optimal run of Taxi-V2 DQN with a confidence interval of .95.

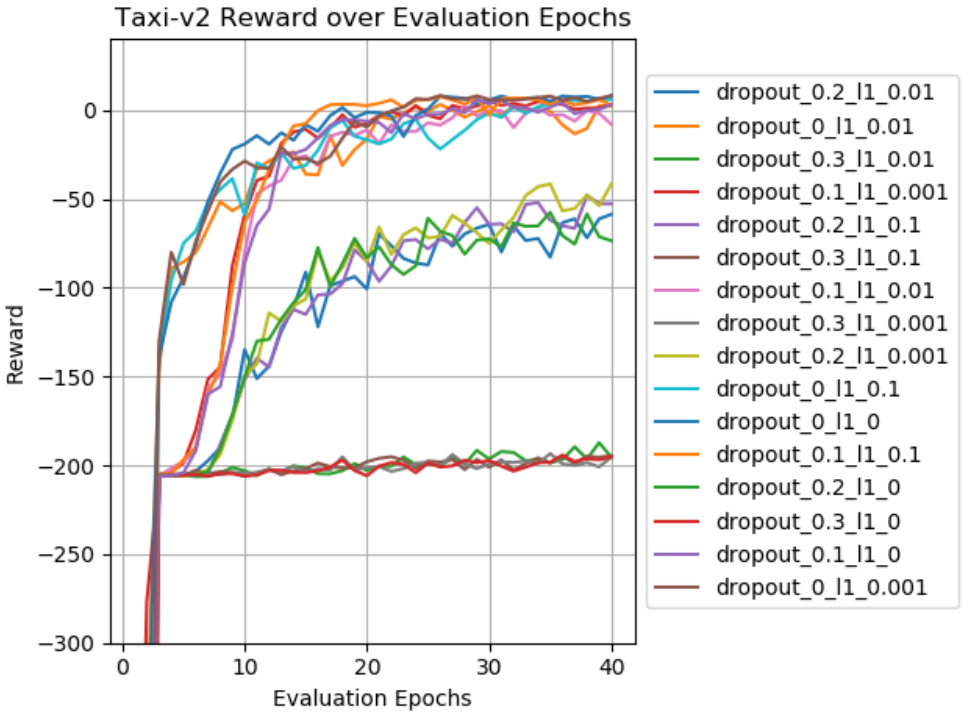


Figure 3: Comparison of different dropout and $l1$ regularization parameters.

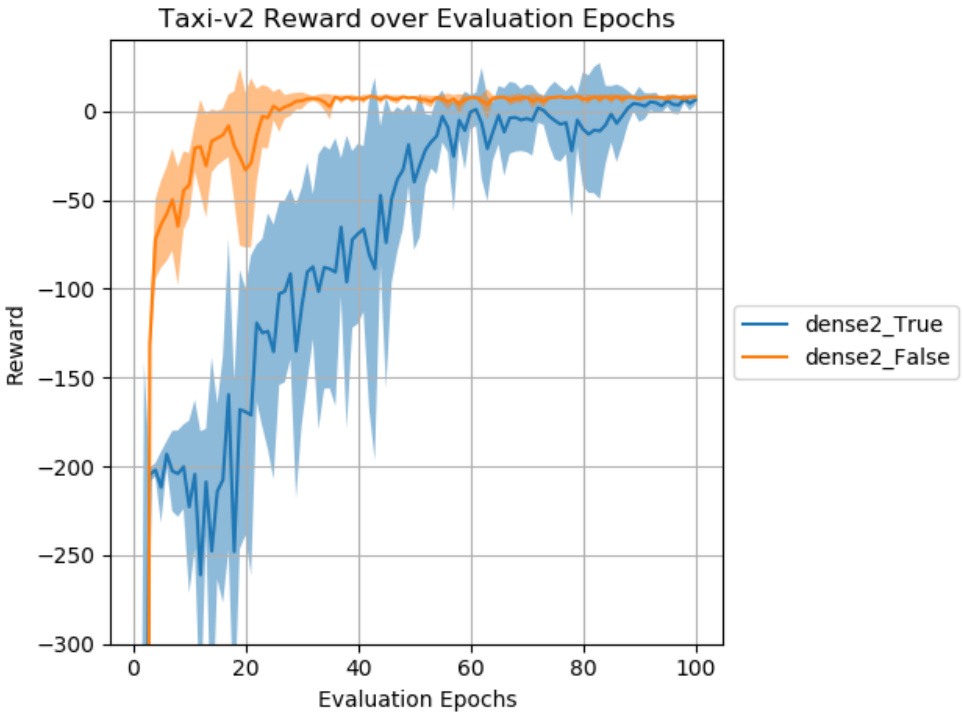


Figure 4: Comparison of different input representations. The orange plot is the original 1-hot representation, while the blue plot is the dense representation described in sec. 1.2.4.

- RMSprop
  RMSprop performs a normalization on the learning rate in a way that causes smaller updates (lower learning rates) for parameters associated with frequently occurring features, and larger updates (higher learning rates) for parameters associated with infrequent features. RMSprop update rule:

$$\begin{aligned} g_t &= \nabla J(\hat{\theta}_t) \\ v_{t+1} &= \gamma v_t + (1-\gamma) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{v_{t+1}+\epsilon}} g_{t+1} \end{aligned} \tag{1}$$

  $v_t$ is an exponential moving average of the square of the gradient. The learning rate is divided by $v_t$, which means that the more frequent a feature (large value in $v_t$) will cause a smaller learning rate.

- Adam
  Adam combines RMSprop with Momentum approach. Momentum adds an aspect of averaged gradient direction to the update, simulating some sort of acceleration and adding robustness to the update. Adam update rule:

$$\begin{aligned} g_t &= \nabla J(\hat{\theta}_t) \\ m_{t+1} &= \beta_1 m_t + (1-\beta_1) g_t \\ v_{t+1} &= \beta_2 v_t + (1-\beta_2) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{m_{t+1}+\epsilon}} v_{t+1} \end{aligned} \tag{2}$$

  $v_t$, as in RMSprop, is an exponentially decaying average of past squared gradients, and $m_t$ is an exponentially decaying average of past gradients similar to momentum.

- Adamax
  Adam implements momentum $m_t$ as the exponentially decaying $l_2$ norm of past gradients. Adamax offers to use $l_\infty$ instead of $l_2$ in order to get more stable results. To avoid confusion with Adam, we use $u_t$ to denote the infinity norm-constrained $v_t$: $u_{t+1} = \beta_2^\infty u_t + (1-\beta_2^\infty)|g|_t^\infty = max(\beta_2 \cdot u_t, |g_t|)$
  Adamax update rule:

$$\begin{aligned} g_t &= \nabla J(\hat{\theta}_t) \\ m_{t+1} &= \beta_1 m_t + (1-\beta_1) g_t \\ u_{t+1} &= \beta_2^\infty u_t + (1-\beta_2^\infty)|g|_t^\infty = max(\beta_2 \cdot u_t, |g_t|) \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{m_{t+1}+\epsilon}} u_{t+1} \end{aligned} \tag{3}$$

  Note that as $u_t$ relies on the max operation, it is not as suggestible to bias towards zero as $m_t$ and $v_t$ in Adam.

The comparison was made with the same parameters described above for all three optimizers. The results are described in figure 5. The most suitable optimizer in this experiment is Adam. This is probably due to the simplicity of our problem which is deterministic, and additional optimizations may not always carry an additional value.

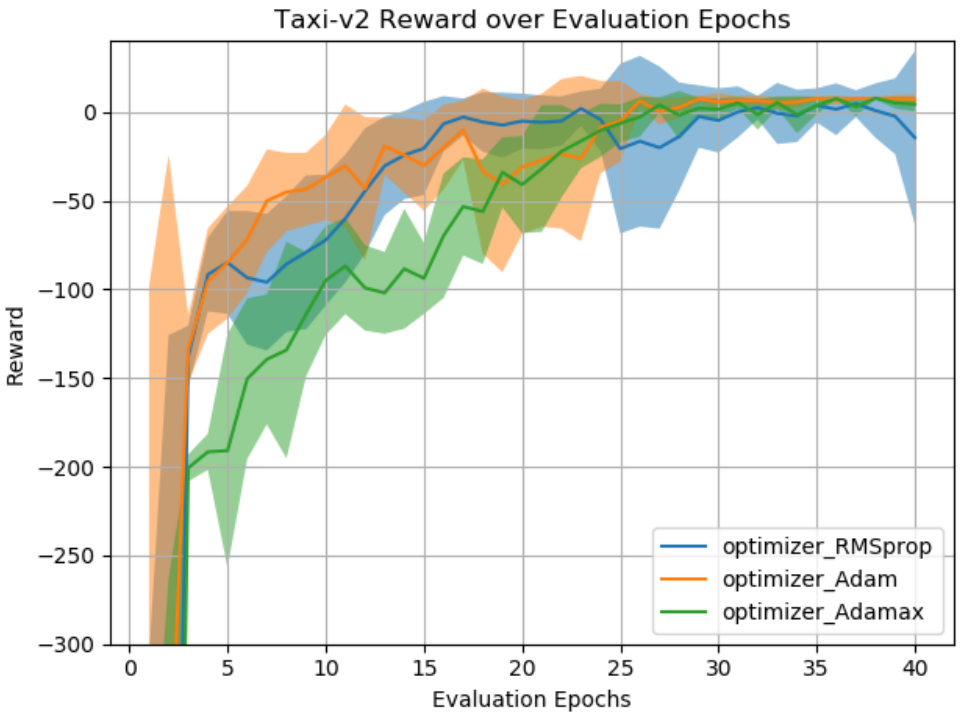


Figure 5: Comparison of different optimizers for the Taxi-v2 task.

## 1.3 Policy Gradient

To solve the Taxi-v2 problem with a policy gradient, we used a variant of the Advantage Actor Critic algorithm. The specifics are described below.

### 1.3.1 Network Architecture

We used a similar architecture for both the actor and critic as used for the DQN described above. We used two fully connected, two-layer neural networks, with 50 neurons in the hidden layer. Both networks used the one-hot state representation as input. The output layer for the policy (actor) network is 6 neurons wide (as fitting the action space dimension), and the value (critic) network has a single neuron in the output layer.

### 1.3.2 Training Parameters

We used A2C with trajectory rollouts of 40 steps each. To update the value and policy networks using batches, we used 64 different "workers", each running a copy of the environment. At the end of the 40-step rollout, we used the collected values ($v_t$), actions ($a_t$) and rewards ($r_t$) to calculate the returns and estimate the advantage ($A_t(s_t, a_t)$) for each step. This in turn was used to calculate the loss functions for the value and policy networks.

To promote exploration, we used an entropy regularizer with a decaying weight coefficient. Entropy regularization enforces high entropy on the actions selected at first, which produces more diverse actions and better exploration. When lowering the weight of the regularization

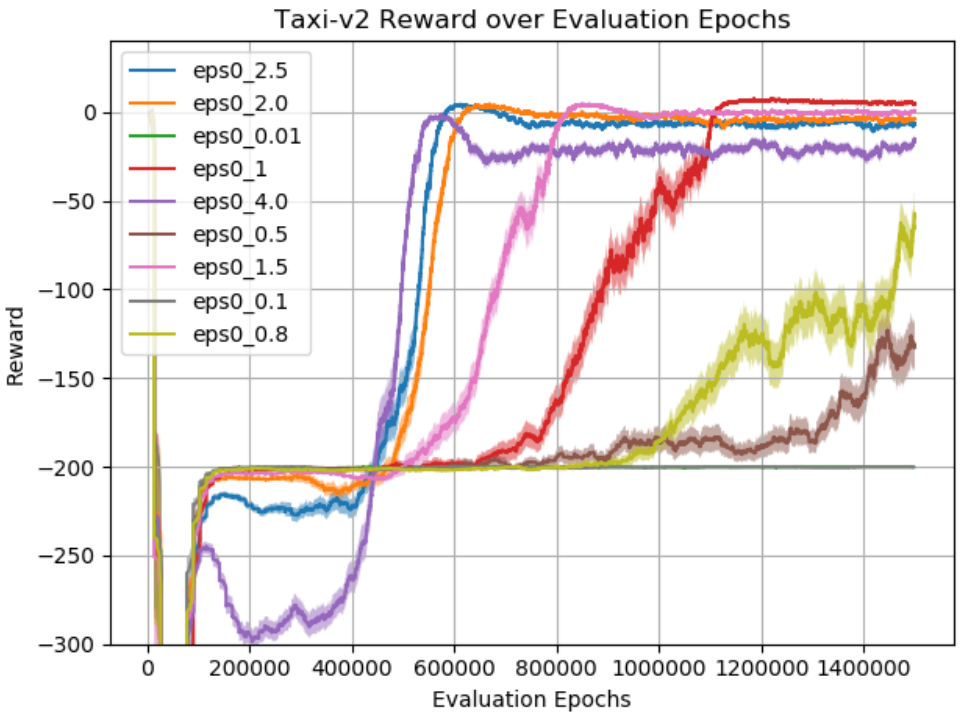


Figure 6: Comparison of different values for the entropy regularization weighting. Values mentioned in the legend are the initial values, then decayed linearly to 0 over $10^6$ steps. The plot is averaged over 200 steps, and the confidence interval presented is of .95. Note that lower initial values may converge more slowly, however they eventually produce better results.

as training progresses, the actions selected become more and more "greedy". The weighting of this regularization term in the loss function turned out to be one of the most influential parameters in terms of convergence speed and values, as can be seen in figure 6.
The model that is included in our git is the best out of the above, with $eps0 = 1$.

# 2 Solving a Control Task from Visual Input

We used DQN with some changes from the architecture described above, to solve the Acorbot-v1 task from visual input. Details are described below.

## 2.1 Preprocessing

We used Gym's render command to obtain the screen image of the Acrobot-v1 environment, which we then converted to grayscale, scaled down to $40 \times 40$ and negated to get a black background. Additionally, to gain information about the motion of the arm, we concatenated 4 consecutive images to create the input state for the network. A sample of our observation space can be seen in figure 7.

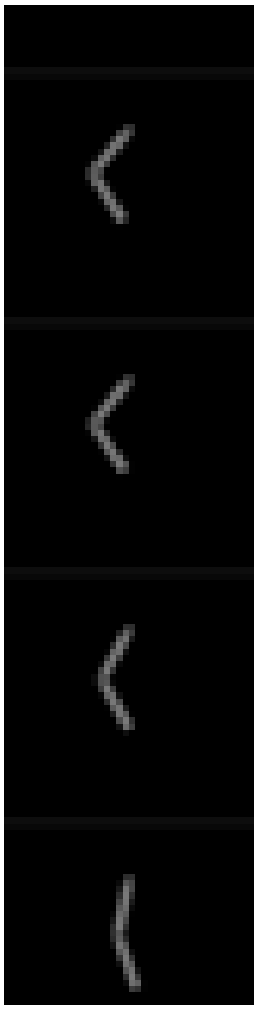

Figure 7: Observation input for the Acrobot-v1 DQN solution. In practice, the concatenation of states was conducted over a third (channel) dimension. This was spread out here over the height dimension for visual simplicity.

## 2.2 Architecture

We built a Dueling DQN architecture using a CNN with the following layers:

- Convolutional layer: 16 Channels, $5 \times 5$ filter, stride 2
- Convolutional layer: 32 Channels, $5 \times 5$ filter, stride 2
- Convolutional layer: 32 Channels, $5 \times 5$ filter, stride 2
- Fully connected layer: 128 neurons to 6 outputs (for Advantage values of the actions) and 128 neurons to 1 for the Value function.
- Aggregating Layer to calculate the Q-values from Advantage and Value, implementing $Q(s,a) = V(s,a) + A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a')$.

We used the ReLU activation function for all layers, and Batch Normalization, which helped stabilize learning significantly. This selection of convolutional layer sizes reduces the image to a $32 \times 2 \times 2$ tensor, which then allows the final fully connected layer to receive this as a flattened input. We found that the described filter sizes were enough to reduce the representation, and therefore we refrained from using additional dimensionality reduction methods, such as MaxPooling.

To combat the sparsity of rewards in the Acrobot task, we used an additional "Success Experience Replay", which saves successful trajectories only. The training process then samples batches from this smaller experience replay with some probability, which becomes smaller as training progresses. This ensures that the network trains on the relatively rare successful trajectories during the early stages of the training process.

To stabilize learning, we used the Double DQN addition (select the best action with the active network, then estimate its Q-value with the target network). This helped us gain more consistent results, and somewhat quickened convergence.

## 2.3 Optimization and Regularization

To optimize our network, we used Adam with learning rate 0.0002. Additional training parameters:

- Experience replay memory with 200,000 slots
- Learning mini-batches of 64 samples
- Discount factor of $\gamma = 0.99$
- Updates to the target network every 1500 steps
- We used an $\epsilon$-greedy exploration policy, with an initial value of $\epsilon = 1$ (fully random actions) decaying linearly to a minimum value of $\epsilon = 0.1$ over $10^6$ training steps. For evaluation, we used an $\epsilon$-greedy policy with $\epsilon = 0.01$.
- We trained the network for 3500 episodes, where each episode continues until the task is completed successfully, or until a timeout occurs after 500 steps.

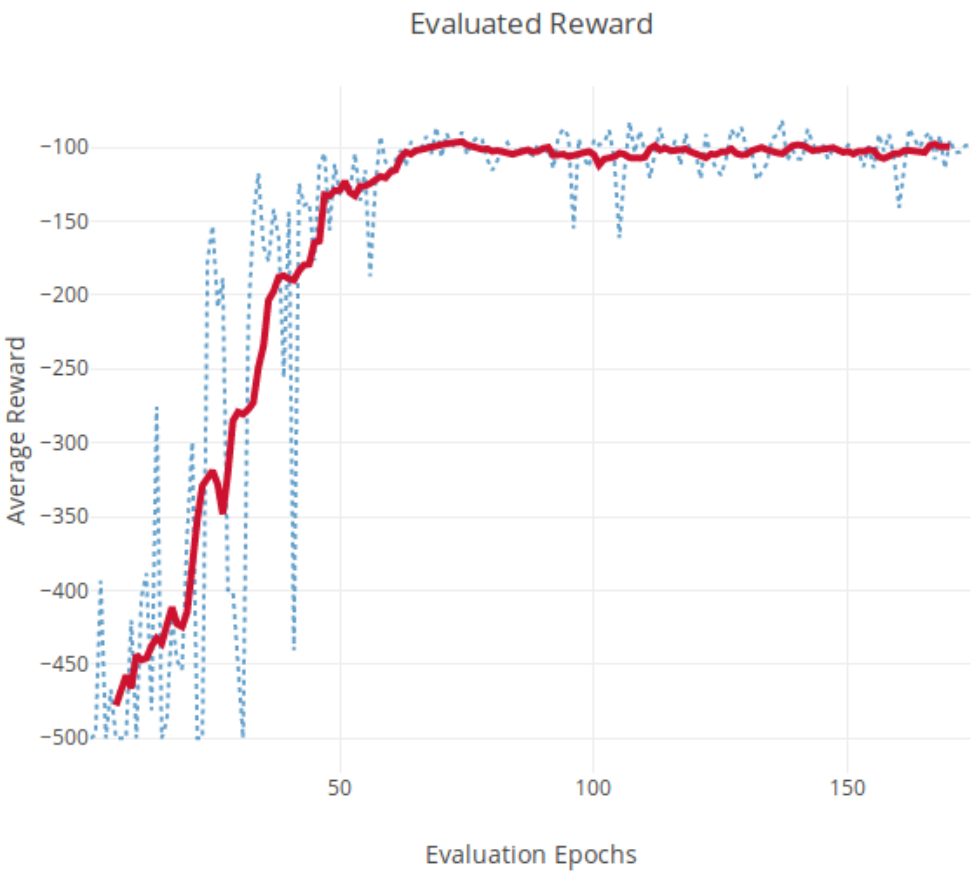


Figure 8: Training process for visual Acrobot-v1 solution using DQN

Evaluation was done every 20 episodes, and conducted by collecting the average reward over 10 episodes with an $\epsilon$-greedy policy, where $\epsilon = 0.01$.

## 2.4 Results

We optimized the system using the parameters described above. The results are shown in figure 8. The best result we've seen was an average reward of $-81.7$ over 10 evaluation episodes (using an $\epsilon$-greedy policy with $\epsilon = 0.01$); the best model achieves a result of $-96 \pm 20$ when averaged over 100 evaluation episodes (using a fully greedy policy). See attached video for best policy results.