TAXI

The basic architecture is 2-fully connected layers. As there are 500 possible distinct states, the input is a one-hot vector sized 500 with the index of the valid state set to 1. The output is of size 6, each index corresponding to an action in [up, down, left, right, pickup, drop-off].
We found that using a hidden layer of size 50 is sufficient to learn to task. The trade-off was that with a larger layer we gain a better ability to correctly learn the Q values, however a larger layer takes longer to train (more parameters) and it hurts generalization. This resulted in 25,356 parameters.

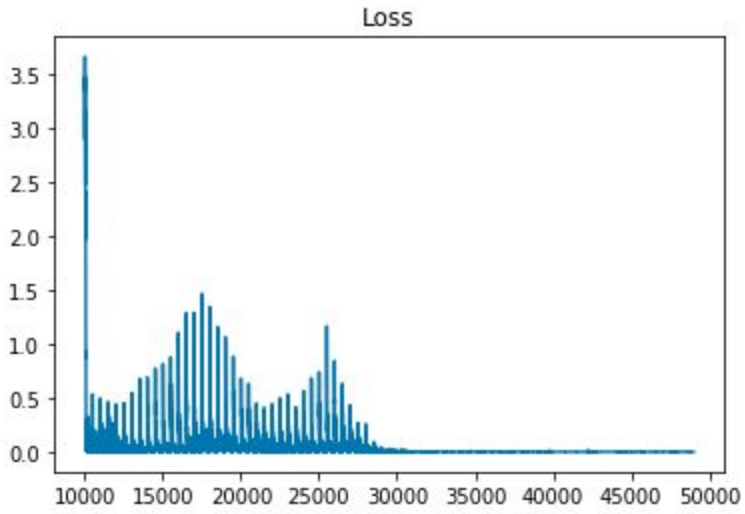
*** In all of our experiments, the X axis denotes the number of ingame steps, not the epoch number. This is in order to emphasize the sample complexity of the algorithms.**

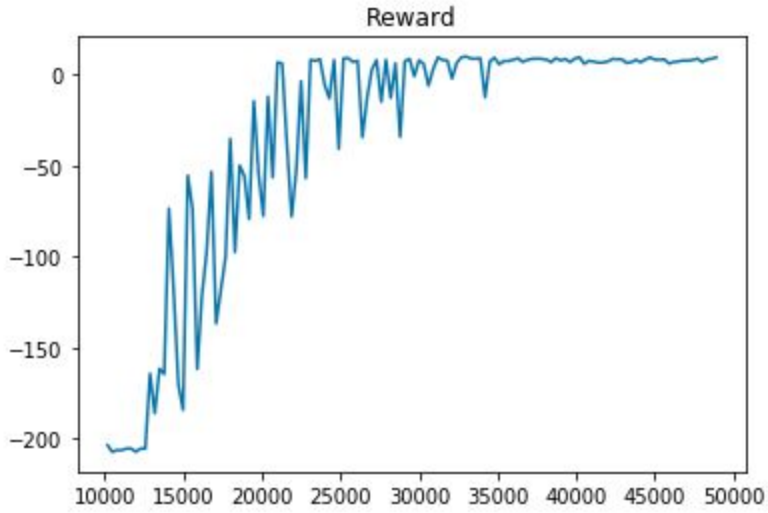
DQN

Our parameters:
mem_capacity = 20000
batch = 256
lr = 0.005
double_dqn = False
gamma = 0.99
num_steps = 50000
target_update_freq = 500
learn_start = 10000
eval_freq = 300
eval_episodes = 10
eps_decay = 10000
eps_end = 0.1
hidden_layer = 50
optimizer = Adam

Since we observe that this problem is deterministic with a small state-action-space, we can exploit this property in order to train the network faster (in less than 50k steps). Hence we used a large experience replay (as all samples are meaningful and not only the most recent ones), we used a large batch size with a small learning rate and we update the target network every 500 steps (since convergence is fast).
Our training begins by 10k random actions (epsilon = 1.0) and then we decay epsilon linearly over 10k steps down to 0.1.
We evaluate our policy every 300 steps and average that over 10 episodes.
We found that double-q learning did not improve our results (it is also observed in the original paper that it doesn't improve the results of all domains).

In addition we make an important observation. The TAXI domain provides a termination signal in two scenarios (1) the game has ended successfully due to the agent dropping the passenger at the correct location (2) 200 timesteps have passed.
While (1) is important to learn from, notice that (2) is a non-markovian property. Q(s,a,t=0) != Q(s,a,t=199). As such, we decided not to store these transitions (not to learn from them) which greatly improved the convergence of our algorithm.



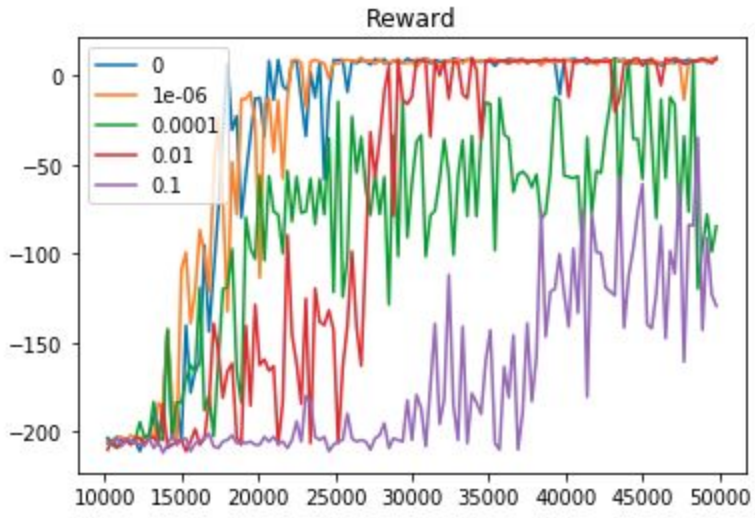


As we can see, the loss reduces very quickly and the spikes are every 500 steps (when the target is updated). In addition, we are able to quickly converge to the optimal policy (~40k steps).

Our training resulted in:
Average reward: 8.447
STD: 2.422641327146881

Dropout

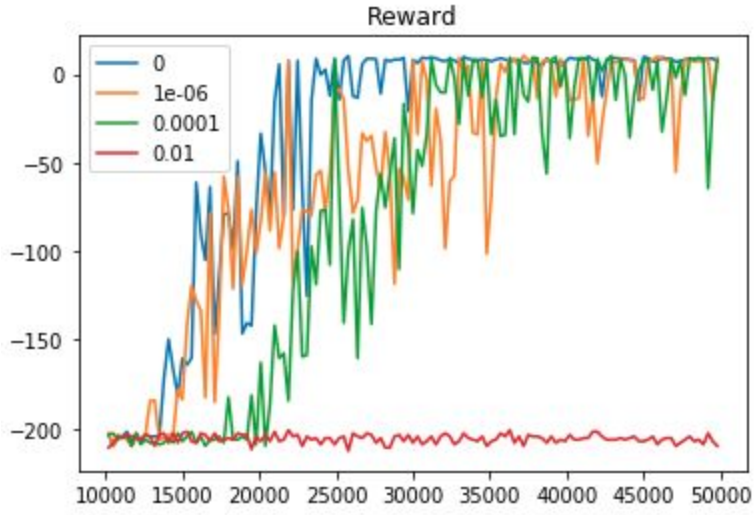
As the problem is a regression problem, dropout doesn't effect the model as it would in a classification problem (in which it can be seen as sampling from a confidence set). As in Q learning we are required to be exact (the Q values may vary slightly between actions and we need to be able to properly distinguish between them) it is expected to damage the performance.



While there exist dropout values in which the model does converge, it doesn't improve performance or convergence speed. In addition, we notice that the performance does not behave "linearly" in the dropout value, e.g., with dropout of 1e-6 and 1e-2 the agent is able to attain good performance, however with a value of 1e-4 it does not converge. We attribute this to the added stochasticity derived from the dropout which does not help especially in such a deterministic domain such as TAXI.

L1 regularization

L1 regularization encourages the network to be sparse. As such, we expect that when the network is not over-parametrized, this regularization will only damage the training regime. Had we wanted to generalize better, we would suggest to use a variant of robust training.



We experimented with 4 different L1 regularization weightings, such that the loss was:

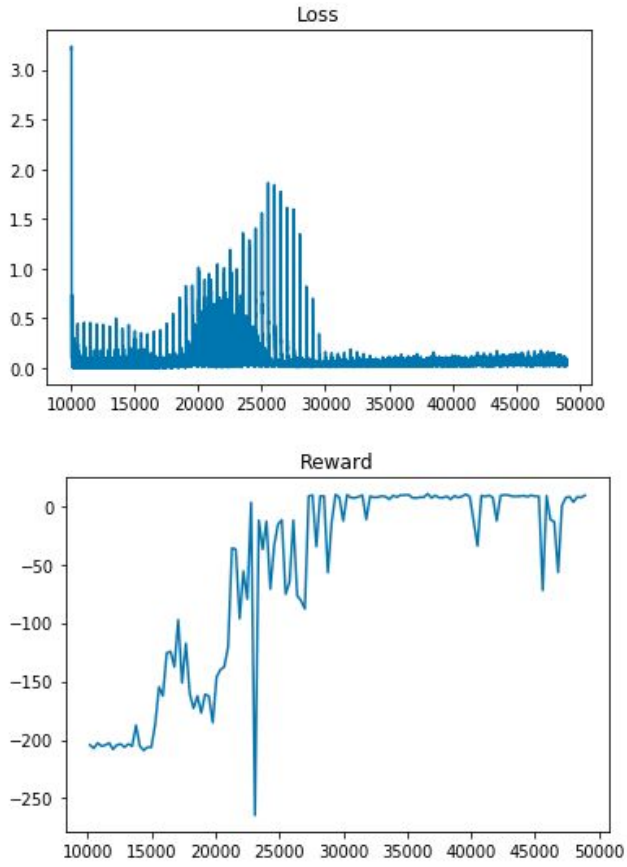
$$L = MSE_{\theta} + \lambda L_1(\theta)$$

Our experiments show that as expected, the L1 does not improve. For small enough values we are still able to learn a good policy, however it is unstable and takes longer to converge.

Alternative encoding

We suggest an alternative encoding which enables us to use much less parameters for training. We flatten the grid (which is 5x5). The vector is all zeros, except for 1 at the index where the agent is located and 2 where the goal is. In addition, we append to the vector a value of 0 if the goal is the passenger and 1 if the goal is the drop-off location.
In total our input is of size 26.

In this case, we used a hidden layer of 500 resulting in a total of 16,506 parameters (as opposed to 25,356 in the one-hot encoding scenario).



Average reward: 4.223
STD: 29.27410581042571

While our approach resulted in a valid policy, it did suffer from larger variance during training and in a larger variance of the final reward. We assume that since the original approach (the one-hot vector) results in an almost tabular setting, it is hard to find a competitive alternative representation given this domain.

Optimizers

Adam optimizer:

The adam optimizer is an adaptive gradient descent optimizer, which uses the previous gradients for momentum (m_t) and scaling (v_t).

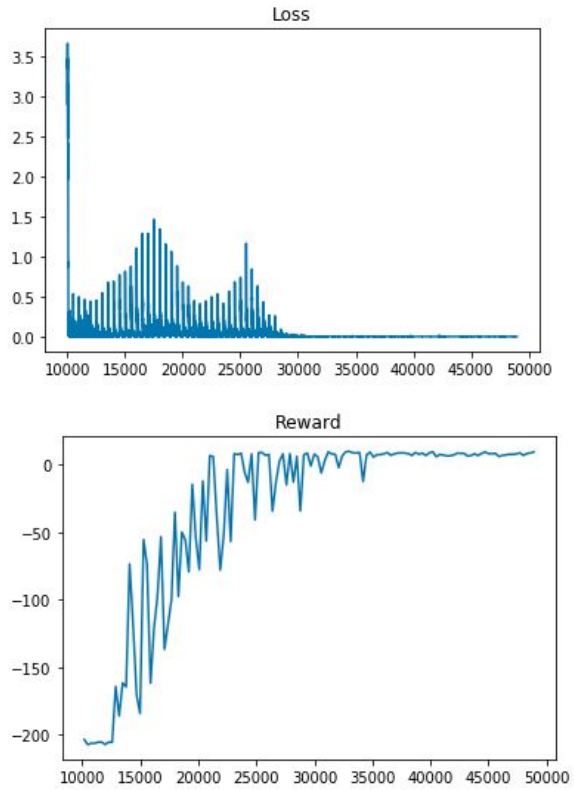
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

These are biased and hence are fixed using a bias correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finally the weights are updated using the following update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$



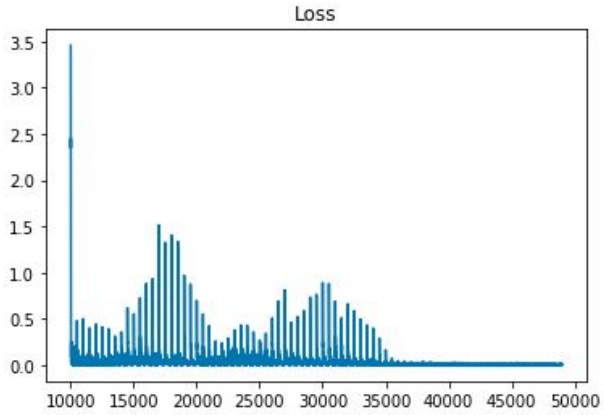
Average reward: 8.447
STD: 2.422641327146881

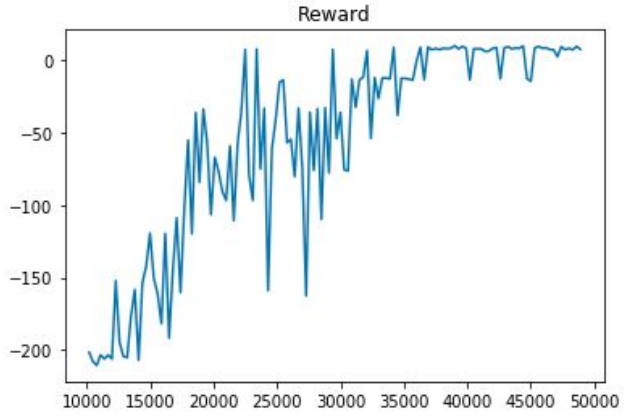
RMSprop optimizer:

The RMSprop optimizer, similar to ADAM, but it only uses the exponentially decaying squared gradients (only scales the gradients and doesn't use the momentum term).

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Below is the result with RMSprop:



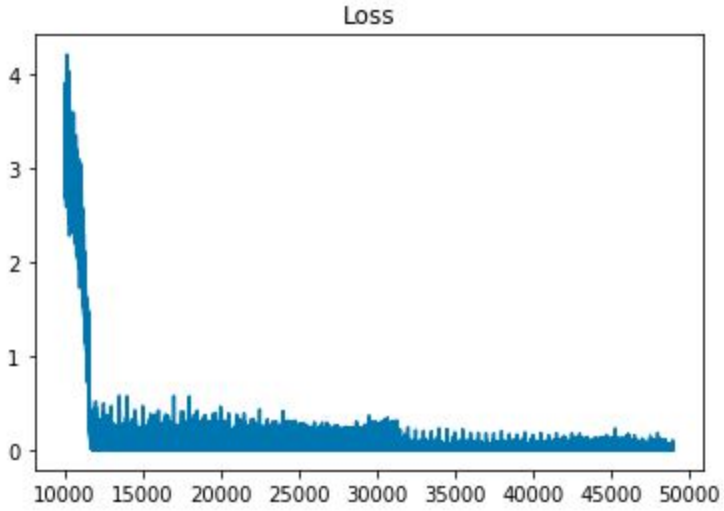


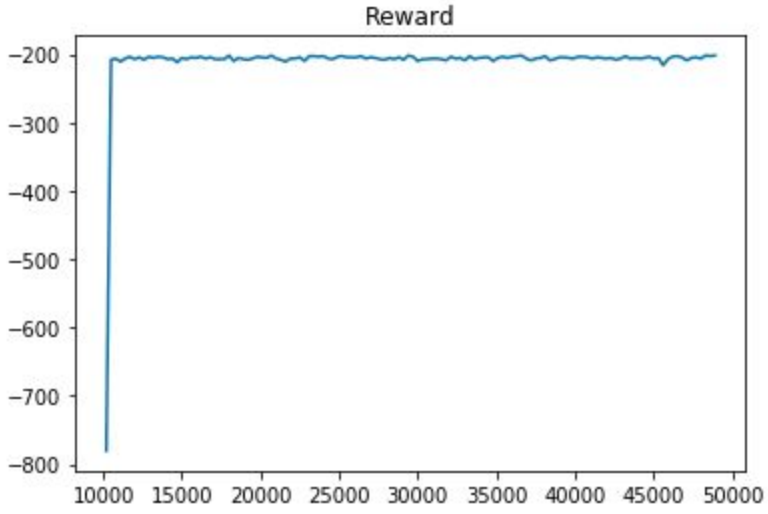
Average reward: 8.536
STD: 2.396811214927033

As can be seen, since RMSprop and ADAM are similar, both result in competitive results, however the momentum term in ADAM does result in faster convergence (in this scenario).

SGD

Stochastic gradient descent is the standard method. As this method is not adaptive it requires proper attention to a decaying learning rate (similar to the requirement in stochastic approximation theory). Since the goal is not to find the optimal parameters, we do not perform such tuning and use the experiments to show this property; e.g., without proper tuning SGD does not converge as the learning rate does not automatically reduce when nearing an optima.





Average reward: -200.0
STD: 0.0

Policy Gradient

We decided to use a variant of A3C called A2C. This is the syncronous version, e.g., Advantage-Actor-Critic.

The architecture we use is similar to the previous section with the addition of another identical network which outputs the value (the critic network).

We found that the policy gradient approach is sub-optimal for solving such a domain. While eventually we were able to train a network to solve the taxi-domain, it a rigorous hyper-parameter sweep to find a set of good parameters.

Our parameters:
lr = 0.01
gamma = 0.99
num_steps = 2e6
rollout_steps = 40
hidden_layer = 50

num_processes = 64
entropy_start = 10.0
entropy_end = 0
entropy_decay_steps = 1e6

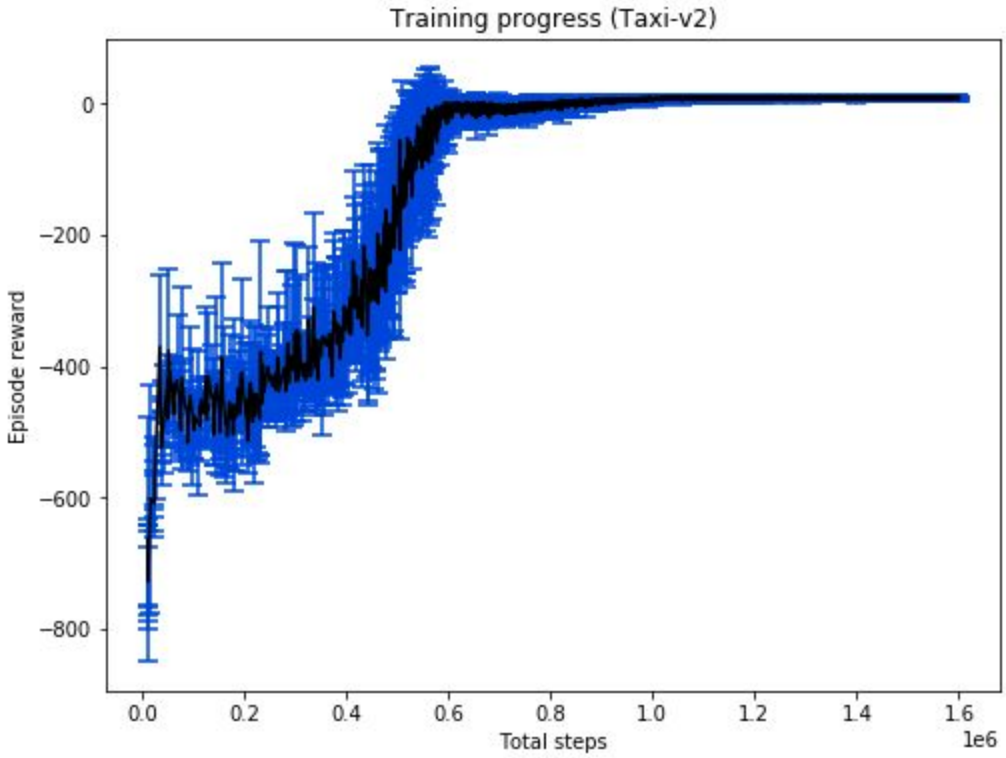
value_coeff = 0.5
lambd = 0.95

As policy gradient methods learn on-policy, they require many more steps as opposed to the DQN (typically they do not reuse previous samples). However, since the DQN ran for 50k steps, it effectively learned from (50k - 10k) * 256 = 10,240,000 samples. We were able to learn a sufficiently good policy using ~1M samples meaning that A2C resulted in a more computationally efficient approach.

In order to converge we ran 64 agents in parallel, each unrolled 64 steps before bootstrapping the value $L = \log \pi_t A_t = \log \pi_t (\sum_{\tau=t}^{t+T} \gamma^{\tau-t} r_\tau + \gamma^{t+T} V^\pi(s_{t+T}) - V^\pi(s_t))$. We used an entropy which linearly decays over 1M steps from a starting value of 10 to 0. In addition, we used GAE with lambda = 0.95 and the learning rate coefficient of the critic set to 0.5.

The most crucial parameters were the entropy + number of parallel environments + rollout length. Smaller entropy or less samples to optimize over led to the agent converging quickly to a

one-hot representation in which it does not use pickup / dropoff at all (since this is a sparse signal, it initially learns to avoid it at all costs). Once the output is a 1-hot representation, the agent is unable to learn anymore since the gradients through the softmax are 0.



Evaluation over 1000 runs resulted in:

Average reward: 8.536
STD: 2.5291706150435957

Acrobot

The state consists of the sin() and cos() of the two rotational joint angles and the joint angular velocities :
[cos(theta1) sin(theta1) cos(theta2) sin(theta2) thetaDot1 thetaDot2].
The action is either applying +1, 0 or -1 torque on the joint between the two pendulum links (e.g. 3 actions).

The reward in this domain is -1 for each step and 0 upon success.

We attempted to solve the domain with both Q learning and Policy Gradient (A2C). In both approaches we made a modification to the basic algorithm in an attempt to improve convergence (final result) and convergence speed, this was motivated by the understanding that in a sparse reward problem it takes a long while to propagate the signal back to the initial state.

Preprocessing of the state:

1) We cut the top 120 pixels of the frame (white space which upon entering the game ends - e.g, irrelevant information).
2) We invert the image (turn black to white). We turn the background from white to black, which reduces computation time.
3) We convert the image to grayscale and downsample it to 40x40 (one channel as it is grayscale).
4) We divide the pixels by 255 such that the input is in the range of [0, 1].
5) We stack 4 frames (similar to the original DQN) in order to capture the dynamics.

DQN

Architecture:

```
DQN_CNN(
  (sequential): Sequential(
    (0): Conv2d(4, 32, kernel_size=(5, 5), stride=(2, 2))
    (1): ReLU()
    (2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): Conv2d(32, 32, kernel_size=(5, 5), stride=(2, 2))
    (4): ReLU()
    (5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Conv2d(32, 32, kernel_size=(5, 5), stride=(2, 2))
    (7): ReLU()
    (8): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (9): Flatten()
    (10): ReLU()
    (11): Linear(in_features=128, out_features=3, bias=True)
```

)
)

Batch normalization:
We found that using batch normalization helped stabilize the training. While in general batch norm is not used in RL, we found that in this task and given our parameters, it worked well.

Double Q Learning:
DDQN samples the action from the active network and uses it for selecting the bootstrap value from the target network. $V(s';\theta_{target}) = Q(s', argmax_{a'}Q(s',\ a';\theta)\ ;\theta_{target})$.

Success experience replay:
The default experience replay stores all transitions. However, in such problems, the meaningful trajectories - those which lead to solving the task - are rare. We observe that as these are rare, with high probability the agent will only learn from transitions far from the goal which in turn will take longer to converge.

The success experience replay is our approach to overcoming this issue. It operates similarly to the standard experience replay, however it only stores trajectories which led to successfully solving the task. The size of the memory is set to 25% of the regular memory. During training we sample w.p. Success_prob from the success memory and from the regular otherwise.

```
# General parameters.
mem_capacity = 200000
batch = 128
lr = 2.5e-4
gamma = 0.99
num_steps = int(1e7)

# Double Q learning.
double_dqn = True

# How often, in steps, to update the target network.
target_update_freq = 500

# Number of random steps, to gather training data, before learning begins.
learn_start = 0 #10000

# Number of previous frames to stack in order to obtain a markovian representation.
hist_len = 4
```

# How often, in steps, to evaluate the network, and for how many episodes.
eval_freq = 10000
eval_episodes = 10

# Decay exploration coefficient linearly over eps_decay steps, down to the value of eps_end.
eps_decay = 1e6
eps_end = 0.2

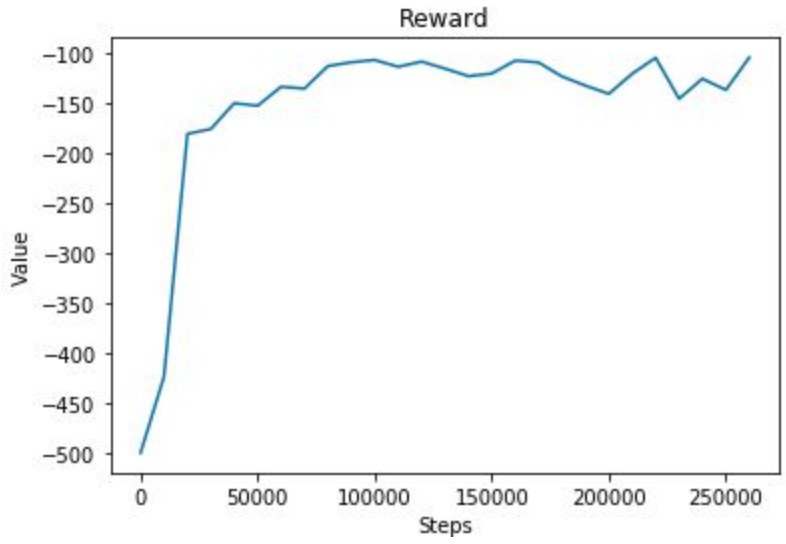
# Success replay memory parameters
success_sample_probability_start = 0.5
success_sample_probability_decay = 1e6
success_sample_probability_end = 0.2
success_traj_length = 200

# Batch norm
batch_norm = True

Total network parameters:
55,075

The results we obtained were:
Average reward: -147.68
STD: 33.334030659372715

Video is provided in the repository.



Policy Gradient

Our architecture:
A shared feature extractor which consists of 3 convolutional layers, followed by a value and policy projection. The value and policy projection are each 2 fully connected layers.

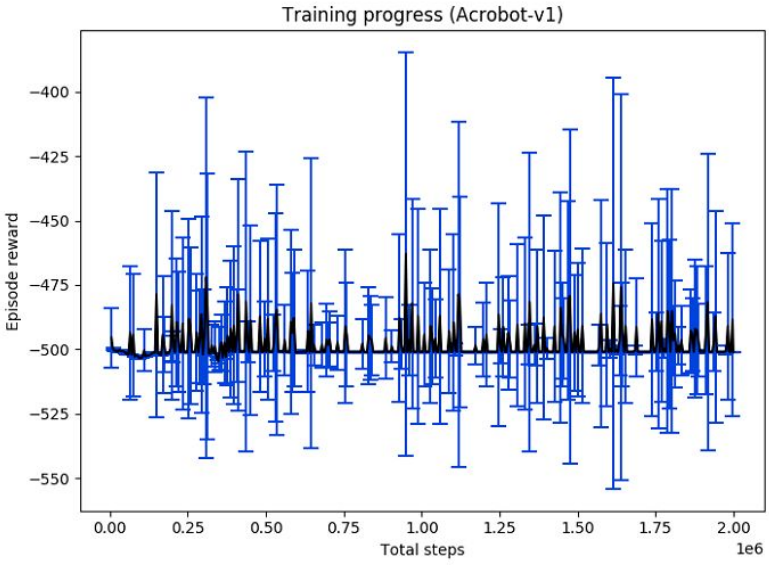
Self imitation learning
The reward in this domain is sparse and delayed. It takes many time steps in order for the agent to successfully solve the task. This is a big issue in both Q learning and policy gradients.
In an attempt to overcome this issue we implemented self imitation learning (Oh et. al. 2018 - https://arxiv.org/abs/1806.05635). Self imitation makes use of a replay memory to reinforce samples which improve our policy.

$$\mathcal{L}^{sil} = \mathbb{E}_{s,a,R\in\mathcal{D}}\left[\mathcal{L}^{sil}_{policy} + \beta^{sil}\mathcal{L}^{sil}_{value}\right] \quad (1)$$

$$\mathcal{L}^{sil}_{policy} = -\log\pi_\theta(a|s)\,(R - V_\theta(s))_+ \quad (2)$$

$$\mathcal{L}^{sil}_{value} = \frac{1}{2}\left\|(R - V_\theta(s))_+\right\|^2 \quad (3)$$

e.g. only when the experienced return $R = r_t + \gamma V_\theta(s')$ is larger than the expectation $V_\theta(s)$ do we use this sample as a reinforcement signal.



As can be seen, even with self imitation learning, the agent is unable to converge. We do not conclude that policy gradient is inferior in this domain, however, we did not have the time or compute in order to perform proper hyper-parameter tuning to compare both methods properly.