Deep Reinforcement Learning – DQN & Policy Gradient

Tal Daniel	taldanielm@campus.technion.ac.il
Mohammed Dabbah	mdabbah@campus.technion.ac.il

GitHub Link: https://github.com/taldatech/pytorch-dqn-policy-gradient

YouTube Link: https://youtu.be/Bm1wXu8YBCY

In this writeup we summarize our experience with DQN and Policy Gradients methods mainly in two OpenAI's Gym environments: Taxi-v2 (Toy Text) and Acrobot v1 (Classic Control). According to OpenAI's official docs (https://github.com/openai/gym/wiki/Leaderboard#acrobot-v1), Acrobot *is an unsolved environment, which means it does not have a specified reward threshold at which it's considered solved.*

Hardware & Software:

- 16GB RAM
- GPU: Nvidia 1050 GTX
- Anaconda with Python 3.5.5
- PyTorch 0.4.1

Implementation Guidelines

We now give a rough explanation of how we designed our models. We wanted to make our models as robust as possible, that is, it would be very easy to create agents for different environments and allow users to easily tune the parameters. Our DQN-based agents include different options, such as:

- Size of hidden layer: users can create agents with different number of hidden neurons (Taxi).
- Regularization and Performance: users can choose whether to apply:
    - Gradient Clipping
    - Batch Normalization
    - Dropout
    - L1 Regularization
- Optimization methods:
    - RMSprop
    - Adam
    - Nesterov Momentum
- Loss functions:
    - MSE
    - Huber (Smooth L1)
- Different CNN architectures (Acrobot):
    - DeepMind's architecture for Atari games
    - PyTorch's sample architecture for Classic Control problems
- Saving and Restoring: allowing to save checkpoint of the agent's learning status (DQNs parameters, optimizer's status, statistics of the agent…) and easily restoring them for playing the environment and continual learning.

Solving the Taxi-v2 Environment

Environment specs

Observations:

There are 500 discrete states (25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is the taxi), and 4 destination locations $25 * 5 * 4 = 500$) .

Actions:

There are 6 discrete deterministic actions:

- 0 – move south
- 1 – move north
- 2 – move east
- 3 – move west
- 4 – pickup passenger
- 5 – drop off passenger

Rewards:

-1 for each action and +20 for delivering the passenger. -10 for executing “pickup” and “drop off” illegally.

Rendering:

- Blue – passenger
- Magenta – destination
- Yellow – empty taxi
- Letters – locations

DQN Approach

Choosing the size of the hidden layer:

We experimented with different sizes, varying between 50-250, and we found 150 hidden neurons to satisfying. There is an obvious trade-off: larger layer will have much more expressive power but would be more complex (that is, more parameters to keep in memory and to tune). According to “The Universal Approximation Theorem for a Single Hidden Layer”, any function can be approximated by a non-polynomial activation function (which means we can increase the layer’s size until we get an accurate approximation).

State encoding:

We tried three different encodings for the states:

- One-Hot Vector – since there are 500 state, each state is represented as a one-dimensional array of size 500. For example, the state 40 is represented as a vector where there is one at the 39th position, and rest are zeros. Thus, the DQN input dimension is 500 (output is 6, as the number of actions).

- Location Tuple – each state is represented as a tuple in the following form: (Taxi Row, Taxi Column, Passenger Location, Destination Index). The board is 5x5, and there are 5 possible locations for the passenger and 4 possible destinations, and so the possible values are: (0-4, 0-4, 0-4, 0-3). Thus, the DQN input dimension is 4 (output is the same, 6 as the number of dimensions). **Got stuck at a local minima.**
- State Integer – one integer (0-499) to represent the state. The DQN input dimension is 1 (output is the same as above). **Got stuck at a local minima.**
- Location Tuple One-Hot – a combination of the first two models. We encode each cell of the tuple as one-hot and concatenate to a binary vector of size 19. Great performance, less complexity, model is x100 smaller in size.

Models summary:

General parameters:

- Exploration scheduling: Exponential (decay of 800 episodes)
- Batch size: 128
- Hidden Neurons: 150
- Loss function: MSE

Model	State Encoding	Optimizer	Learning Rate	Best Mean (100) Reward	Pretrained Filename	Regularization
1	One-Hot	RMSprop	0.0003	7.1	taxi_agent_pretrain.pth	None
2	One-Hot	RMSprop	0.0003	7.01	taxi_agent_one_hot_dropout.pth	Dropout (p=0.4)
3	One-Hot	RMSprop	0.00025	-227	taxi_agent_one_hot_l1.pth	L1 (lambda=0.6)
4	Location Tuple One-Hot	RMSprop	0.0003	7.8	taxi_agent_location_one_hots.pth	None
5	One-Hot	Adam	0.0003	6.98	taxi_agent_one_hot_adam.pth	None
6	One-Hot	Nesterov Momentum (momentum =0.9)	0.0003	-4	taxi_agent_one_hot_nesterov.pth	None

Training results:

We separate the results for each model and detail its parameters and hyper-parameters.

Model 1:

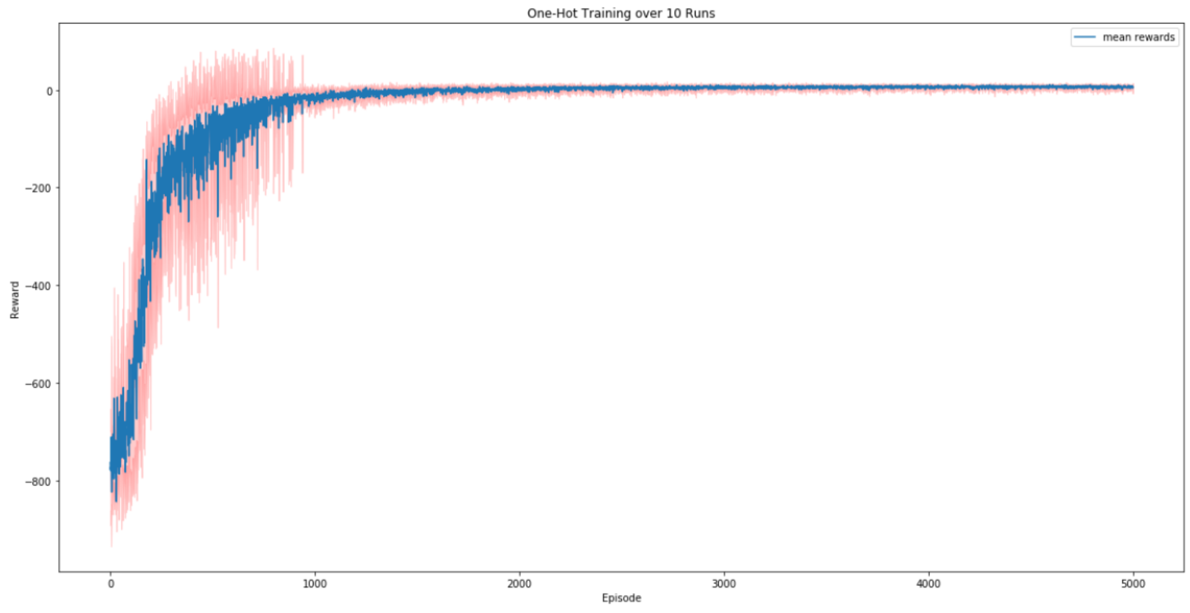
- Note – we trained 10 models of this type in order to observer the training variance.

Insights:

- Optimizer – little difference between Adam and RMSprop with a slight advantage to RMSprop. We stick with this one for the rest of our models.
- Hidden Neurons – we found 150 hidden units to be enough. More resulted in roughly the same results, but the model was more complex which led to longer training and larger size on disk. We also stick with this number for the rest of the models (Though for models that got suck at local minima, we tried different values but were not able to overcome this problem).
- Loss – as we are trying to predict continuous values, MSE loss seemed like the best option and also performed as expected.
- Regularization – none for this model.
- Size on disk – 610 kb

Variance of 10 training sessions: 18665.125

The results (the shaded red area is one STD from the mean):

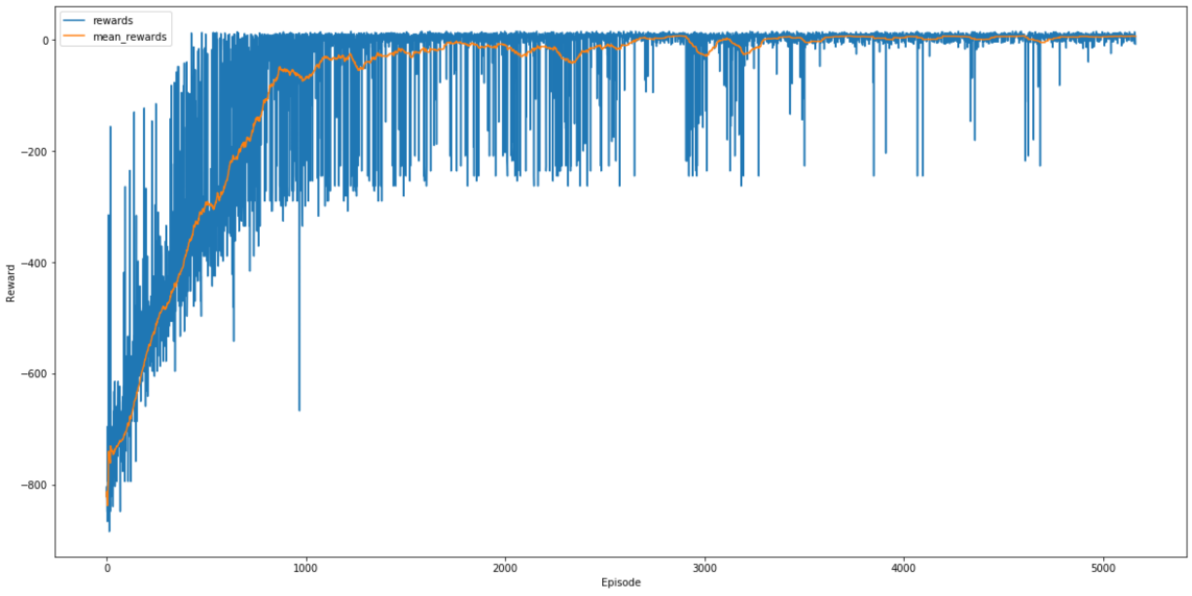


It can be seen that the model converges, and also the variance (standard deviation) also decreases as we reach the optimum.

Model 2:

We now apply Dropout (as a regularization method) on the training network (the second network was set to evaluation method, thus the dropout probability is 0).

The results:



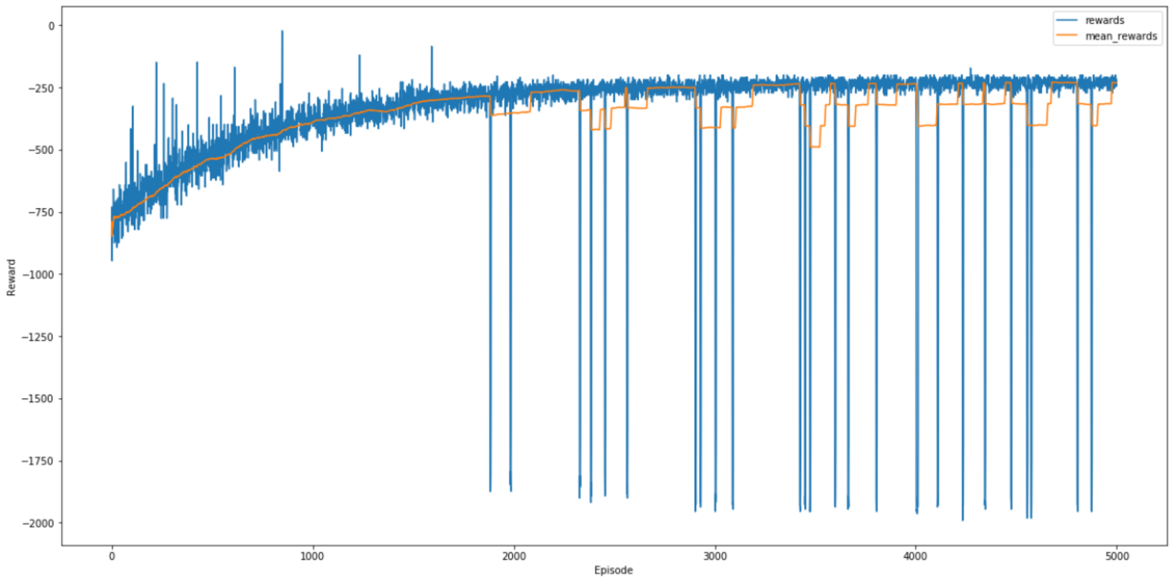
Insights:

- The model reached the optimum, though at a slower rate than without dropout.
- Because of the dropping of layers, we see these spikes in the rewards.
- Dropout is a regularization method to keep the model from overfitting in supervised learning. In this RL problem, we don't face this issue, thus it is not a common practice (unlike Computer Vision tasks).

<u>Model 3:</u>

We now apply L1 regularization on the weights, with $\lambda = 0.6$. It is worth mentioning that this type of regularization is meant to keep the weights at a certain scale.

The results:



Insights:

- The training was much slower, each training step took more than without applying the L1 regularization.
- The model got stuck at a local minima (we assume). Nevertheless, we tried different parameters but ended up with the same results (around the same mean reward).
- Both Dropout and L1 are regularization methods intended to avoid overfitting, but such symptom is of no concern in this particular RL problem. Moreover, these techniques are not common practices in deep RL.

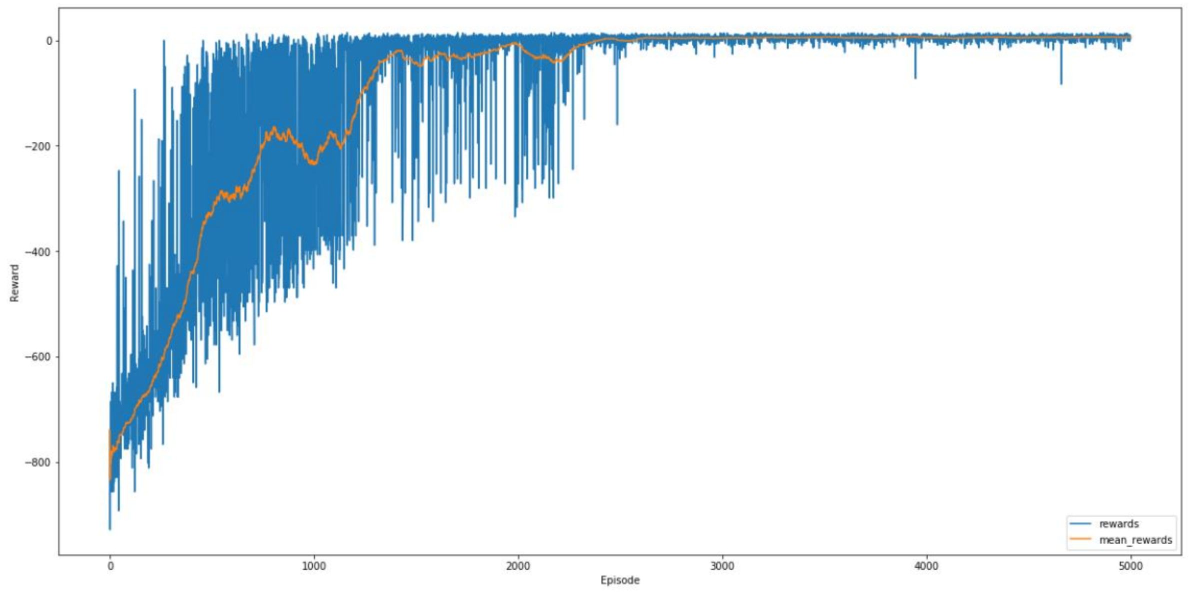
Model 4

Here lies our “state-of-the-art”. At first, we thought of another way that the state can be represented and we even found one which is officially supported by the environment. We call it “Location Tuple Representation” as the state is a tuple of the form (Taxi Row, Taxi Column, Passenger Location, Destination Index) which can take the following values:

- Taxi Row – the board is 5x5, thus the rows are indexed 0-4.
- Taxi Column - the board is 5x5, thus the columns are indexed 0-4.
- Passenger Location – can be located at one of the four corners, and inside the taxi, thus we have 5 possible locations indexed 0-4.
- Destination Index – can be located at one of the four corners, thus indexed 0-3.

The first try was taking this 4-tuple, normalizing it and using it as the input to our DQNs. We tried many hyper-parameters variations but always ended up at a local minima. Then, we thought of combining this with One-Hot representation such that every cell in the tuple is converted to a One-Hot-Vector and then concatenating it to one vector of length 19. For example, the state (3,2,4,1) is encoded as [00010 00100 00001 0100]. Moving for a 500-length vector to 19-length vector has a great impact on the model as we will see.

The results:



Insights:

- Much faster training time.
- Convergence came at a later point than in the original One-Hot.
- Some spikes closer to convergence (probably need more exploration than the original One-Hots).
- Size on disk: 33 kb (x20 smaller than the original One-Hot).
- Complexity: there are much less parameters and thus less multiplications.

Optimizers comparison

We focus on 3 different optimizers:

1. RMSprop – dividing the gradient by a running average of its recent magnitude. Keeping a moving average of the squared gradient for each weight. It originated from the scale problem which causes vanishing/exploding gradients. It is said that in comparison to Adam, RMSprop is well-suited to handling non-stationary environments (problems where the independence -IID assumption does not hold, like in RL). Suggested values are $\gamma = 0.9, \eta = 0.001$.
2. Adam – Adam stands for Adaptive Moment Estimation, which can be thought of a combination between RMSprop and momentum. It keeps an exponentially decaying average of past squared gradients (like in RMSprop) and also an exponentially decaying average of past gradients (like in Momentum). It is known as computationally efficient with little memory requirements and well-suited for problems with large data and parameters. In practice, its hyper-parameters require little tuning and it was found appropriate for non-stationary objectives (https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/).
3. Nesterov Accelerated Gradient (NAG) – Yuri Nesterov observed that the value of regular SGD with momentum was still high when the gradient reaches local minima (which may lead to overshooting). In this version the gradients are calculated after the momentum update.

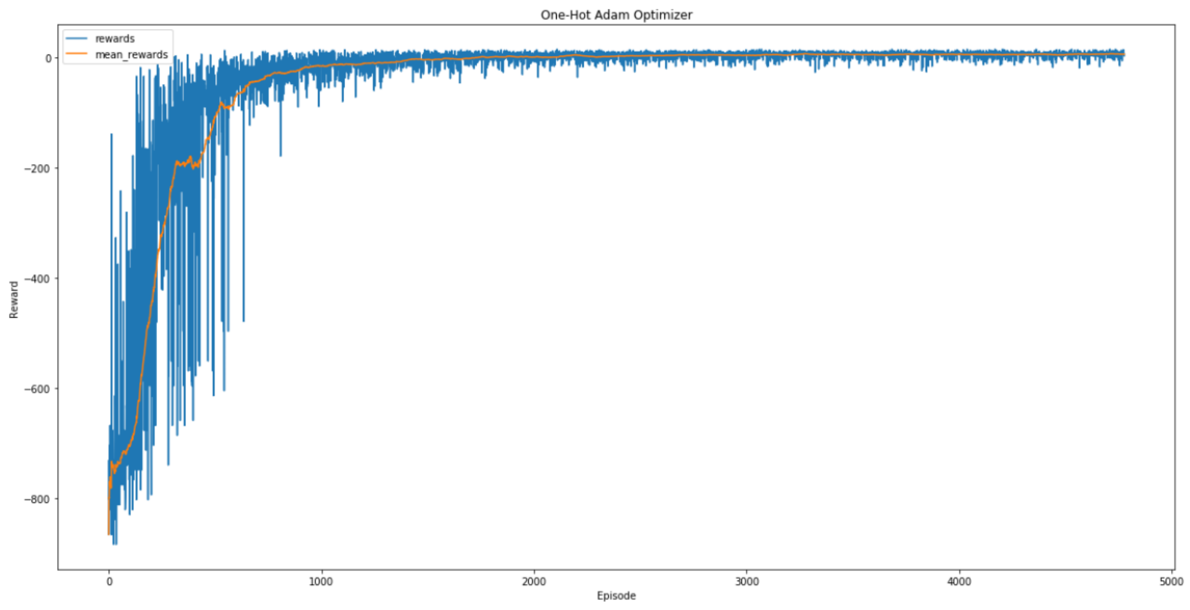
For RL problems, it seems that RMSprop and Adam are best fitted, since they can handle non-stationary environments. We will compare them using models 5-6.

Through this project we used the RMSprop, and the results for this optimizer are depicted in Model 1.

Model 5

We use Adam optimizer while the rest of the parameters stay the same.

The results:



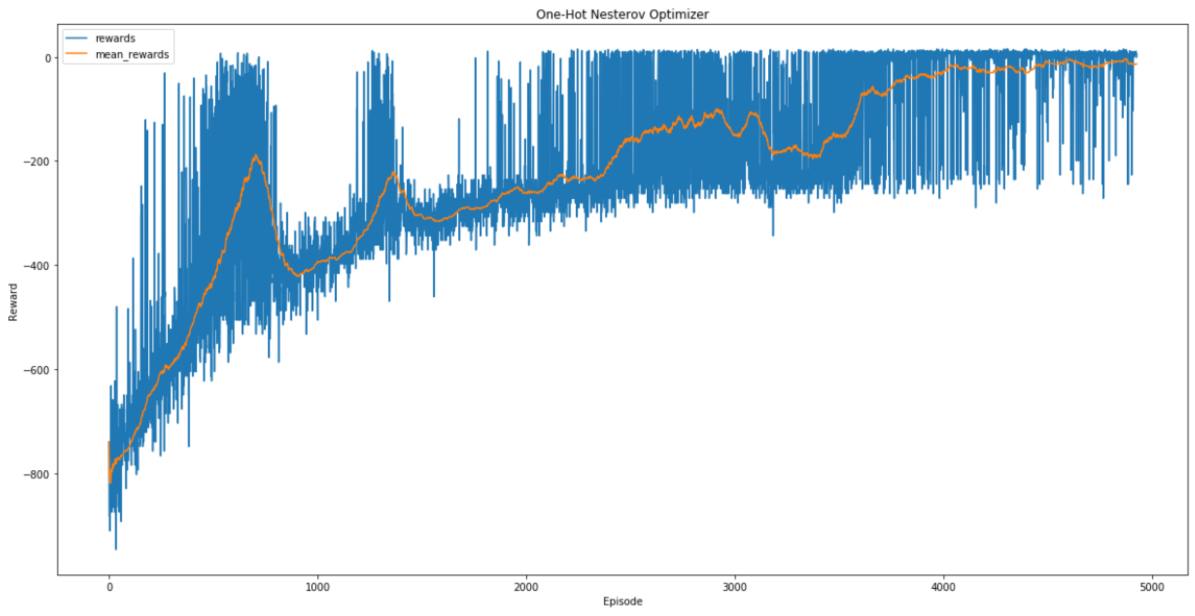
Insights:

- As expected, convergence achieved at approximately the same rate.
- Training time took a little longer. RMSprop is the favorable optimizer between the two for performance reasons.

Model 6

This time we used Nesterov Momentum optimizer.

The results:



Insights:

- As anticipated, the results are not satisfying. Convergence is slow, noisy and unstable.
- Nesterov Momentum (and even regular Momentum SGD) are not well fitted for this task and require a lot of tuning to get the convergence right, unlike RMSprop and Adam.

Policy Gradient Approach

The policy is represented by a neural network and the loss is calculated by the Policy Gradient theorem.

$$\nabla_\theta J_\pi(\theta) = \sum_{i=0}^{N}\sum_{t=0}^{T} \nabla_\theta log\pi\left(a_t^i \,|s_t^i, \theta\right) \cdot Q_\pi$$

We have experimented with many approaches such as changing the neural network architecture, adding exploration and using variance reduction approaches.

We started out with vanilla PG with no exploration, based on an NN architecture with 1024 hidden neurons that takes a one-hot vector of size 500 that represents the state where the output is the policy distribution from which we sample the action from (softmax of a vector of size 6). The policy parameters quickly stabilized on choosing just navigational actions due to the reward structure (-1 for each action and +20 for delivering the passenger, -10 for executing “pickup” and “drop off” illegally). With final reward requiring too much steps and pick up and drop off actions being too destructive if executed incorrectly, it was reasonable to see the policy getting stuck of a local minima where it preferred to do only navigational actions. We saw a faster convergence as we increased the number of hidden neurons from 128 to 1024,. We have also tried stacking 2 hidden layers (shapes 500x2048, 2048x1024) where we observed even faster convergence, but still to the local minima we described.

Random Exploration:

To solve the local minima problem, we decided to implement epsilon greedy strategy where the action chosen was sampled from a uniform distribution if we were to choose exploration. This approach did not work well too, possibly because the random actions only added noise instead of helping the agent learn to use the pick up and drop off actions wisely.

Smart Exploration:

Next, we tried to encode prior knowledge about the problem in our exploration, due to the nature of the problem we could easily write a policy that chooses the action in a deterministic way given the state. The state is represented by a number from 0 to 499 that represents the location of the taxi, location of the goal destination and the location of the passenger (including the case if he’s in the taxi). Such deterministic policy can be described by the following pseudo code: (full python code is attached in submission)

```
taxirow, taxicol, passloc, destidx = decode(state)

if passloc == 4:
    # passenger is in the taxi, we go straight to destination |action chosen includes drop_off
    return go_to_location(taxirow, taxicol, destidx, have_passenger=True)
else:
    # we go to passenger location |action chosen includes pickup
    return go_to_location(taxirow, taxicol, passloc, have_passenger=False)
```

Thus, whenever greedy exploration chooses to explore (depends on sampling a number smaller than exploration factor epsilon) we choose the correct action to do according to the state.

We've tried different exploration factors and there was a trade of to be made:
With high exploration factor, the action chosen was more likely to be from the coded fixed policy and the network parameters would adjust poorly because in order for the PG theorem to work, the actions preformed in an episode should be sampled from the distribution given by the policy π itself (that's why PG method does not inherently support epsilon greedy exploration, the action needs to be polled from π itself), but with low exploration factor the agent would get stuck in the local minima we described earlier.
Best results were obtained from starting with an exploration factor of 0.1 and diminishing it by 0.99 every 2000 episodes so the leaned policy would be less reliant on the coded fixed policy as the training continues.

Baseline Reduction and Actor Critic:

With the introduction of exploration we noticed that although the agent started to solve the problem, the policy loss given by $\sum_{t=0}^{T} log\pi(a_t \,|s_t, \theta) \cdot Q_\pi$ did not converge to 0 no matter the how much episodes we trained on. This could be because PG method does not support epsilon greedy exploration as we explained earlier. To solve this issue we tried to incorporate a baseline reduction variant, namely Actor Critic method for variance reduction (this could be thought of as a kind of regularization for the PG approach).

$$\nabla_\theta J_\pi(\theta) = \sum_{i=0}^{N}\sum_{t=0}^{T} \nabla_\theta log\pi\left(a_t^i \,|s_t^i, \theta\right) \cdot (Q_\pi - V_\pi) = \sum_{i=0}^{N}\sum_{t=0}^{T} \nabla_\theta log\pi\left(a_t^i \,|s_t^i, \theta\right) \cdot A_\pi$$

To implement the actor critic we modified the network to have "two heads": one head takes the hidden layer, passes it through an FC layer to output the policy distribution (same as before), and the second one takes the hidden layer and passes it through an FC layer to output the value function which we use in the final policy loss as an approximation of V_π, and now the network policy has another term that trains the second head to approximate V_π as the cumulative discounted reward of the episode.

Results:

From all the variants that we tried we report a comparison between the following two models:

Agent 128:
Hidden layer size: 128 neurons,
Uses diminishing exploration from coded fixed policy: True
Exploration parameters: epsilon 0.1, diminishing factor 0.99 every 2000 episodes
Training period: 100,000 episodes
Uses Actor Critic: True

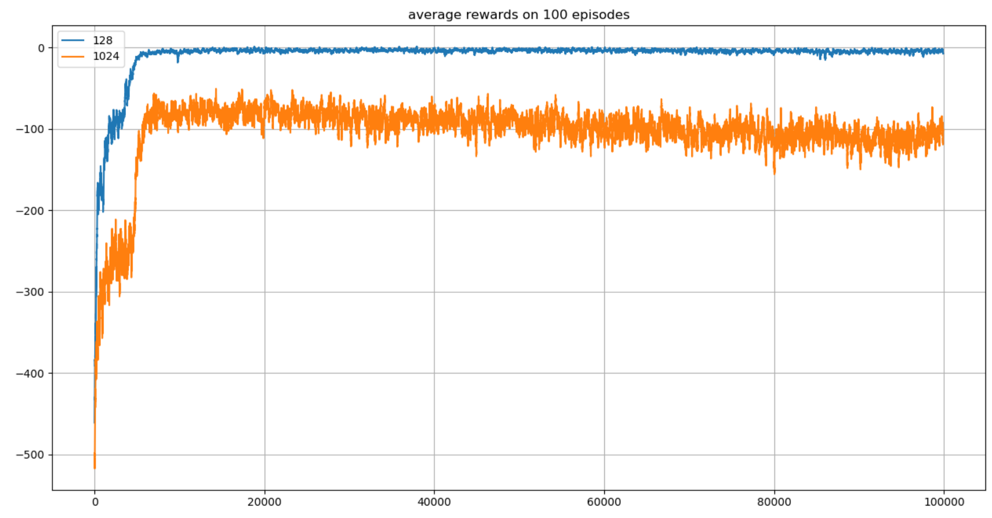
Agent 1024:
Hidden layer size: 1024 neurons,
Uses diminishing exploration from coded fixed policy: True
Exploration parameters: epsilon 0.1, diminishing factor 0.99 every 2000 episodes

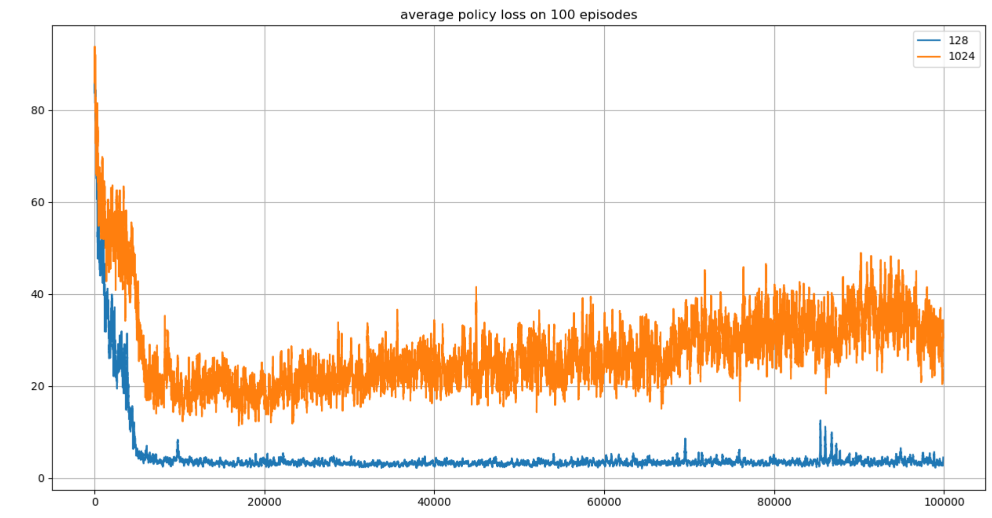
<u>Training period:</u> 100,000 episodes
<u>Uses Actor Critic:</u> True

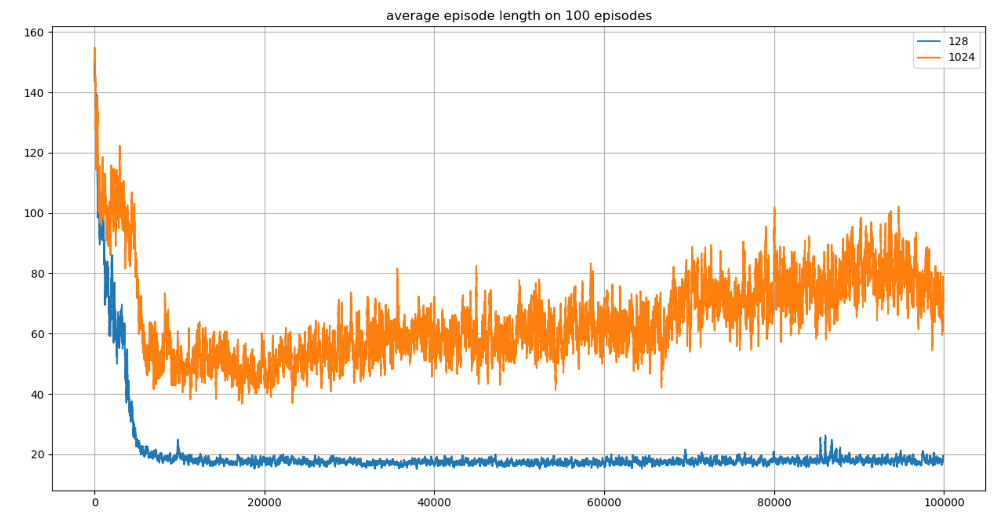
We compared the agents by the following 3 criteria:

- Moving Average reward
- Moving average on the network loss
- Moving average on episode length to see how fast the agent solves the puzzle

Note: we use moving average of window size 100 episodes to see the overall trend the agent has as it trains. We're less interested in those statistics on a single episode.







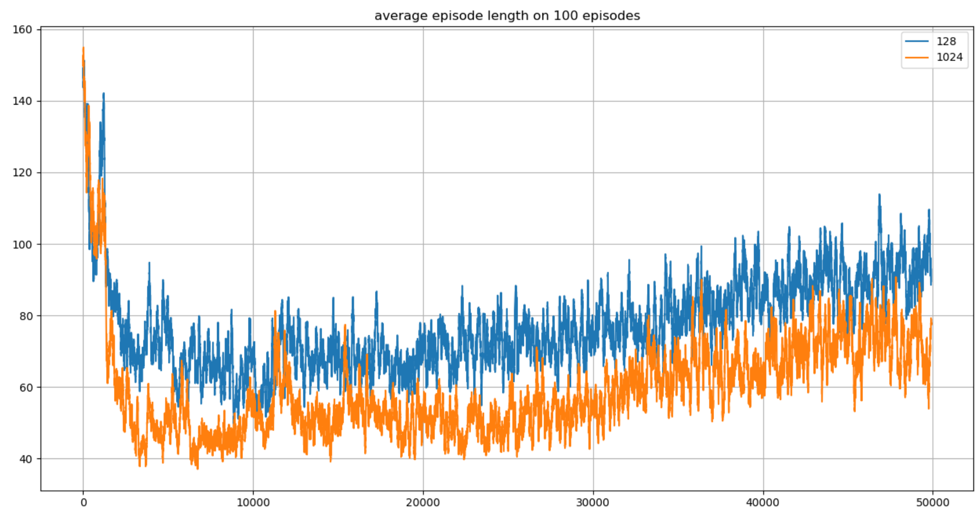
From the graphs we see that the 128 neurons model won the contest with the 3 criteria and seems to have converged, while the 1024 model seems to be on an upward trend to diverge.

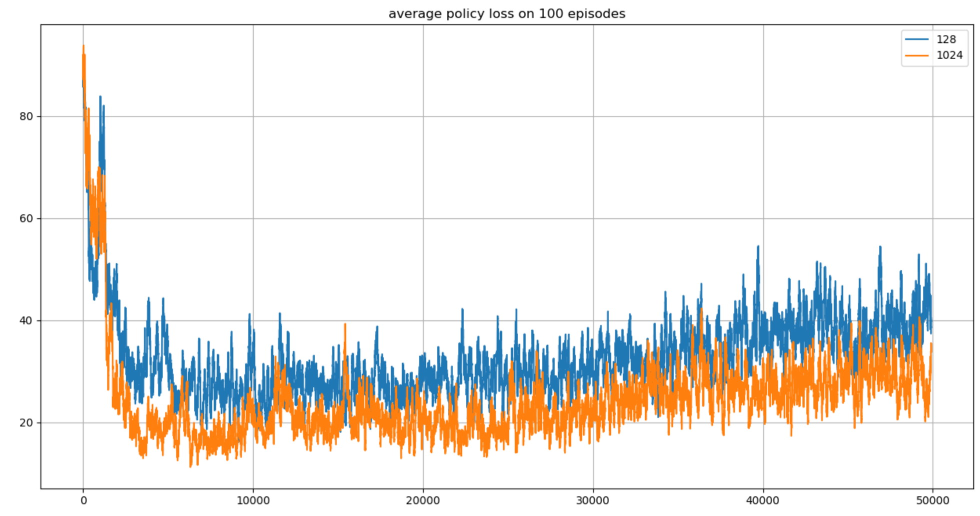
With that being said, when training with slightly different parameters we see 1024 model being superior.

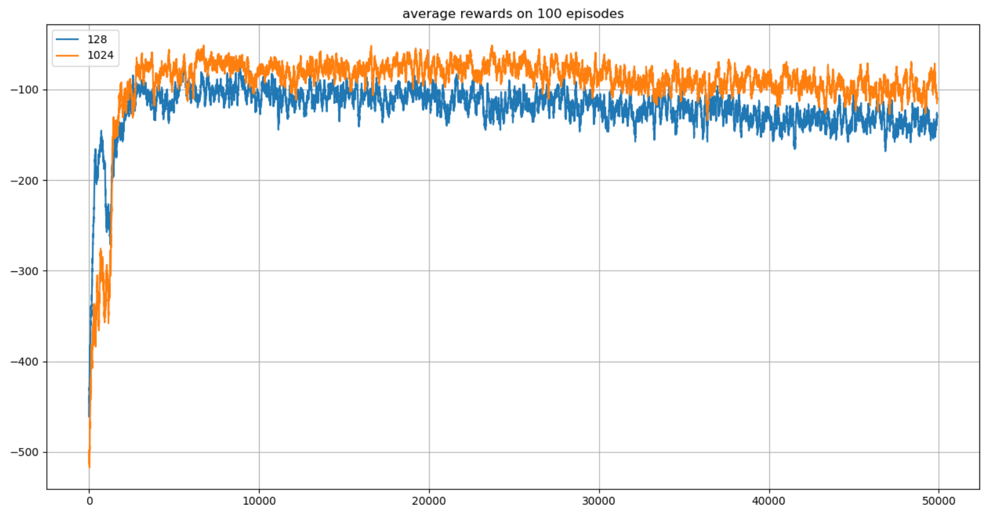
Following is a comparison between the same agents but with slightly different exploration parameters than above:

Agent 128:
Hidden layer size: 128 neurons,
Uses diminishing exploration from coded fixed policy: True
Exploration parameters: epsilon 0.1, diminishing factor 0.99 every 1000 episodes
Training period: 50,000 episodes
Uses Actor Critic: True

Agent 1024:
Hidden layer size: 1024 neurons,
Uses diminishing exploration from coded fixed policy: True
Exploration parameters: epsilon 0.1, diminishing factor 0.99 every 1000 episodes
Training period: 50,000 episodes
Uses Actor Critic: True


average episode length on 100 episodes
128
1024
160
140
120
100
80
60
40
0
10000
20000
30000
40000
50000


average policy loss on 100 episodes
128
1024
80
60
40
20
0
10000
20000
30000
40000
50000


average rewards on 100 episodes
128
1024
−100
−200
−300
−400
−500
0
10000
20000
30000
40000
50000

Here we see that 1024 neuron model was a bit more stable and got better results before they both diverged.

Conclusions:

- The reward structure of the given problem is problematic, and it was impossible to train without some kind of exploration (exploration was a must).
- Policy gradient methods do not support epsilon greedy exploration natively and training a PG agent with exploration is highly unstable and a slight change with the exploration parameters preferred different network architectures w.r.t the tested criteria
- The best results were obtained from the following agent:
  Agent 128:
  Hidden layer size: 128 neurons,
  Uses diminishing exploration from coded fixed policy: True
  Exploration parameters: epsilon 0.1, diminishing factor 0.99 every 2000 episodes
  Training period: 100,000 episodes
  Uses Actor Critic: True
- Actor Critic base reduction is good 😊

Solving the Acrobot v-1 Environment

Acrobot is a 2-link pendulum with only the second joint actuated.

Initially, both links point downwards. The goal is to swing the end-effector at a height at least the length of one link above the base. Both links can swing freely and can pass by each other, i.e., they don't collide when they have the same angle.

Environment specs

Observations:

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].

For the first link, an angle of 0 corresponds to the link pointing downwards. The angle of the second link is relative to the angle of the first link. An angle of 0 corresponds to having the same angle between the two links. A state of [1, 0, 1, 0, ..., ...] means that both links point downwards.
We will use a current RGB array snapshot of the environment as the state.

Actions:

There are 3 discrete deterministic actions:

- 0 – apply +1 on the torque
- 1 – apply 0 on the torque
- 2 – apply -1 on the torque

Rewards:

-1 for each action until reached -500 (500 steps) or termination when the bottom pendulum link reached above the base.

Rendering:

A 500x500 pixels RGB arrays.

We tried different approaches for the Acrobot-v1 environment:

1. Frame Difference – as suggested by PyTorch tutorials for Classic Control problems.
2. Frame Sequence – as suggested by DeepMind's approach to Atari games.

Note: we save frames in the Replay Memory as UINT8 types, as it is much more memory efficient. When we draw samples, we convert to FLOAT32 to do more operations (like normalizing).

Preprocessing:

The general target was to resize the frames to be 84x84, since it is much more efficient when run on GPU (matrix multiplications as a result of the convolution operations).

1. Frame Difference – frames were resized to 84x84 while keeping the 3 color channels. The resulting state is a (3x84x84) array representing the difference between two consecutive frames.

2. Frame Sequence – the three channels were averaged to 1 channel such that the gray image had brightness and contrast features that were distinguishable. We then resize the frames to 84x84 and lastly, frames were collected to a sequence of 4 frames, similar to the approach used by DeepMind for Atari games.

CNN Architecture:

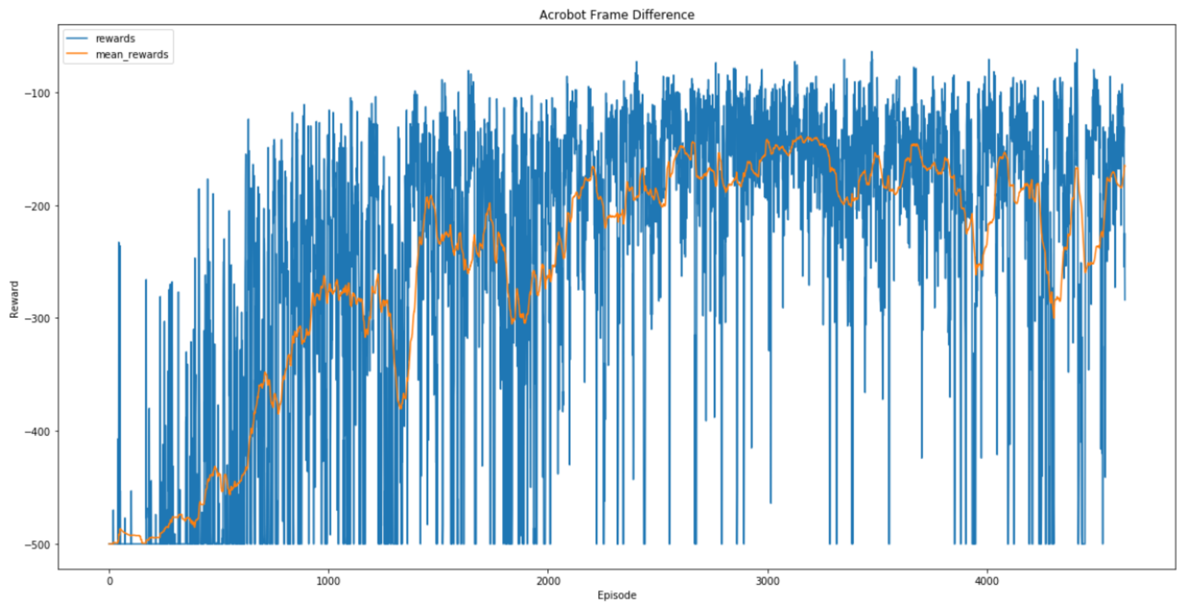
1. Frame Difference – similar to PyTorch tutorial on RL architecture, there are 3 Convolutional layers (with Batch Normalization applied) and 1 Fully Connected layer.
2. Frame Sequence – similar to DeepMind architecture for Atari games, there are 3 Convolutional layers and 2 Fully Connected Layers.

Regularization: we used Clipping Gradients ([-1,1]) to avoid exploding gradients symptom.

Hyper-Parameters:

- Optimizer: RMSprop
- Learning Rate: 0.0003
- Loss function: Huber (L1 Smooth Loss)

Frame Difference model results:



As expected, the training process was very noisy, but we achieved somewhat good mean rewards. This environment is considered unsolved, and solving visually using frames as states is not an easy task. Training the Frame Difference model took about 2 days on the GPU, and we assume that given more time and exploration we could reach an even better policy.

Unfortunately we were not able to train an optimal model using the Frame Sequence approach. We considered several reasons for that:

- In contrast to Atari games, the Acrobot problem has 2 main colors, and since the pendulum links are of the same color, when they overlap, the CNN has hard time extracting meaningful features from it. We thought of applying some Computer Vision methods like edge detection and using the image's gradients, but we observed heavy impact on the performance with practically almost the same results.
- In contrast to Atari games, 2 consecutive frames contain a lot of information, and thus using a sequence of more than 2 frames as input may harm the CNN's ability to learn by increasing the state/observation space which leads to the need of more exploration and results in a much longer training time.

In addition, DeepMind's CNN architecture is more complex and deeper, and also the state representation has its toll (Frame Sequence model weighs 13MB in comparison to the 350KB Frame Difference model).