# Analysis of Proximal Policy Optimization Algorithms – Midterm Report

Achyutha Sreenivasa Bharadwaj (1213094192) and Ninad Jadhav (1213245837)

***Abstract*— In this project, we are analyzing the reinforced learning algorithm proximal policy approximation that has been introduced in the paper [4]. To carry an in-depth analysis of the algorithm, we started with collecting baseline data for comparison with the algorithm. Also we have explored some techniques for preprocessing the input image (Atari game – MS-Pacman) for faster training of our neural networks. We also highlight some of the challenges we faced while setting up the required environment (OpenAI gym environments) on different OS platforms.**

***Index Terms*—reinforcement learning, observation, state, reward, history, environment.**

## I. Introduction

Using deep learning methods AI can almost learn anything that can be modeled or simulated. In this project we will try to construct an agent that will learn how to play Pacman game on its own using Reinforcement learning strategy. The idea of the game Pacman is to make the agent eat as many dots as possible and to keep the agent from getting killed by any of the monsters by moving it up, down, left, right, etc. When all the dots are eaten the level is cleared and the agent advances to the next level. Total score gained by the agent is computed by the number of dots eaten. Our goal is to create an AI agent that will learn how to move around in this environment without getting killed and achieve maximum high score.

There are many environments available for creating the agent but, for this project will be using the OpenAI gym's ATARI environment. For developing the agent, Python3.5 is used along with Tensorflow, Anaconda, Keras, OpenAI's gym and Tflearn packages. In section II we give stats regarding the baselines by directly running the gym code. In Section III we explain the different strategies that we used for running the simulations on CPU. Section IV explains the algorithmic modifications done and the simulation ran using using it. In the final section V, we give the results of using different models to run the simulations on GPU using Google colaboratory environment.

## II. Testing Baseline Implementation

To get information regarding how the OpenAI Gym model works and get some intuition regarding the behavior of an agent in the game environment, we started with running a simple model based on the example code given in OpenAI gym online documentation. We analyzed the behavior of open source proximal policy optimization baseline implementation provide by Open AI community for the MSPacman-v0 game in Atari environment based on the following parameters –

- Average timeout for *Game Over*
- Max time required before *Game Over*
- Max reward achieved by the *random policy* agent

| | **84X84X1** | **210X160X3** |
|---|---|---|
| EpLenMean | 213 | 199 |
| EpRewMean | 21.4 | 152.0 |
| EpisodesSoFar | 4603 | 564 |
| TimeElapsed | 5.6e+04 | 2.98e+03 |
| TimeStepsSoFar | 960043 | 111309 |
| Loss_KL | 0.034473605 | 1.1600983e-06 |
| Loss_vf_loss | 2.140352 | 615.77795 |
| Loss_ent | 0.7003318 | 1.7872881 |
| Loss_pol_entpen | -0.0070033185 | -0.017872881 |
| Loss_pol_surr | 0.007533414 | 0.0013152664 |

Table: 1

As we can see from the above table, the mean reward after over 950 thousand time-steps is just 21.4. Although the baseline implementation could learn better policies for most of the games, when it comes to MSPacman, it clearly couldn't learn a good policy. When we started analyzing the issue, we found that in order to reduce the load, there was a preprocessing step done in the baseline which was cropping the 210X160 resolution input image into a 84X84 image. Even though it made time taken for each episode shorter, it was not able to learn a better policy as a lot of important information was being lost. When we removed this preprocessing, the mean reward increased to 152.0. But, the training speed was too slow. In this paper we will be using this baseline implementation without preprocessing as the benchmark to compare our implementations.

## III. CPU Based Simulations

Due to initial lack of GPU, we trained some models on CPU with constraining the timesteps to 100000 for most of the simulations. We used the starter code of 'ppo1' given in the GYM baseline repository to run the CPU simulations. To start the simulations:

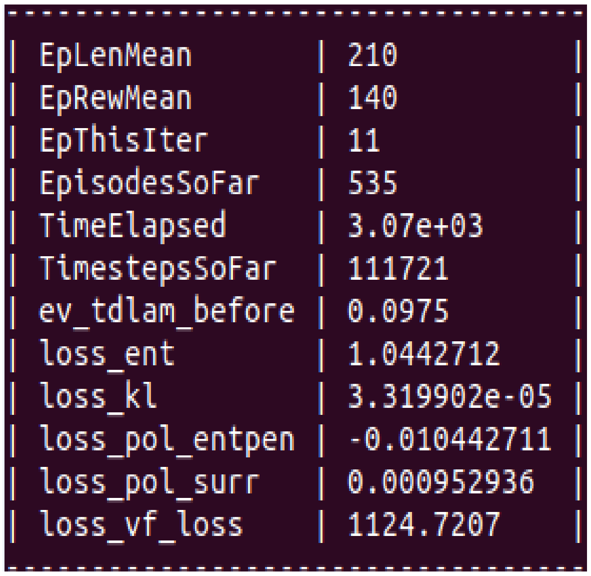
- Download the OpenAI gym baseline code from github
- To run CPU simulations run the following command from the baseline directory:

```
mpirun -np 8 python -m
baselines.ppo1.run_atari -env
MsPacmanNoFrameskip -num-timesteps 100000 -
seed 0
```

,where *seed* is the input to the random state generator of the gym environment. We used seed value of 0, 18 and 1992 during our simulations.

## A. *RGB to GreyScale Conversion:*

To begin the simulations with some image modifications, we started with duplicating one of the wrappers in '*baselines.common.atari_wrappers*' and updated it by keeping the dimensions same as raw image. The images were also converted to grey scale . (Refer '*wrap_RGB_GS(env)*'). The output at the end of the simulation – 100000 timesteps was as follows:



```
----------------------------------
| EpLenMean       | 210          |
| EpRewMean       | 140          |
| EpThisIter      | 11           |
| EpisodesSoFar   | 535          |
| TimeElapsed     | 3.07e+03     |
| TimestepsSoFar  | 111721       |
| ev_tdlam_before | 0.0975       |
| loss_ent        | 1.0442712    |
| loss_kl         | 3.319902e-05 |
| loss_pol_entpen | -0.010442711 |
| loss_pol_surr   | 0.000952936  |
| loss_vf_loss    | 1124.7207    |
----------------------------------
```

Fig : 1

On running the saved model we found that the agent kept on moving towards the lower end of the frame (for default seed value = 0). There was considerable performance difference for different seed values , which lead us to the conclusion that the trained model will give above average result for some specific environments of the game as compared to the random policy.

## B. *Adding a Maxpooling layer in the neural Network*

For faster computation of the raw image which was modified to greyscale as well as to maintain the spatial correlation in the frame, we added a maxpool layer in the neural network model. The model (used by ppo1) has the following layers:

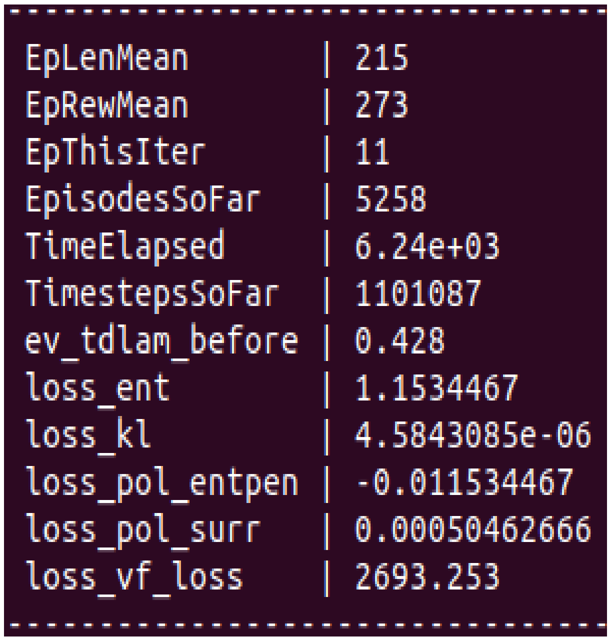
| **Layer** | **Filters** | **Filter Size** |
|---|---|---|
| M1 | - | - |
| C1 | 32 | 1 |
| C2 | 64 | 2 |
| C3 | 64 | 3 |

Table:2

To make modifications to the CNN layers add additional layers in the *baselines.ppo1.cnn_policy* file under *kind == 'large'* condition.

*tf.layers.max_pooling2d( X , 2, 2)*

This simulation was run for 1 million timesteps since we did not see any good results for 100000. The final output at the end of the simulations was as follows:



```
-----------------------------------
EpLenMean       | 215
EpRewMean       | 273
EpThisIter      | 11
EpisodesSoFar   | 5258
TimeElapsed     | 6.24e+03
TimestepsSoFar  | 1101087
ev_tdlam_before | 0.428
loss_ent        | 1.1534467
loss_kl         | 4.5843085e-06
loss_pol_entpen | -0.011534467
loss_pol_surr   | 0.00050462666
loss_vf_loss    | 2693.253
-----------------------------------
```

Fig : 2

Even after training to 1M timesteps there was no considerable improvement in the performance. To see the policy in action, run the following command from baseline directory

```
python -m baselines.ppo1.run_atari -env
MsPacmanNoFrameskip -num-timesteps 0 -seed 0
```

## C. *Clipping parameter modification*

The clipping parameter decides how much to penalize the objective function while learning the policy. All the simulations use a default value of 0.2, so we modified it to see what impact it would have on the policy training. For a clip = 0.4, we observed a very bad policy that did not gain any reward in-spite of there being not much of a change in the value loss. The non – normalized data plot for reward and loss vs 100000 timesteps (keeping all other values same as done in previous iterations) is as follows:

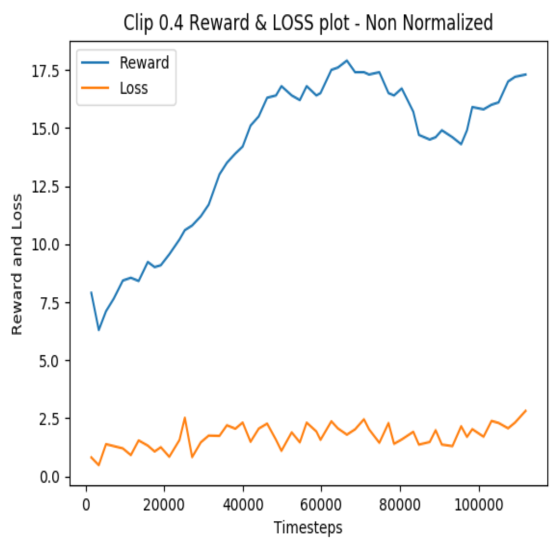


Fig :3

The metric values for 0.4 and 0.6 clip at the end of the simulation are as follows:

| **Metric** | **Clip=0.4** | **Clip=0.6** |
|---|---|---|
| EpLenMean | 232 | 182 |
| EpRewMean | 17.3 | 170 |
| EpThisIter | 14 | 9 |
| EpisodesSoFar | 532 | 615 |
| TimeElapsed | 2.96e+03 | 4.62e+0.3 |
| TimestepsSoFar | 111985 | 111985 |
| ev_tdlam_before | 0.662 | 0.0787 |
| loss_ent | 0.8314841 | 0.0 |
| loss_kl | 6.614765e-05 | 0.0 |
| loss_pol_entpen | -0.0083148405 | 0.0 |
| loss_pol_surr | 0.0130475825 | 6.3695596e-0.5 |
| loss_vf_loss | 2.8206496 | 1603.1711 |

Table : 3

We see that for a clip=0.6, the reward is decent for the given number of timesteps. To modify the clip value in the code , Update '*clip param'* in *pposgd_simple.learn* function call in file *baselines.ppo1.run_atari*

## IV. Random Walk

Often times when the agent is learning how to play the game, it gets struck at some place either due to boundaries or monsters. In such case, the agent will continuously use the same policy and get the same reward each time. Hence, there will not be any progress in the agent's policy for a large number of episodes. This will cause a large delay in the training process and also reduce the quality of the learned policy for a given number of episodes. To avoid this issue, we borrowed the concept of Random Walking from Page Rank algorithms. Consider a graph $G(V, E)$ and an agent is present at a random node in this graph and the agent from the current node, selects a random connected node and moves to that node. The question is, can the agent discover all the nodes in the graph if it keeps doing this random walk? If the graph is fully connected, then the random walk can discover all the nodes of the network. But, if the graph is disconnected, then
the agent will be struck in one of the disconnected parts of the graph and will never be able to discover the rest.

The solution to this problem is that, when the agent is picking a random node to travel to, with certain small probability it will jump to a random node across the entire graph. This way the agent can make random jump to the disconnected parts of the network and discover all the nodes in the graph. The situation of the agent in our case is similar to this. So, what we did was when the agent predicts what action to perform by using the deep network, if with a small probability it chooses to perform a random action. This way, even if the agent got struck at a corner, it has a higher chance of getting unstruck because of the random action.

| | **With RW** | **Without RW** |
|---|---|---|
| EpLenMean | 228 | 199 |
| EpRewMean | 201.0 | 152.0 |
| EpisodesSoFar | 590 | 564 |
| TimeElapsed | 2.99e+03 | 2.98e+03 |
| TimeStepsSoFar | 111786 | 111309 |
| Loss_KL | 6.5872396e-06 | 1.1600983e-06 |
| Loss_vf_loss | 680.9152 | 615.77795 |
| Loss_ent | 1.5750606 | 1.7872881 |
| Loss_pol_entpen | -0.015750606 | -0.017872881 |
| Loss_pol_surr | 0.00037099834 | 0.0013152664 |

Table: 4

As we can see from the above figure (EpRewMean: (a)201 (b)152), with the random walk the algorithm performed significantly better during the training process. For this implementation, we have tested random walk with a uniform probability for all the actions. We can improve this model to have customized probabilities for different actions. Although random walk does improve the performance, we believe this is limited to initial learning process. Because, even without random walk, although it might take more time, the agent can learn to better play the game without being struck. So, in future implementations, we plan to introduce a decay rate to slowly decay the random jump probabilities.

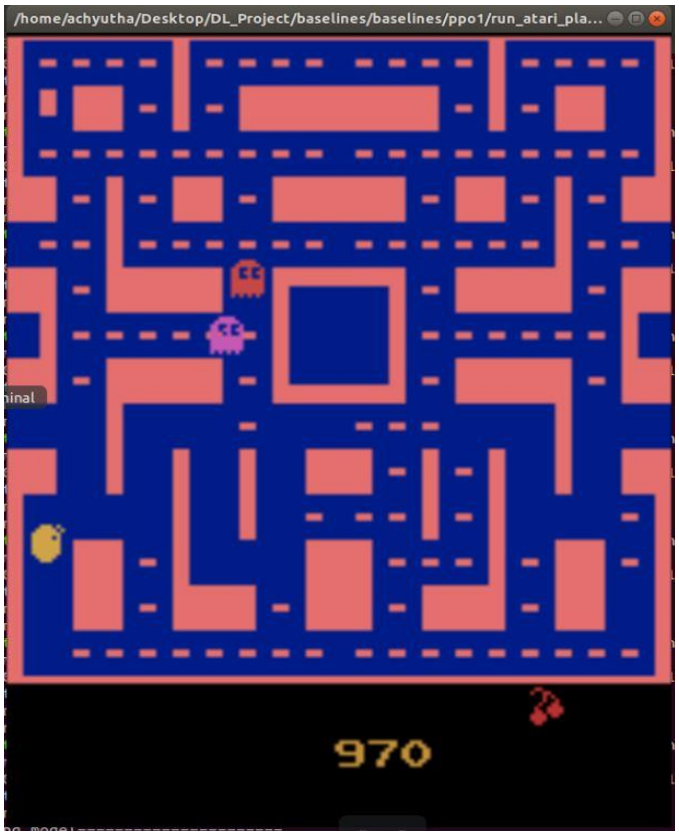

/home/achyutha/Desktop/DL_Project/baselines/baselines/ppo1/run_atari_pla...
970

Fig : 4 – 1 million timesteps

## V. GPU Based Simulations

The GPU simulations were run on Google Colaboratory using the baseline GYM code ‘ppo2’ that was optimized for GPUs. Different neural network architectures were used to train the Pacman agent.

### *A. Image Processing:*

Since we were using a GPU environment, the entire frame image was used without any preprocessing. One discrepancy we found was that in dimensions for the input image were (210 ,160, 4) compared to the CPU based simulation were the dimensions were (210,160,3). Possible reason could be that the frame itself was being modified using the environment property.

### *B. Using CNN as a neural network model:*

The Convolutional Neural Network model has 3 convolution layers and 1 Fully connected layer.

| **Layer** | **Filters** | **Filter Size** | **Stride** |
|---|---|---|---|
| C1 | 32 | 8 | 4 |
| C2 | 64 | 4 | 2 |
| C3 | 64 | 3 | 1 |

Table : 4

Where C1, C2, C3 are the convolutional layers which were then connected to a fully connected layer of size 512. No maxpool layers were used in the model. This model was used for training the agent for 5 million timesteps, the details of metrics used for measuring the performance are as follows:

| | |
|---|---|
| Maximum MeanReward | 2780 |
| Maximum Loss Value | 19577.08 |
| Total Time for training | 2.09e+04 seconds (~ 6 hrs) |

Table:

The normalized reward and loss values were plotted against timesteps.

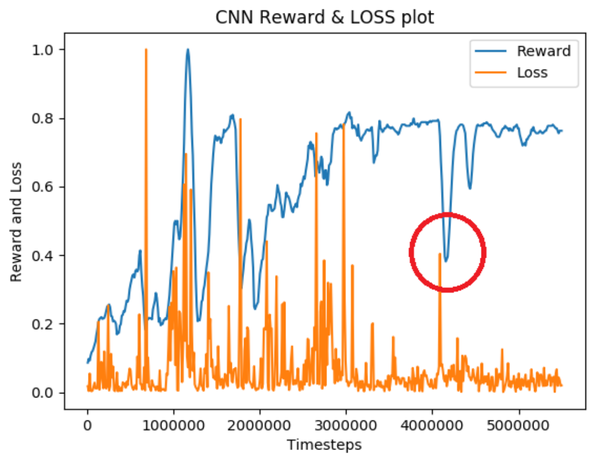


Fig: 5

As we can observe, the Reward gain is very noisy till 4 Million timesteps, but shows an increasing trend. Same is the case for the loss where the general trend is decreasing, but we see sudden spikes where the policy learns an undesirable move. An interesting point in the graph is the one circled in red where we see a clear correlation between learning a bad policy (low reward) and corresponding high loss. The model then recovers as seen from the reward and loss trend which start converging.

### *C. Using LSTM as a neural network model*

The LSTM model is an extension of the CNN model were along with the initial Convolution layers, an additional ‘LSTM’ layer is also included instead of the fully connected layer. It is expected that since we are using an additional layer of LSTM, there will be more calculations involved and consequently the training time will be more as the training strategy was similar to that of the CNN model and the values of some performance metrics are as follows:

| | |
|---|---|
| Maxium MeanReward | 795 |
| Maxmum Loss Value | 20489.45 |
| Total Time for training | 2.96e+04 seconds (~ 8 hrs) |

Table : 5

Compared to the CNN model, the max reward gained by LSTM is low (less than 50%) while the loss value are almost the same for both the models. Later in the section we also plot the actual reward and loss values for both the models for comparison.

Plot of normalized reward and loss values is as follows:

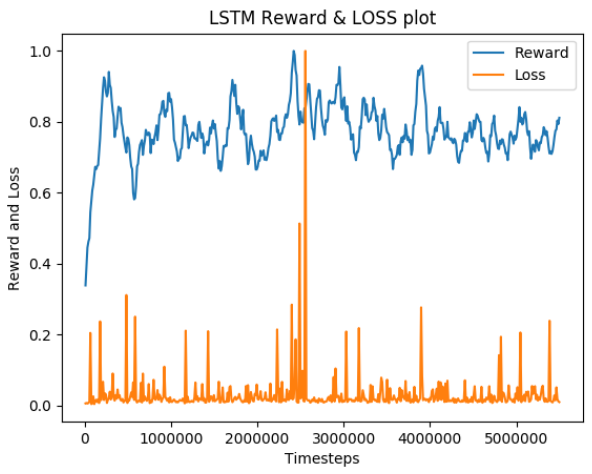


Fig:6

We see that the LSTM policy keeps the loss value contained except for a few instances. But the corresponding reward values are noisy and do not change much till the max timesteps are reached, although the noise starts to decrease.

## D. *Comparative Study of LSTM and CNN policy*

The final values of all the metrics from the baseline code, once the simulation ends for both the models are as follows:

| **Metric** | **CNN** | **LSTM** |
|---|---|---|
| approxkl | 1.20430625e-08 | 7.161611e-10 |
| clipfrac | 0.38745117 | 0.07104492 |
| eplenmean | 653 | 673 |
| eprewmean | 2.12e+03 | 645 |
| explained_variance | 0.99 | 1.65e-05 |
| fps | 261 | 187 |
| nupdates | 5368 | 5368 |
| policy_entropy | 1.0624752 | 1.486851 |
| policy_loss | 5.904658e-06 | -4.8603397e-08 |
| serial_timesteps | 687104 | 687104 |
| time_elapsed | 2.09e+04 | 2.96e+04 |
| total_timesteps | 5496832 | 5496832 |
| value_loss | 395.45374 | 190.38414 |

We can see that value loss (loss value used for plotting) is less for LSTM compared to CNN model, but LSTM has a slight higher episode length mean (eplenmean), but it is not clear how many episodes were played during a batch.

On the other hand, the final reward value for CNN is very high compared to LSTM. Since we could not find a clear documentation regarding what all the metrics mean, it will require a more in-depth analysis of the GYM's baseline repository

The plot for Loss and Rewards for CNN and LSTM models are as follows:

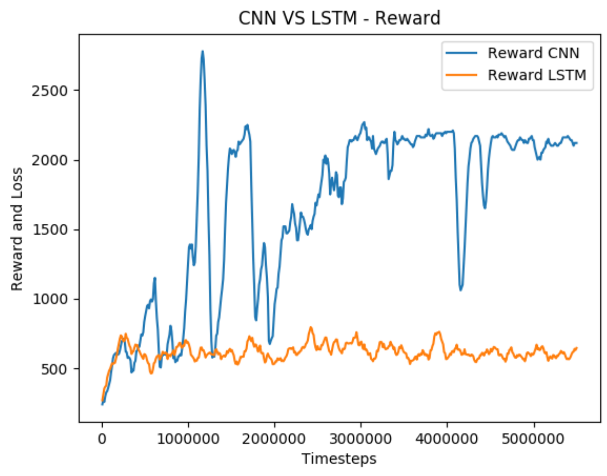


Fig :7

It is evident that for LSTM, the reward value starts to converge around approx. 650 and there is no general increasing trend, but the data is less noisy as well. For CNN however we see convergence at around approx. 2000 although there is mode noise in the output. Also around 1.3 million timestep, we see a huge dip in the reward value for CNN as it goes from very high to very low, indicating a large discrepancy in the learned policy.

The value loss plots for CNN and LSTM are as follows:

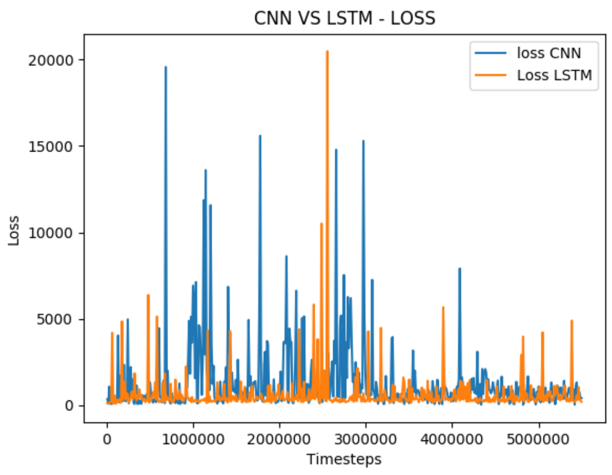


Fig:8

The trend for value loss for LSTM shows less noise compared to CNN, but there are more sudden spikes during the later stage of LSTM model. Also we can observe that the high loss value occurs very early for CNN , but for LSTM, it is around the middle. It could be that

- the 5 million timestep value might be low for LSTM to learn a good policy
- We might not see any specific trend once the 5M threshold is crossed.

The loss value for CNN clearly starts to converge during the last 1M timesteps and comparing the trend to its reward we see convergence there as well.

## VI. SIMULATION ENVIRONMENT & HARDWARE

### A. *For CPU based Simulations:*

We used Ubuntu 17 OS on VMware Workstation. A virtual environment was used in the OS to install all the Tensorflow and GYM dependencies. RAM: 13 GB, virtual Processors: 4 processors with 2 cores each

### B. *For GPU based Simulations: (submit code for this)*

We used the free GPU environment available on 'Google Colaboratory'. The cloud VM was Linux based and the simulation code was mounted from our Google drive. Jupyter notebook was used to edit and run the code. RAM : 11 GB, Processors : 2 CPUs - Intel(R) Xeon(R) CPU @ 2.30GHz, 1 GPU - Tesla K80 major: 3 minor: 7 memoryClockRate(GHz): 0.8235

## VII. CHALLENGES

### A. *Environment and Simulation*

- Setting up OpenAI gym's environment is tough on a windows machine as there is no support for ATARI environment on windows. So, we had to switch to a Linux machine.
- Apart from that there are a lot of dependency packages like python 3.5+, tensorflow, etc. that takes a lot of time to set up before we can proceed with the development of our agent.
- The training time for the model also requires more setups. Since we did not have access to GPU environment initially, we had to setup the environment on multiple systems in CIDSE computer Lab and run different models individually on different machines. It is difficult to monitor multiple simulations and we did loose outputs of some early simulation runs
- For GPU environment, using the Google Colaboratory environment was also challenging initially and it has some constraints like limited 12 hour time for simulations and reconfiguring the environment (setting up the dependencies) after every 12 hours.
- Lack of documentation regarding the various commands/ output format/ model saving/ libraries on the GYM's baseline code meant investing time in understanding the multiple modules in the repo to identify what code base to alter.

### B. *Huge learning time*

- From the paper and some other implementations, we found that the learning time is huge before the agent becomes as good as a normal human, usually more than a day of training is required (for CPU simulations). If there are any problems it is difficult to fix and retrain. We are still trying to find a fix for this. Currently we thought of saving weights regularly on a flat file so that we can read weights and continue training, but we did not find and specific documentation for the baseline code.
- Although we were able to save some models for the CPU simulation, we could not do the same for the GPU based simulations.

## VIII. CONCLUSION AND FUTURE WORK

During the course of this project we explored different Reinforcement learning algorithms based on the baseline code shared by OpenAI gym environment and its observation/ reward model. We experimented with different scenarios like changing the input image dimensions (greyscaling/ maxpooling), making algorithmic modification during learning (random walk) and modifying the clip parameter in the gradient function. We also ran the simulations on CPU and GPU so as to more accurate results. To run the GPU

simulations we leveraged the service of Google Cloud colaboratory.

This work can be extended by optimizing the base code further so as to run faster simulations on cloud environments like Google Colaboratory and adding more explicit model saving and testing code in the baseline repository.

.

## IX. INDIVIDUAL CONTRIBUTION

| | |
|---|---|
| Achyutha Sreenivasa Bharadwaj | Environment setup, Base line testing. Simulating and collecting environment states with random actions to create state history. CPU simulations. Random walk algorithm. |
| Ninad Jadhav | • Maxpooling and Greyscaling for CPU simulations<br>• Modification of clip value and running simulations for the same<br>• GPU based simulation and comparative study between the models |

## X. REFERENCES

[1] DEEP REINFORCEMENT LEARNING: PONG FROM PIXELS
Karpathy.github.io. (2018). Deep Reinforcement Learning: Pong from Pixels. [online] Available at: http://karpathy.github.io/2016/05/31/rl/

[2] TF.IMAGE.RGB_TO_GRAYSCALE | TENSORFLOW
tf.image.rgb_to_grayscale | TensorFlow. [online] Available at: https://www.tensorflow.org/api_docs/python/tf/image/rgb_to grayscale

[3] OPENAI GYM
Gym: A toolkit for developing and comparing reinforcement learning algorithms. [online] Available at: https://gym.openai.com/docs/

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," arXiv preprint arXiv:1707.06347, 2017. [Online]. Available: https://arxiv.org/pdf/1707.06347.pdf

[5] Li, Ke and Malik, Jitendra. "Learning to optimize," arXiv preprint arXiv:1606.01885, 2016. [Online]. Available: https://arxiv.org/pdf/1606.01885.pdf