CS230

Policy Gradient Methods with Pong

Jeffrey Hu
Department of Computer Science
Stanford University
jeffhu@stanford.edu

Gregory Soh
Department of Computer Science
Stanford University
gregsoh@stanford.edu

Abstract

The goal of much of artificial intelligence and deep learning research is to develop ways to automate "human" tasks. Among the several fields that exist in AI, two important ones for developing further automation are reinforcement learning and computer vision. To sufficiently solve several of the problems that humans can do, what is needed is an ability to process videos and learning the proper actions that should be taken. One particular example of the intersection of these two fields can be found in the Policy Gradient algorithm. In this paper we apply the REINFORCE policy gradient algorithm to the Atari video game *Pong* and analyze the performance of different models applied to the task.

1 Introduction

The problem that is considered in this paper is analyzing different types of policies for use in solving the *Pong* video game. More specifically, the policies used are different types of deep neural networks (DNN). In the DNN community, there is a recognized problem of understanding what model architectures are suitable for which problems. While *Pong* is by no means a new problem, this paper attempts to analyze which model architectures are more suited towards this problem.

The inputs in this problem are the pixels of *Pong*, served as a 210 by 160 by 3 array, where 3 represents the rgb color channels. The output of the models tested are softmax arrays with probabilities of actions to be taken. The agent interacts with an environment through actions; at each timestep, the environment gives the agent an observation, and the agent returns an action. This back and forth occurs until the end of each episode.

This project is used in both CS230 and CS221. The part of the project dedicated to policy gradient is for CS221, while the part focused on different types of models is for CS230.

2 Related work

The most well known paper regarding *Pong* would most likely be the DeepMind DQN paper. While their paper does not use policy gradient methods, the results using a Deep Q-Network are state-of-the-art, achieving a solid 20 out of 21 points on each run. Our approach differs from theirs on three accounts. The first and primary is the fact that rather than using a Deep Q-Network, we used Policy Gradients. Secondly, the environment used in the DeepMind paper was deterministic, with a standard frameskip of 4 frames. The environment tested in this paper is stochastic, leaving the possibility of 2, 3, or 4 skipped frames per action. Finally, while their paper focused on achieving results at at any cost (spending millions of frames to train), this one compares training times, and analyze the time it takes until convergence.

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

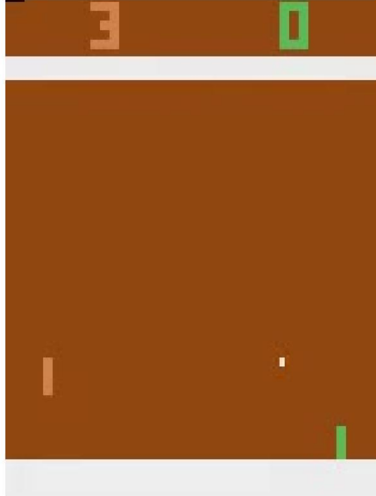

Figure 1: An example of a game state. The green represents the autonomous agent, while the orange is the hard coded bot. The goal is to get the ball past the opponent.

Another prominent related work is Andrej Kaparthy's blog post, *Pong from Pixels*. He gives an overview of the policy gradient algorithm as well as the broader implications of such an algorithm. The approach he takes is not amazing. Rather than trying to perfect his code, he focuses more on making it conceptually clear. As such, the score that his model ends up converging to is 0 points; as good as the hard coded bot.

3 Dataset and Features

The "dataset" used in this project is the OpenAI Gym. In particular, the environment used was Pong-v0. This differs from the environment of the DQN paper, as they used the equivalent of PongNoFrameskip-v4. At every timestep, the agent is supplied with an observation, a reward, and a done signal if the episode is complete. The reward is given every time a point is finished. If the agent won the point, the reward is +1. If the agent lost the point, the reward is -1.

The images were preprocessed to assist the agent in understanding the velocity of the ball and reduce computation time. To be precise, the image that was fed into the model was the difference between the current image and the last image, grayscaled. Along with preprocessing, after each episode the discounted reward was calculated to find the corresponding reward for that timestep. To calculate a discounted reward for a timestep t, the formula $\gamma^t r_t$ was used.

4 Methods

4.1 Algorithm

As stated earlier, the reinforcement learning algorithm used is REINFORCE (Williams, 1992). This is the vanilla policy gradient algorithm, and advantage was not implemented. The goal of REINFORCE is to maximize the expected return of an episode, where an episode is a Markov Decision Process. Formally, this can be stated as the following.

$$\arg\max_{\theta} \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right]$$

Where θ is the parameters of the policy, π is the policy, and T is the length of the MDP. The goal of policy gradient is to learn the value of θ that maximizes the expected value above. To do so, gradient ascent is performed. Taking the gradient of the expected value with respect to the parameters, and then simplifying, we get

$$\nabla_\theta \mathbb{E}[R(\tau)] = \mathbb{E}\big[R(\tau)\ \nabla_\theta \log p(\tau|\theta)\big]$$

Since the reward function is unknown, the method to calculate this gradient is by using a Monte Carlo method of sampling the space for trajectories. To do so, the inside of the expected value on the right side of the above equation is repeatedly sampled. To calculate the gradient of the log term, the following equation is used.

$$\nabla_\theta p(\tau|\theta) = \nabla_\theta \sum_{t=0}^{T-1} \log\pi_\theta(a_t|s_t)$$

Intuitively, this equation sums up the log probabilities of taking each action given the state at time t. Then, the gradient is taken with respect to the parameters, and gradient ascent is performed. Combining all these equations together, we get the raw gradient, which can be expressed as the following.

$$\nabla_\theta \mathbb{E}_{\pi_\theta}[R(\tau)] = \mathbb{E}_{\pi_\theta}\left[R(\tau) \cdot \nabla_\theta\left(\sum_{t=0}^{T-1} \log\pi_\theta(a_t|s_t)\right)\right]$$

Of course, the derivations covered are more involved, and can be found in the original paper, *Policy Gradient Methods for Reinforcement Learning with Function Approximation* (Sutton, et al.). To implement policy gradient algorithmically, there are a number of methods, including Trust Region Policy Optimization, Proximal Policy Optimization, and Asynchronous Advantage Actor Critic. Here we use vanilla policy gradient which implements an algorithm known as REINFORCE.

Algorithm 1 Monte Carlo Policy Gradient[3]

1: **procedure** REINFORCE
2: Initialise θ arbitrarily
3: **for** each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\}$ **do**
4: **for** $t = 1$ to $T - 1$ **do**
5: $\theta \leftarrow \theta + \alpha\nabla_\theta \log \pi_\theta(s_t, a_t)v_t$
6: **return** θ

The policy that is used in this project is a Deep Neural Network, as it has the property of being a universal function approximator, theoretically allowing it to learn the optimal policy. The neural networks tested are discussed in the following section.

4.2 Models

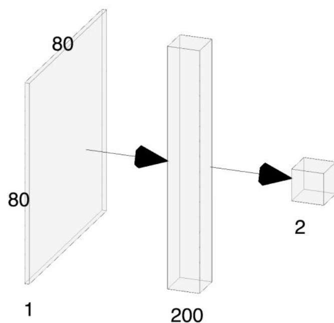

Figure 2: Model architecture for models 1 and 3.

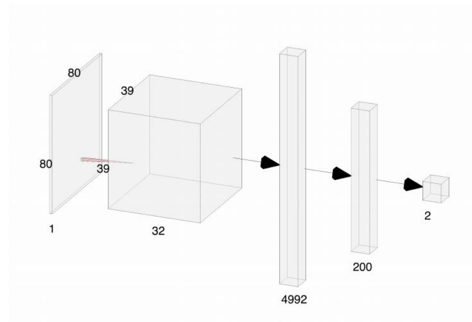

Figure 3: Model architecture for models 2 and 4.

The model architectures tested were comprised of two main types. The first type was a simple three layer neural network, comprising of an input layer, a 200 neuron hidden layer, and a softmax output. The input layer formed by flattening the input image into a vector of dimension 1600, and fully connecting it to the hidden layer.

The second architecture was a 6 layer fully connected network. The implementation used a convolutional layer followed by a flattening layer, which fed into two hidden layers. The convolutional layer 32 filters of size 6 by 6 with a padding of same. The first hidden layer was size 64, and the second, 32. The final layer was the same as the first type, with a softmax output.

Both models used ReLU activations for all the neurons. The first model used a random uniform distribution for weight initialization, while the second used He initialization.

For models 1 and 2, the output size was 2, while for 3 and 4 the output size was 3. The size of 2 corresponded to the action set `[UP, DOWN]`, while size 3 corresponded to the set `[UP, NOOP, DOWN]`.

5 Experiments/Results/Discussion

The models were tested on AWS p2.xlarge clusters that were GPU accelerated with an NVIDIA K80 GPU. The models usually took about 12 hours to train until 7000 episodes.

Figure 4 displays the both of the models that were tested with only two actions. As expected, the convolutional model learned much faster, reaching a score of over -10 by the end of episode 2000. Model 1 took a little while to start learning, and then it more steadily converged to the final policy. What was found in the tests was that often times some of the models would be stuck at -21 for a while until they received sufficient reward to start learning the proper actions. We noticed that even though the convolutional layer should have much better performance than the single layer, they both ended up converging to roughly the same value. Originally, two outputs were used to help increase convergence speed by forcing the model to learn to hit the ball. DeepMind's paper stated that they used the entire Atari range of actions, so it was suspected that that increasing the action space would help performance. Along with that, it could be seen that often times the models would try to move away and shake to time the hit, knowing that they couldn't stay in one place.

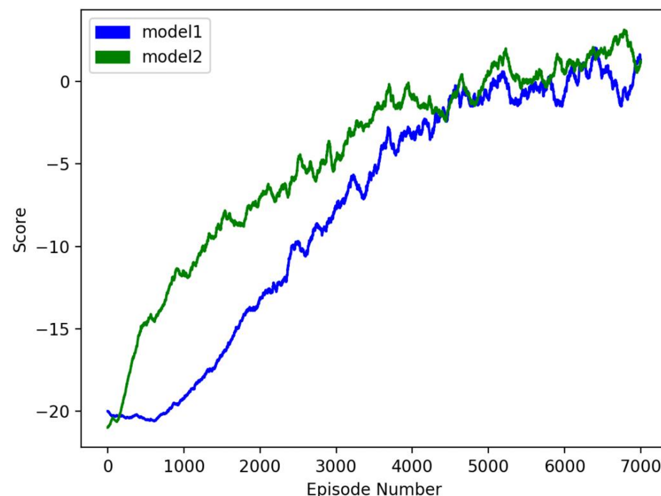

Figure 4: Graph of episode vs game score for models 1 and 2.

Figure 5 displays the models that were tested with the action space that included `NOOP`'s. Beyond just increasing the learning rate of the convolutional network, it also drastically increased the performance of it, eventually giving it a convergence score of +10. This blew all of the other tested models out of the water. Interestingly, the model with one hidden layer did not improve much, if at all, compared to the previous iteration. We suspect that the increased action space did not affect the convergence rate much (perhaps slowing it slightly). More likely the reason that the model ended up reaching the same convergence score is due to the fact that it was reaching the limit of how accurate of an approximatable function could be achieved by 200 neurons.

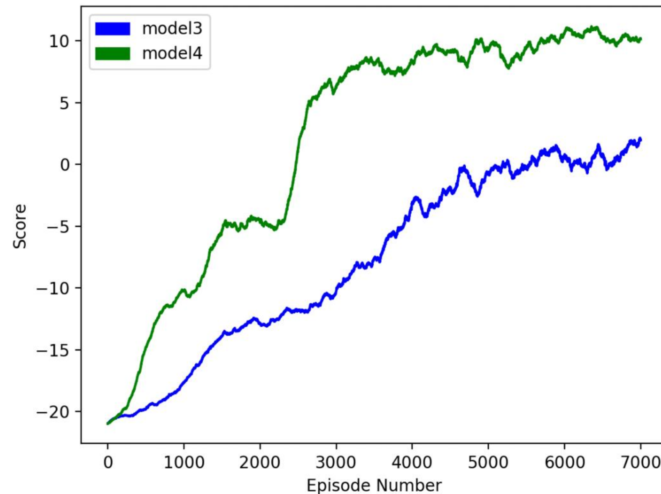

Figure 5: Graph of episode vs game score for models 3 and 4.

All models were tested with the following hyperparameters. An Adam optimizer was used with momentum and a learning rate of 0.001. The mini-batch size was a single episode. This differs from stochastic gradient descent in that for each episode, thousands of actions are recorded as well as the corresponding rewards.

Metric	Model 1	Model 2	Model 3	Model 4
Trials until 0	5056	4616	5185	**2471**
Highest Score	2	3	2	**11**
Mean Score (overall)	-8	-5	-8	**2**
Mean Score (last 1k)	0	2	1	**10**

Table 1: Numerical Results

6 Conclusion/Future Work

Considering that the policy gradient agents came in with no understanding about anything in the world, the results of being able to successfully play is impressive. The best model tested ended up achieving a solid 10 points, which beats the human level performance of 3 by a decent margin. Of course, this result is not optimal; other people have achieved scores of 20 on *Pong*. However, the goal of this paper was to investigate some of the different types of models as successful function approximators rather than just achieving the highest score. It can be seen that as a whole, the convolutional neural network did substantially better than that of the deep neural network. Along with that, giving it the capability of three softmax outputs also had a substantial boost on performance.

For future work, testing more models with different implementations to see the comparisons would be ideal. The fact that each model took 10+ hours to train was a big bottleneck in the iteration process; spending more time to speed up the code through vectorization would be beneficial as well. By implementing different model architectures and gradient methods, we could develop a more comprehensive report. We hope that this paper serves as a good overview of the comparisons of policy gradient architectures.

7 Contributions

Jeffrey Hu developed most of the code and models, while Gregory Soh developed testing pipelines for training the model.

References

[1] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).

[2] Kaparthy, Andrej. "Deep Reinforcement Learning: Pong from Pixels." *Andrej Karpathy Blog*, 31 May 2016, `karpathy.github.io/2016/05/31/rl/`.

[3] Silver, David. "UCL Course on RL." *David Silver's Webpage*, `www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`

[4] Recht, Benjamin. "The Policy of Truth." *Arg Min Blog*, 20 Feb. 2018, `www.argmin.net/2018/02/20/reinforce/`.

[5] Seita, Daniel. "Going Deeper Into Reinforcement Learning: Fundamentals of Policy Gradients." *Seita's Place*, 27 Mar. 2017, `danieltakeshi.github.io`.

[6] Sutton, Richard S., et al. "Policy gradient methods for reinforcement learning with function approximation." *Advances in neural information processing systems*. 2000.

[7] Kweon, Kyung Mo. "Mo's Notes." *Mo's Notes*, `kkweon.github.io/`.

[8] Schulman, John. "Policy Gradient Methods: Tutorial and New Frontiers." *Microsoft*, 3 July 2017, `www.microsoft.com/en-us/research/video/policy-gradient-methods-tutorial-new-frontiers/`.

[8] Frans, Kevin. "Simple Reinforcement Learning Methods to Learn CartPole." *Kevin Frans*, Kevin Frans, 7 Jan. 2017, `kvfrans.com/simple-algoritms-for-solving-cartpole/`.

Libraries Used: Tensorflow, Keras, Matplotlib, OpenAI Gym, Numpy