Learning to Play Pong with Policy Gradient Methods

Botond A. Nás
Department of Computer Science
Eötvös Lóránd University
Budapest, HU
khupib@inf.elte.hu

1 Introduction

Reinforcement learning (RL) tasks deal with learning the Markov Decision Process (MDP) model with the following parameters: state-values, V; action-values, Q; and policy, π. Video games, such as Atari's classical *Pong* provide rich and challenging domains to test RL methods and algorithms. RL learns to behave in an environment, even though the model of the environment is not available, by randomly sampling the state space and noting action and state pairs that yield higher rewards. Traditionally, RL learning techniques implemented tabular approaches e.g., temporal difference learning (TD), Q-learning, and SARSA. The tabular RL approach works well when the state space is not too large because the RL can thoroughly explore the state space [Phon-Amnuaisuk, 2018]. However, most real life problems have a large state space. The issue could be mitigated if the balance between exploitation and exploration is maintained. The balance between exploring unvisited states and exploiting rewarding states ensures that near optimal policy is learned within a reasonable time [Phon-Amnuaisuk, 2018]. The exploration-exploitation problem has sparked interest in applying *deep learning* (DL) to RL, resulting in deep reinforcement learning methods (DRL).

One such class of methods is the Policy Gradient methods. Policy Gradient methods rely on optimizing parameterized policies with respect to expected cumulative rewards using gradient descent. That is, they model and optimize the policy directly, not needing the value function. The game-play is simple: the agent receives an image frame and decides if it wants to move the paddle up or down (i.e. a binary choice). After every single choice, the game simulator executes the action and gives a reward: either a +1 reward if the ball went past the opponent, a -1 reward if the agent missed the ball, or 0 otherwise. The goal is to move the paddle such that the agent gets the most reward. This is an example of a classical reinforcement learning task where the image (ball and paddle locations) are the states and the actions are the aforementioned possible paddle movements.

2 Related Works

In 2013, Volodymyr Minh, a researcher at DeepMind, published a paper with fellow co-collaborators at DeepMind in which they developed a system that uses deep RL to play various Atari games, including Pong. The system was trained purely from the pixels of an image from the video-game display as its input, without having to explicitly program any rules or knowledge of the game. Their model is a convolutional neural network (CNN), trained with a variant of Q-learning (Q-learning with function approximation found in Sutton and Barto [1998]), whose input is raw pixels and whose output is a value function estimating future rewards [Mnih et al., 2013]. Implementations of this solution are also detailed by Parmelee [2021] and Torres [2020]. Both of their solutions use a Deep Q-Network (DQN), modeled by a CNN, that represents the optimal action-value function as a neural network. However, Mnih et al. have shown Policy Gradients to work better than Q-Learning when tuned well [Karpathy, 2016]. In 2016, using a different approach, Andrej Karpathy wrote a 130-lines-of-Python algorithm (using only a couple dependencies) to solve Pong. He trained a 2-layer policy network with 200 hidden layer units using RMSProp on batches of 10 episodes, applying the concept of Policy Gradients. After training the network for three nights, he trained a policy that is slightly better than the computer [Karpathy, 2016]. In his blog post, he admits that an adequately trained CNN, like the one used by Mnih et al., performs better than his solution. His solution, however, is a good demonstration of deep RL in action and, coupled with his blog post, intuitively illustrates the main ideas of deep RL and Policy Gradients.

3 Problem Description

Pong is a table-tennis themed video game manufactured by Atari and originally released in 1972. The game features simple two dimensional graphics. The player controls an in-game paddle by moving it vertically across the left or right side of the screen. They can compete against another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth.

Pong can be viewed as a classic reinforcement learning problem, as there is an agent within a fully-observable environment, executing actions that yield differing rewards, using the magnitude of collected reward to self-optimize. Furthermore, Pong serves as a good example of a MDP, as every state (defined here as the position of the agents and the ball within the gamespace) is independent of each other. In the context of this environment, the *action* is the output of the model: whether the paddle should be moved up or down. The *reward* is -1 after missing the ball, +1 if the opponent missed the ball. An episode is over when a player reaches 21. The goal is to train an agent to play Pong and maximize its reward (i.e. win). This is done by training the policy, or the strategy, of the agent using Policy Gradient methods. Policy Gradient approaches rely on optimizing policies with respect to long-term cumulative reward by gradient descent. The policy is parameterized using a deep neural network with parameters θ. Using OpenAI Gym's environment allows for the simulation of Atari games, providing a direct interface to the methods and variables within the games. First, each frame is inputted into an untrained network and the predicted binary probability is used to predict an action to take in the game. This is repeated for the entirety of the episode, accumulating X, Y, and rewards in arrays. After the episode is finished, the network is trained using the X, Y, and reward arrays [Xu, 2019]. The x-variable is generated by the game and represents the difference between two frames, which give the state of the game to the network as an input. The original 210 pixel by 160 pixel by 3 color channel frame contains two many unnecessary details irrelevant to the actual gameplay itself so it is pre-processed by applying a crop and a grayscale [Xu, 2019]. The final input image is an 80 pixel by 80 pixel difference of the two grayscale images (see Figure 1) which is then converted to a 6400 element 1D float array.

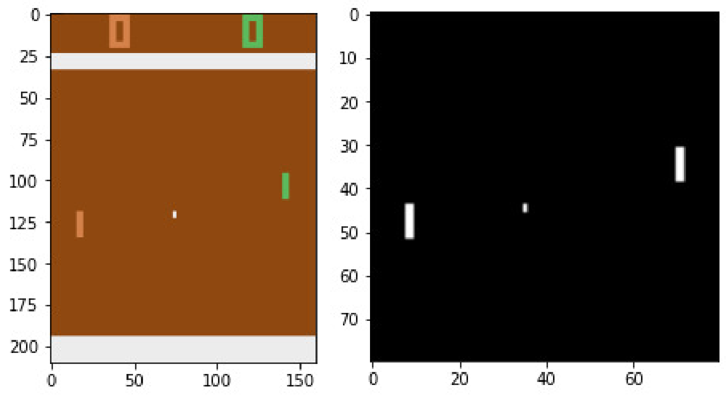

Figure 1: Original frame and difference frame after pre-processing based on Karpathy [2016]

The y-variable, generated by the network, represents a predicted action of the network based on the input difference array. The reward assists in judging the suitability of different actions, punishing or rewarding the agents for undesired and desired behavior and is also generated by the game.

4 Methods

Policy Gradient methods learn a parameterized policy that can select actions without consulting a value function [Sutton and Barto, 1998]. The objective of a reinforcement learning agent is to maximize the "expected" reward when following a policy π. Policy Gradient learning differs from the tabular solution methods which update their tables entry by entry. The tabular approach is inefficient if the state space is large because having a large state space often leads to a poor generalization of ability since the whole state space cannot be thoroughly explored [Phon-Amnuaisuk, 2018]. In contrast, policy learning efficiently approximates the policy by back-propagating errors to adjust the weights of the policy network. Policy π is often parameterized using a deep neural network π_θ with parameters θ [Levine, 2017]. The goal is to solve for θ^* that attains the highest expected episodic return

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]. \tag{1}$$

An estimation of the objective function can be attained by summing over samples collected by means of π_θ

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} \approx \frac{1}{N} \sum_i \sum_t r(s_{i,t}, a_{i,t}). \tag{2}$$

The policy parameter is learned based on the gradient of the performance measure $J(\theta)$ with respect to the policy parameter. Differentiating the estimation of the objective function with respect to θ results in the gradient

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t} \right). \tag{3}$$

The methods aim to maximize performance, so their updates approximate gradient ascent in J:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t), \tag{4}$$

where α is the learning rate [Sutton and Barto, 1998].

I used the REINFORCE Policy Gradient learning algorithm, summarized by Algorithm 1. REINFORCE relies on estimated return by Monte Carlo methods using episode samples to update the policy parameter θ. REINFORCE works because the expectation of the sample gradient is equal to the actual gradient. Therefore we are able to measure return from real sample trajectories and use that to update our policy gradient.

Algorithm 1: REINFORCE: Monte Carlo Policy Gradient Control (episodic) for π_θ

```
Input: a differentiable policy parameterization $\pi_\theta(a, s)$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\theta$
Loop forever:
    sample $\tau_i$ from $\pi_\theta(a_t, s_t)$ (run the policy)
    $\nabla_\theta J(\theta) = \sum_i \left(\sum_t \nabla_\theta log\pi_\theta(a^i_t|s^i_t)\right) \left(\sum_t r(s^i_t, a^i_t)\right)$
    $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
```

As I mentioned before, policy π is often parameterized using a deep neural network π_θ with parameters θ. My solution, inspired by Karpathy's solution, includes image pre-processing and reward discounting methods with discount factor $\gamma = 0.99$. The model is a deep neural network with two densely-connected layers. The hidden layer consists 200 units and ReLU activation, taking a pre-processed frame as input. The output layer uses Sigmoid activation to get either 0 or 1, representing the action. The model is compiled with Adam optimizer with 0.001 learning rate and binary-cross entropy loss, resulting in 1,280,401 parameters.

Training the network starts with collecting data using the pre-processing function. The input is fed into the network and an action is predicted. The x and y variables are appended to the datasets, in preparation for training, and the actions are fed into the Pong environment, from which a reward value is obtained and stored. Once an episode has finished, the total reward acquired is reported and the model fit on the training data to generate an association between the states observed and the actions taken. The weights of the inputs are weighted by their relative contribution to the rewards.

5 Results

Following the implementation detailed by Xu and Phon-Amnuaisuk, I modeled a network with one hidden layer of 200 nodes with ReLu activation and an output layer with 1 output unit with Sigmoid activation. ReLU is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance. Using Sigmoid activation in the output allows the output to be a probability value between 0 and 1, representing the probability of choosing the 'move paddle up' action. The model used the Adam optimizer with the default 0.001 learning rate and binary cross-entropy loss. Binary cross-entropy loss is optimal in this situation because the goal is to classify the different pre-processed frames (states) based on the action that should be taken in that state. The loss is defined as $L = -\sum_{i=0}^{n} log(P_{a_i})D_i$ where P_{a_i} is the probability that the agent predicted the action it actually took at time i and D_i is the discounted reward from time i [Ritchie, 2019]. If an action has a low probability and a high reward or it has a high probability and a low reward then its loss will be high (see Figure 2). During training, I

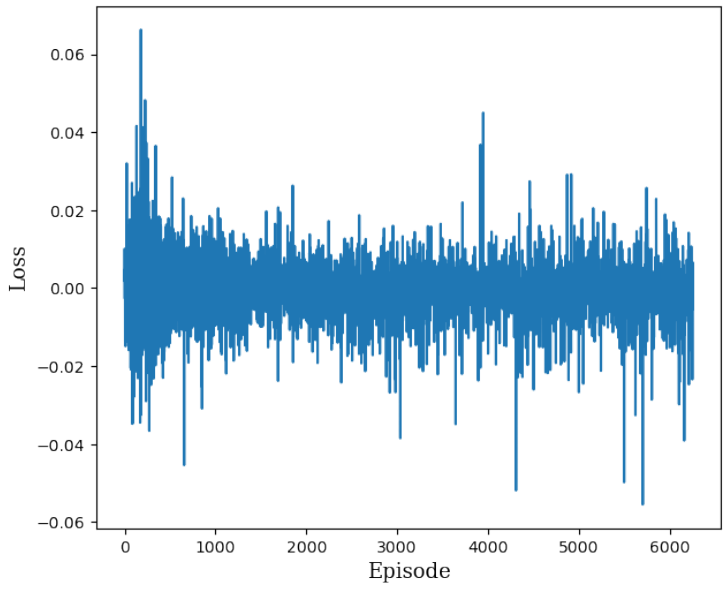

Figure 2: Average loss for each episode

passed an array of discounted rewards, with a discount rate of 0.99 (as suggested by Karpathy), used for weighting the loss function.

I trained the policy network for 6,254 episodes over the course of six days and nights. The rewards distribution of the model after training can be seen in Figure 3. Only on one occasion during the training did the policy network achieve a reward greater than zero, meaning it won a game. Considering our objective to train the network to win the game, that is a quite disappointing result. However, the network's improvement can be observed over the course of the episodes. The improvement is more easily seen in the bottom plot of Figure 3, where the average rewards are taken over every 20 episodes. As the training progresses, the agent is able to score more and more points in each match. Testing the policy yielded the same results. The agent lost every game of 100 simulated games of Pong where the agent acted based on the policy from the network, although with varying scores each time. The number of episodes I trained the network is simply not enough to learn a policy that allows the agent to win a game of Pong. My testing supports the findings of Phon-Amnuaisuk [2018] that at least 10,000 training episodes must be completed for this model to produce a policy that can win a game. These results emphasize a major drawback to Policy Gradient method implementations using deep RL: they are slow to converge.

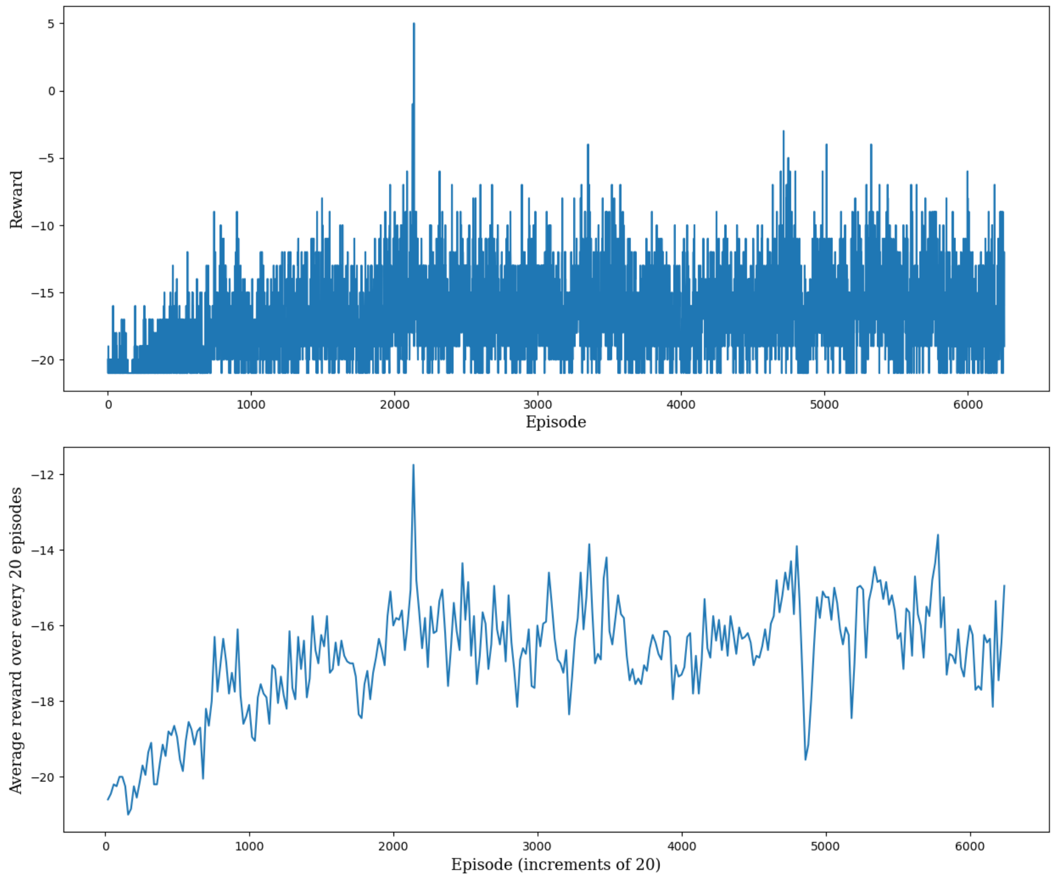

Figure 3: Aggregate reward after each episode (top) and average aggregate reward over every 20 episodes (bottom)

6 Conclusion/Future Work

Reinforcement learning is a process in which an agent learns by doing. The agent attempts to learn a policy from observations to actions it takes in specific states, in order to maximize its returns. One way to do this is to consult the value function and choose the action with the highest expected future reward at each state. These methods, such as Q-learning and deep Q-learning, rely on updating the value function based on observations. Policy Gradient methods, however, are more advantageous in many situations. Policy Gradient methods optimize the policy directly by parameterizing the policy. As a result, they are more effective in high dimensional action spaces and can learn stochastic policies, eliminating the need to implement an exploration/exploitation trade-off.

Training an agent to play the classical video game Pong with Policy Gradient methods requires parameterizing the policy with a deep neural network. The parameters of the policy network are trained with the observation data made by the agent after each episode (each game). After enough episodes, the policy network learns which action to take in each state to win the game. On insufficient resources, however, training the network to this point takes a lot of time. Given more time and resources, I would first train my model enough to win a game and improve my current model by changing the parameters and observing the network's behavior. For comparison, I would implement the solution proposed by Mnih et al. [2013] (using Deep Q-Learning) and attempt to design a Deep Q-Learning model, using convolutional networks, that converges faster to the optimal policy than Karpathy's solution. This model would consist of three 2D convolutional layers with ReLU activation and kernel sizes of 8, 4, and 3 respectively. The output layer and compilation parameters would be identical to the densely-connected model, resulting in 78,241 parameters.

References

A. Karpathy. Deep reinforcement learning: Pong from pixels, May 2016. URL `http://karpathy.github.io/2016/05/31/rl/`.

S. Levine. Policy gradients. `http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_4_policy_gradient.pdf`, 2017. Accessed: 2021–05-03.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

D. Parmelee. Getting an ai to play atari pong, with deep reinforcement learning. `https://towardsdatascience.com/getting-an-ai-to-play-atari-pong-with-deep-reinforcement-learning-47b0c56e78ae`, Jan 2021. Accessed: 2021–05-010.

S. Phon-Amnuaisuk. Learning to play pong using policy gradient learning, 2018.

D. Ritchie. Hw6: Reinforce and reinforce with baseline, Sep 2019. URL `http://cs.brown.edu/courses/cs1470/projects/public/hw6-reinforce/hw6-reinforce.html`.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. URL `http://www.cs.ualberta.ca/~sutton/book/the-book.html`.

J. Torres. Deep q-network (dqn)-i: Openai gym pong and wrappers. `https://towardsdatascience.com/deep-q-network-dqn-i-bce08bdf2af`, Aug 2020. Accessed: 2021–05-06.

A. Y. Xu. Fundamentals of reinforcement learning: Automating pong with a policy model— an implementation in keras. `https://medium.com/gradientcrescent/fundamentals-of-reinforcement-learning-automating-pong-in-using-a-policy-model-an-implementation-b71f64c158ff`, Oct 2019. Accessed: 2021–05-03.