Reinforcement Learning Using Deep Q-learning

Toshal Ghimire
University of Colorado Boulder
toshal.ghimire@colorado.edu

Bryce Melvin
University of Colorado Boulder
bryce.melvin@colorado.edu

Nikhil Rajaram
University of Colorado Boulder
nikhil.rajaram@colorado.edu

Abstract

Deep learning has provided a promising method to model functions of many parameters and outputs. For this reason, its applications in reinforcement learning are being explored in order to have an agent learn an optimal policy given a general input — particularly, images of the environment. We decided to take on the topic of reinforcement learning by training a Convolutional Neural Network (CNN) to use with Q-learning to beat the Atari game Pong. To build the network, we used the Keras API with the Tensorflow backend. Although solving Pong as a problem is not of much interest or utility to the world, the task partly showcases that a computer is capable of learning a generalized set of actions to complete a specific task without — to some extent — the need of a programmer defining the task explicitly. For example, personalized recommendations, robotics, and even chemistry are all real world uses of reinforcement. The ability to have an algorithm learn an optimal set of actions in a given environment by rewarding the algorithm theoretically allows for any problem under this structure to be solvable. This is an important task for the team to tackle as the benefit of learning such concept can have a positive impact if used correctly.

1 Problem formulation

1.1 Markov Decision Processes

Markov Decision Processes (MDPs) are a framework used to help an agent make decisions in a stochastic environment. This technique can be used to mathematically formalize sequential decision making in applicable environments. It is the basis used to solve many reinforcement learning problems as most reinforcement learning problems can be formalized as MDPs. There are four key parameters for MDPs: a state set S, an action set A, a transition model $P(s'|s,a)$, and a reward function $R(s)$. The goal of solving a MDP is to find the optimal policy $\pi(s)$ for a given environment. The policy can be thought of mapping a state $s \in S$ to an action $a \in A$. If the MDP is solved, then $\pi(s)$ will return an action that maximizes the reward in state s. The transition model $P(s'|a,s)$ is a model that takes a state and action pair, and yields the probability that taking action a in state s will lead to state s'. Finally, the reward function $R(s)$ will return the reward achieved for state s.

In order to have the input for Pong be nonspecific (i.e. at least applicable to other Atari games), we fed it image representations of the game. Thus, our state set S was defined by these images. In Pong, there are technically three potential actions for an agent to take – up, no move, or down. To simplify

Preprint. Work in progress.

our model and speed agent exploration, we chose to stratify the action set A to only up or down actions. Additionally, our reward function $R(s)$ was defined simply as follows:

$$R(s) = \begin{cases} 1, & s \text{ is a terminal state, agent has won the game} \\ -1, & s \text{ is a terminal state, agent has lost the game} \\ 0, & s \text{ is not a terminal state} \end{cases} \tag{1}$$

1.2 Q-learning

Introduce Q-learning: a method that outputs a policy $\pi(s)$ after learning a Q-function $Q(s, a)$ that is tied to the utility value of a state as follows:

$$U(s) = \max_a Q(s, a) \tag{2}$$

Q-learning is good for our purposes for two reasons: it is model-free and, given a finite MDP and correct learning rate, will converge to the (global) optimal policy(1). It being model-free serves us well as we do not *explicitly* know the transition model $P(s'|s, a)$. Additionally, it converging to the global optimal policy is ideal as it can tell us to what extent Pong is solvable.

A vanilla Q-learning agent initializes a Q-table indexed by states and actions to zero, explores its environment, records rewards at state action pairs (s, a), and updates the Q-table value at (s, a) according to the equation:

$$Q[s, a] \leftarrow Q[s, a] + \alpha(R(s) + \gamma \max_{a'} Q[s', a'] - Q[s, a]) \tag{3}$$

where α is the learning rate of the agent and γ is the discount factor, which defines how far ahead the agent looks. This may look familiar as it is an abridged version of the Bellman Equation.

1.3 Deep Q-learning

Q-learning poses a challenge with environments that have large state-action spaces as the agent will only learn on what it explores. Because of this, researchers have introduced Deep Q-learning (2), a variant of Q-learning where the agent uses a deep neural network to approximate values of the Q-table. Since neural networks are good function approximators and can often extrapolate well, this allows Deep Q-learning agents to act more optimally in unexplored states than their vanilla Q-learning counterparts.

The corresponding update equation, then, is as follows:

$$Q^*(s, a) = R(s) + \gamma \max_{a'} Q(s', a') \tag{4}$$

where Q^* is the updated Q-function and γ is the discount factor, same as before. With Deep Q-learning, this equation manifests itself in a Deep Q-Network that takes in a state and returns Q-values for the actions from the state space.

2 Environment setup

2.1 External tools

OpenAI have come out with an open-source API for testing reinforcement learning algorithms called Gym. We used this API to create our Pong environment with only a few lines of code. The environment allows easy determination of the current state, action space, and reward. Gym environments have two methods of returning the state of a game, one is returning the RAM usage by the game and another was to return an image representation of the game in a 210 by 160 RGB pixel image. Since we wanted our model to be as general as possible, we decided to use the image representation of the game and structure a CNN to learn the Q-function.

2.2 Network structure

We chose to follow the structure of the network that Google used in their paper. This was a convolutional layer of 16 8x8 filters followed by a convolutional layer of 32 4x4 filters, then a fully connected layer with 256 units, and finally a fully connected output layer with 2 units. The output layer has 2 units as this corresponds to the number of actions in our action space (up or down).

One thing to take note of that is different from most CNNs is that we must not perform max pooling after either of the convolutions. We chose to do this because, since exact pixel positioning is very important for an agent to learn in Pong, max pooling can result in clouding the model's perception of the location of the ball or paddle(s).

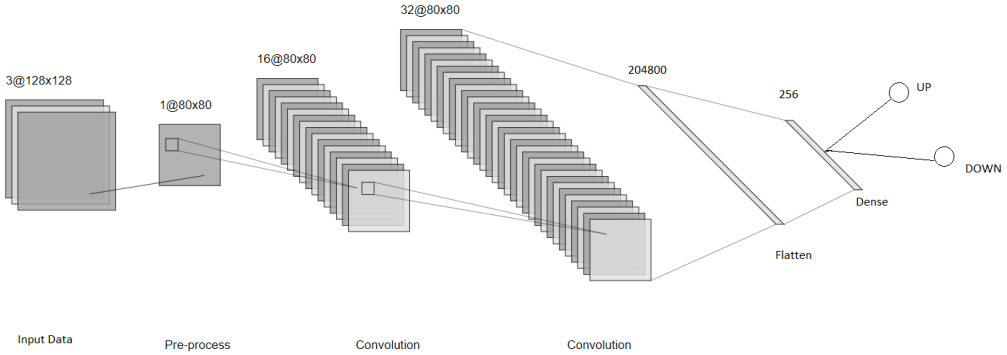
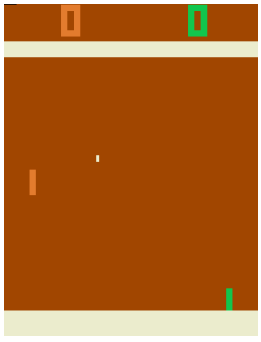


Figure 1: Visualization of network structure

2.3 Data engineering

The 210 by 160 RGB pixel images are represented as three-dimensional arrays of shape $(210, 160, 3)$. Additionally, since we want the model to be fed a temporal representation of the states so as to provide a sense of motion, we will pass the model the past 4 frames. However, the dimensionality of this input this poses a problem for us as, with the network structure we want to use, we have 825,803,570 trainable parameters.

For this reason, we chose to reduce the features of our input data. This was done by cropping the image to the game board, which changed the dimensions of each image to $(160, 160, 3)$. Then, we downsampled the image by a factor of two and grayscaled the image, leaving us with input arrays of shape $(80, 80, 4)$. This reduces the number of trainable parameters in our network to 52,441,906.



(a) Original image

(b) Downsampled image

Figure 2: Rendered images of the input data given by Gym and the processed equivalent our model sees

3 Challenges

Our current implementation of deep Q-learning has been shown to work, but due to time constraints and computational power we were unable to see if the model would learn effectively.

One challenge we ran into was integrating Gym with a Windows development environment, as the computer with an NVIDIA GPU we wanted to leverage was booting Windows.

Because we were experiencing extremely slow learning, we decided to attempt to implement a few strategies to try and speed up convergence of the Q-function.

3.1 Replay memory

Fitting batches of new agent experiences sequentially introduces some undesirable temporal correlations which can affect convergence of Q-learning (3). Thus, we implemented a Ring Buffer to hold past experiences that could then be randomly sampled from for the model to train.

This did, however, introduce some problems with memory consumption. Because our Ring Buffer did have to store many past experiences in memory, it ended up using nearly 32 GB of RAM, making us move to a Google Cloud environment where we had access to a machine with sufficient specifications.

3.2 Separate target network

In the update for Deep Q-learning, the model is continuously learning on its outputs. Because of this, it is common for the model to get stuck in feedback loops and slow or prevent convergence. Because of this, we implemented a target network. Every 1000 train batches, we would save the network weights and load them into another network. Then, the outputs of this fixed target network would be what the model would train on.

3.3 Parallel Gym environments

We ran into a bottleneck with Gym itself as the environments did not step as fast as we needed for the model to train. Gym and Python do not cater well to programmatic parallel environments, so we instead implemented a local server that would store the contents of the replay memory Ring Buffer. Then, we created separate processes for Gym Pong environments that pushed data to the local server. After this, we noticed an improvement with how the agent was able to explore.

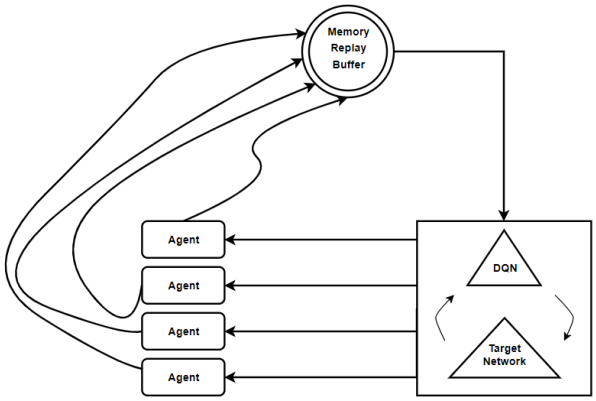


Figure 3: Visualization of the parallel Gym environments (agents) interfacing with the DQN, target network, and Replay Memory Buffer

3.4 Huber loss

Originally, we were using Mean Squared Error as our loss function for training the model. However, this turns out to be suboptimal. If the model experiences a large loss on a training batch, it will compensate by a large factor, but since it is training on its own output, this may easily lead to a divergent feedback loop. Thus, we decided to use the Huber loss metric clipping squared losses between -1 and 1:

$$L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2, & |y - \hat{y}| \leq 1 \\ |y - \hat{y}| - \frac{1}{2}, & \text{else} \end{cases} \tag{5}$$

4 Results

4.1 Instability of Q-function

We noticed that, in our implementation of Deep Q-learning, the average Q-values of our random samples was not generally increasing, as one would hope, but rather stochastically spiking and showing a potential overall decreasing trend.

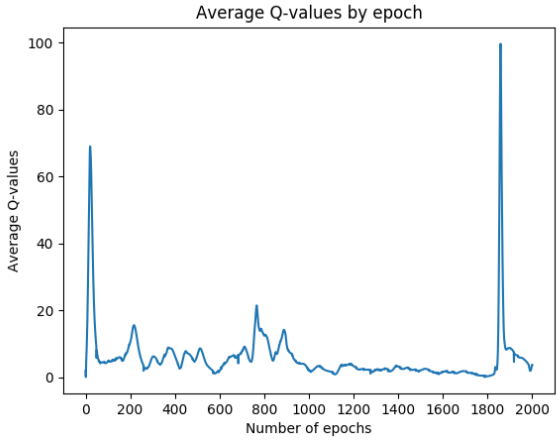


Figure 4: Stochasticity of the Q-function

This seems to be an underlying issue behind why our agent does not converge to an optimal policy. The reason could be a simple lack of enough training data, or, a mistuning of hyperparameters. It was mentioned before that Q-learning yields a globally optimal policy with a finite MDP, however, if the hyperparameters (learning rate, Q-function optimizer parameters, replay memory buffer size, etc.) are not tuned within a certain range, the Q-function may not converge.

4.2 Sparse rewards

Many reinforcement learning problems suffer from a problem called the credit assignment problem. In Pong, this manifests itself as non-zero rewards in the game being very sparse (1 if the agent wins a point, -1 if it loses). This introduces a shortcoming in how fast the agent will be able to learn, and is thus another suspect for the non-convergence of our agent.

5 Conclusion

Ultimately, despite all our attempts to accelerate convergence, we were not able to produce an agent that plays Pong well.

5.1 Team contributions

As a team, all individuals spent an equal amount of effort in programming and researching. Whenever any progress occurred for the project all team members were around each other that way everyone was on the same page at all times. Overall, we all learned a significant amount from self-learning, but now realize the true difficulty of applying machine learning to a real problem. Reinforcement learning based models require a large amount of time and, when feeding images as input, similarly large computational power. We feel that the project is a good project for students to tackle since Reinforcement Learning is not taught in the course, but the amount of time needed is a bit unrealistic for undergraduates to complete in the span for the project. We do however believe that this project was successful as we learned a lot about a branch of machine learning not taught in the course.

References

[1] Francisco S. Melo *Convergence of Q-learning: a simple proof*, Institute for Systems and Robotics, Instituto Superior Técnico.

[2] Volodymyr Mnih et al. *Human-level control through deep reinforcement learning*, Google DeepMind

[3] Ruishan Liu, James Zou *The Effects of Memory Replay in Reinforcement Learning*, Stanford University