Reinforcement Learning with Pong

CMPT 419 Project Report

John Liu, Jacky Lee

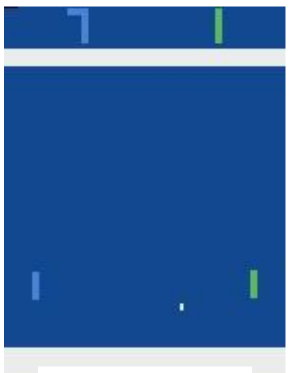
Source code
See https://github.com/bk202/Play_pong_with_reinforcement_learning for source code and instructions on running the agent on your device.

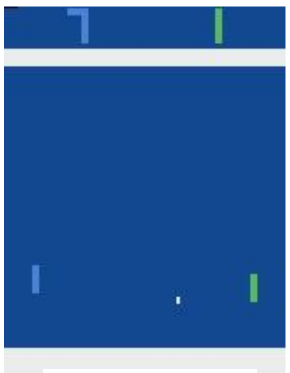
The task
The game of Pong is an excellent example of a RL task. The agent receives an image frame and decides whether to move the paddle up or down. The game simulator then executes the action and gives the agent a reward. Our goal is to move the paddle and maximize the rewards.

Preprocessing
We applied two preprocessing techniques to input frames to accelerate convergence of the model.

1. We use a helper function written by Andrej Karpathy, which down samples the frame and crops out unnecessary parts (score board and background) of the frame, preserving only the paddle and the ball. The intention of this is to reduce noise in input, so the model does not have to learn unnecessary features. The processed frame is a 80 by 80 array that preserves the necessary information of the game.

2. Difference between two frames are taken as the input to model. The motion of ball and paddles are detected by using this technique, again, the intention behind this is to help the model converge earlier. We believe convergence is still possible without this technique, tradeoff is a much longer training time.


Original frame 1


Original frame 2

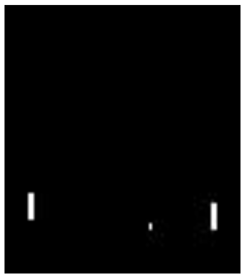


Figure 3 Processed frame1

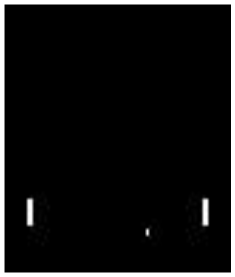


Figure 4 Processed frame2

Figure 5 Difference between processed frame 1 and processed frame2

Action space
Pong-v0 in Gym Atari has eight actions in the action space, [‘NOOP’, ‘FIRE’, ‘RIGHT’, ‘LEFT’, ‘RIGHTFIRE’, ‘LEFTFIRE’]. However, only ‘RIGHT’ and ‘LEFT’ are effective actions. They move the paddle up and down. As a result, our model has only 1 output, which predicts the probability of going up, activated using sigmoid activation function.

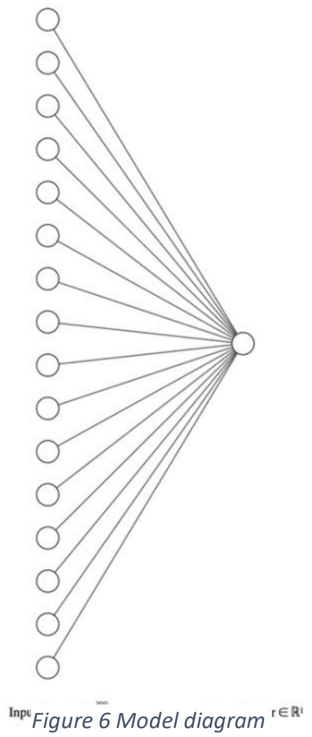


Figure 6 Model diagram

Model Architecture
There are numbers of architecture options for solving this problem. Convolutional neural nets and Long-short term memory neural nets are both capable of solving this issue. Both of which aim to converge earlier and a higher win rate. For this assignment, we chose a simple

architecture with a single 200 neurons dense layer for predicting the paddle's movements. This simple architecture has previously been proven to be competent for this task.

Optimization

We use log loss, also known as cross-entropy, for calculating the model loss.

```
self.epsilon = 1e-7
self.loss = tf.reduce_mean(
   (self.rewards * self.actions * -tf.log(self.network + self.epsilon)+
    (self.rewards * (1. - self.actions) * -tf.log(1. - self.network + self.epsilon)))
)
```

In mathematical notation, the loss function is expressed as

$$-\sum A_i * log(y_i|x_i) = \sum A_i * [y_i * -log(f(x_i)) + (1 - y_i) * -log(f(x_i))]$$

where y_i is the sampled action our model taken for frame$_i$ or x$_i$, $f(x_i)$ is the model's prediction for x_i, and A_i is the reward generated by taking action y_i. An epsilon value is added to the model prediction to avoid $log(0)$ calculations.

Reward is multiplied with sampled actions to classify whether an action is a good action or a bad action. We then multiply the result with the logged probability of the model's prediction. The model's high confidence in its prediction of a good action gives a low loss value, and vice versa. By minimizing the loss value with the Adam optimizer, the model's performance improves.

Intuitively, this is identical to supervised learning where there is a ground truth label for the input state, and we maximize the model's confidence in predicting the correct ground truth, except the dataset is constantly changing.

Discounted reward

Discounted rewards are calculated before using the rewards for loss calculations, as only the terminating frame produces a -1/+1 reward, other frames contain no information on indication of good/bad action, since their rewards are 0. By multiply the cross-entropy loss with 0-reward would produce 0-loss, causing gradient vanish problems.

Since each game is independent of each other, the discounted rewards are calculated independently of each game. We use the following formula:

$$R_t = \sum_k \gamma^k R_{t+k}$$

where R$_t$ is the discounted reward at frame t, and gamma is the discount factor (0.99 in our training config). Ultimately, each frame will have some reward of its consequence reward in range of (0, 1). The discount factor also acts as a long-short term reward factor, large gamma implies long term reward is preferred, while small gamma implies short term reward is desired.

Training

Each episode of training consists of 20 games, and a gradient update is performed every episode. At each frame, corresponding action and discounted reward are used into calculating loss and optimization.

The following plots demonstrate the training process:

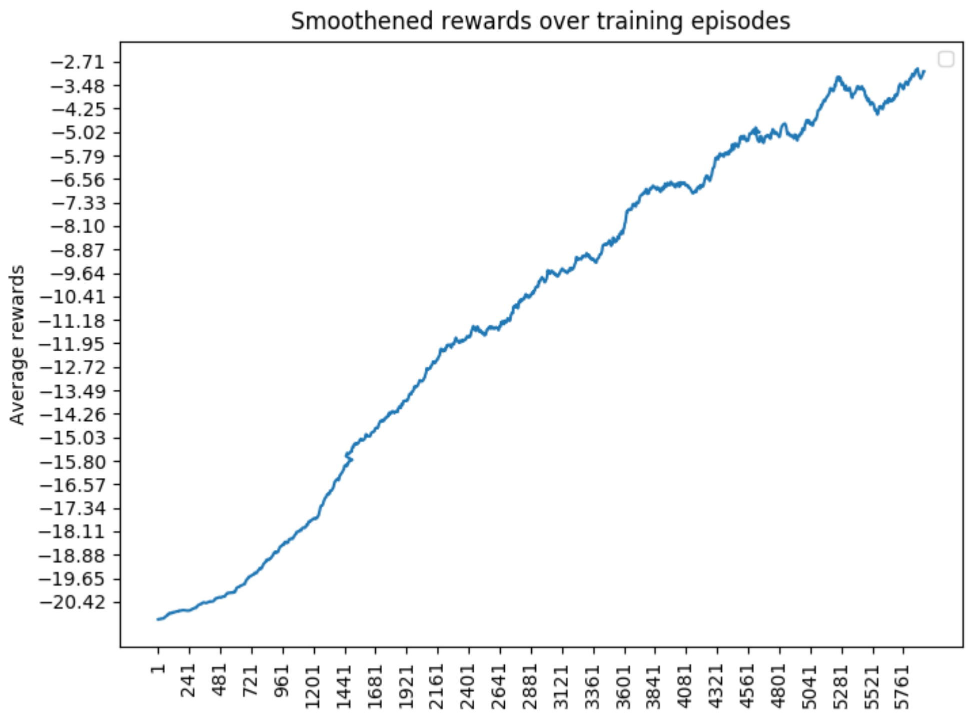


Figure 7 Smoothened reward over 5800 training episodes

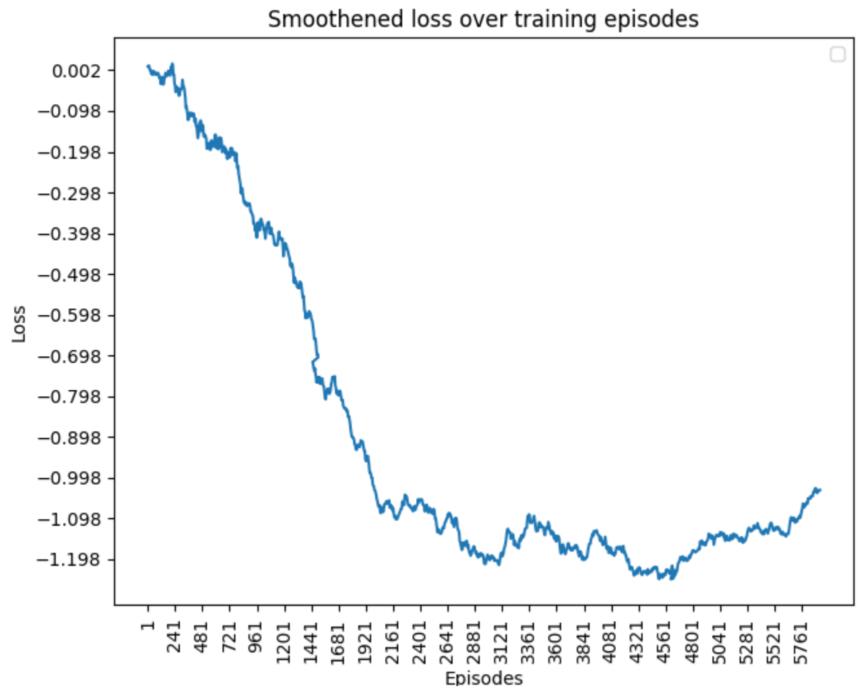


Figure 8 Smoothened loss over 5800 training episodes

The two plots of reward and training loss over episodes have shown a direct relationship between the two. As loss decreases, the in-game reward increases. Unlike supervised learning, the dataset is constantly changing, so there are no measurements of overfitting prevention taken here.

Problems and observations during training

The credit assignment problem: Suppose the agent performed a series of expertise level actions initially but fails to defend the ball in the last 20 frames. Then the network will mark that series of actions as "bad actions" and discourage gradients of this behavior, even if some of the moves were "good actions". Since the task for this project was simple enough, this problem can be overcome with thousands of episodes of training. However, this problem might lead to training divergence when the task is complicated enough where a series of random actions never generate positive rewards,

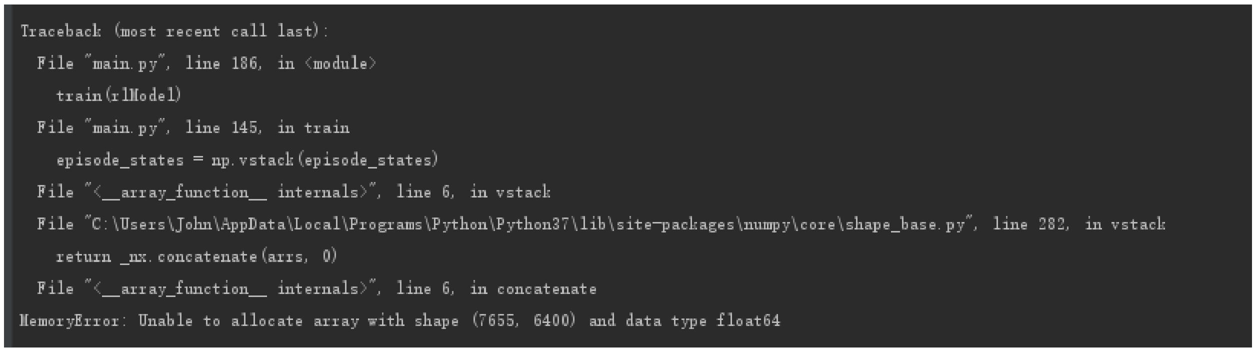
To alleviate this issue, there are two approaches:

1. Dense reward setting: by generating small rewards for achieving intermediate goals. For example, give a +0.1 reward every time the paddle hits the ball. The trade-off of this approach is that it may lead to the model learning the local minima (where the agent focuses on hitting the ball with the paddle) and not the global minima (where the agent focuses on scoring more points).
2. Imitation learning: by collecting data from expertise plays, the model will learn the collected data in a supervised learning setting. Once the model achieves expertise level, it moves on to an unsupervised environment and continues the training process. The

trade-off of this approach is that data collection from an expertise is required. In some scenarios, manual data labeling is also required.

Long training time: As the model performance improves over time, the length of frames for each episode increases significantly. As a result, the training time is much longer. This causes two issues:

1. Gradient updates become much less frequent. This can be alleviated by collecting data from multiple agents' game plays and then combining them to perform a larger gradient update. It is important to note that the other agents are only playing the game and not being trained.
2. Memory issues begin to occur. Buffer size for storing collected data grows as the game length increases. The can potentially lead to buffer overload. Batch size reduction or gradient update during an episode can help avoid these issues. If not, then adding more memory to the machine should do the job.



```
Traceback (most recent call last):
  File "main.py", line 186, in <module>
    train(rlModel)
  File "main.py", line 145, in train
    episode_states = np.vstack(episode_states)
  File "<__array_function__ internals>", line 6, in vstack
  File "C:\Users\John\AppData\Local\Programs\Python\Python37\lib\site-packages\numpy\core\shape_base.py", line 282, in vstack
    return _nx.concatenate(arrs, 0)
  File "<__array_function__ internals>", line 6, in concatenate
MemoryError: Unable to allocate array with shape (7655, 6400) and data type float64
```

Figure 9 Over-sized buffer causing memory problem

Citations

Mnih, V., Kavukcuoglu, K., Silver, D. *et al.* Human-level control through deep reinforcement learning. *Nature* **518,** 529–533 (2015) doi:10.1038/nature14236

Karpathy, A. (2016, May 31). Deep Reinforcement Learning: Pong from Pixels. Retrieved November 30, 2019, from https://karpathy.github.io/2016/05/31/rl/.