Final Project Write-Up

Crushing the Opponent in Game of Pong

Prepared by: Zhe Rao -250918939
Prepared to: Professor Scott Leith
Date: 2022-04-22

Overview

This was an extremely exciting project. I initially thought it would be pretty straightforward given that the code is provided for. It turns out, that understanding the code, and modifying the code to achieve a better result is not trivial.

Recall that the game of Pong's reward (scoring) is $21 - S$ if the player reaches 21 points first (where S is the points received by the opponent). However, if the opponent reaches 21 points first, then the reward (scoring) for the player is $S - 21$ (where S is the points received by the player). Note that during each round, the winning side will be rewarded with 1 point. For example, if during a game of 30 rounds, the player reached 21 first, then the reward for the player is 12 $(i.e. 21 - (30 - 21))$ whereas the reward for the opponent is -12.

In the book ("Mastering PyTorch", chapter 9), they claimed that they had trained the model to have the best running average of 7 (average of 100 episodes) after 2000 episodes (link: https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter09/pong.ipynb). After more than a week of careful analysis and training, I was able to train an agent that completely crashes the opponent in the game of Pong. It achieved a best running average of 20.32! In addition, I modified the code to test the agent (playing games without training the model). I ran 10 batches of games with 100 episodes each, the agent was able to achieve an overall average of 20.203! During the 1000 games played, the agent lost 7 times if "lose" was defined as scoring below 15 points in the match. In the traditional sense, it lost exactly 0 times to the opponent. Recall that this means, the opponent was able to score more than 6 points in only 7 rounds out of 1000 rounds played, and it never managed to win a single round. Please note that this result is better than the result in the paper.

While I have trained an agent with incredible power in this project, I must confess that I had some luck. There are still some questions I had regarding the process, methods, and some potential improvements. In this report, I illustrate my process of completing the project along with discussions about my thinking.

Please note that the code, outputs, plots are available in the following link:

https://github.com/ZheRao/Python-Projects/tree/main/Beat%20Atari%20Games%20with%20RL/Game%20of%20Pong

Please note that, there are two versions of the code, the one with the training outputs (long, long lists) is named, for example, "Solve Pong with DQN - Modified.ipynb" (non-modified version of the notebook contains some small modifications and it contains the initial training run, modified version of the notebook contains rest of the training); the corresponding version without the training outputs (for easier browsing) is named "Without displaying training output - Solve Pong with DQN - Modified.ipynb"

Also note that the code for testing the agent (i.e. just playing games without training) is in this directory https://github.com/ZheRao/Python-Projects/tree/main/Beat%20Atari%20Games%20with%20RL

Where it also contained my attempts at solving space invaders (the fully developed epsilon scheduling is implemented here)

Method

This is a modified implementation of the double DQN algorithm proposed by Mnih, et al. in their paper "Playing Atari with Deep Reinforcement Learning" in 2013 (Mnih, et al., 2013). The algorithm is an extension based on Q-learning. In the essence, the algorithm tries to find an exact value for each state-action pair so that the agent could choose the best action (i.e. with the best state-action value which represents the current reward plus the expected future reward) in every state it ran into. If the value estimator is accurate, the agent always performs the best action (in the greedy scenario) which would result in the best reward. Therefore, the reward function is pretty important, however, in this case, I did not come up with a better reward function (note that, in the paper, it was kept as between -1 and 1, but I used the round-scoring for the reward function).

Representing States: Skipping-Frame Technique

In this game, the state would be the pixels values representing the current game situation. For example, it captures where the ball is, where the opponent is, or where the agent is. I have included an example of the state below (the image is taken from the book)

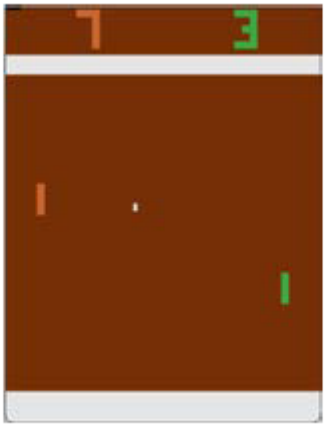


Figure 9.2 – Pong video game

Since in this single image, there is no way to tell the dynamic of the ball (i.e. is it going forward or backward), there is no way to determine an action value. As such, we used 4 consecutive frames collectively to represent a state.

The traditional way of retrieving a Q value is through table lookup. However, considering the enormous amount of states possible, the Q table would be too much to keep track of. Therefore, we used convolutional neural networks, the universal approximator, to estimate the Q values.

Let's take a look at how large the table must be to possibly capture the states. After resizing the frame, it became an 84 * 84 image, assuming the player and the opponent's position can only move among 80% of the vertical pixels (i.e. 67) and the ball can move among 67 * 50 pixels (i.e. the center portion), a frame can have 67 * 67 * 67 * 50 = 150,381,150 different values. Now, assuming in the next frame, the opponent or the player can have 3 different positions (i.e. move up, stay, or move down). The ball can be in 1 different position for the third and fourth frames (i.e. if the ball is going straight left, and it is at position (30,25), the

next position will be (30, 24)). Because given two frames, the motion of the ball is deterministic. On the other hand, for the second frame, the ball can be in 6 different positions (assuming it can't go straightly up or down). Now, the total number of values for 4 frames is 150,381,150 * (3*3*3) * (3*3) * (3*3) = 3.3×10^{10} ! Even given the computational power today would still find that table enormous!

In order to cope with this change, the **step** function is overwritten. The function takes *action* as input, takes the action 4 times in the current environment, and append the 4 resulting observations (i.e. "next_state") to a buffer. It uses max-pooling on the final 2 observations and produces the next_state that the algorithm uses to advance.

The Network Architecture

The original paper uses 2 convolutional layers with 16 size-8×8 filters and 32 size-4×4 filters respectively. In our implementation, we used 3 convolutional layers with 32 size-8×8 filter, 64 size-4×4 filters, and 64 size-3×3 filters respectively. Our more complexed network structure could imply higher capabilities to find meaningful representations of the input (i.e. states) that can result in more accurate output (i.e. Q value). It should be noted that the previous argument is based on proper training.

The original paper uses 256 neurons in the final fully connect feed-forward layer, we used 512. Note that there is no non-linear activation after the layer, because the output is linear in nature. However, if we wish to cap the values between $[-1, 1]$ or $[0, 1]$, we could apply a hyperbolic tangent or sigmoid activation in the end. (I didn't try it, just a thought)

It is noteworthy that the problem of overfitting is not apparent in this case. Since we are directly observing the training loss or having a validation loss, it is problematic if the learning rate is too large so that the model overfits. I believe this is one of the culprits that caused the moving average to go down (i.e. stopped learning). Imagine that during the initial phase of training, the behavioral model is extremely bad (because the weights are randomized), the learning rate could be big so that every iteration of weights update would make the model more accurate. However, during the later phase of training, while the model is somewhat accurate, a large weight updates might make the model worse. For example, if the "true" weight value is 2 (which is never known), the current weight value is 2.01, the learning rate is 0.1, and the gradient is calculated to be 10. The updated weight would be $2.01 - 0.1 * 10 = 1.01$. This is further from the "true" value. On the other hand, if the current weight value is 4, the update would be $4 - 0.1 * 10 = 3$, which is closer to the "true" value. Therefore, the learning rate should be adjusted according to the phase of training. This is one of the modifications that I made to facilitate better training. However, what the learning rate should be reduced to, or start with, there is no explicit guide, I used some training as experience and make modifications accordingly.

The "Learning" Phase

To ensure that the data used to train the network is identically and independently distributed, we used a replay buffer to keep 40,000 tuples of (state, action, reward, next_state, finish). During each training phase, the model would be trained on 1 batch of size 64 tuples sampled randomly from the replay buffer. I considered increasing this to size 128 in the later phase of training, it made the training much slower, and I just gave up on that.

In addition, the buffer is constantly updated (a new tuple replaces the oldest tuple). Ideally, the buffer size should be big so that a variety of samples could be produced. The original paper uses 1 million as the

size of the buffer, but my computer can only handle 40,000, and it takes 90% of my memory (anything larger will crash my computer, speaking from experience). This is a potential downfall to my training. To know concretely how small the size 40,000 is, I printed out how many tuples each around would produce. In the beginning, it was short of 300, (it took the opponent 300 iterations to beat the agent in one round, which is like $21 - 0$). As model learns , the number goes above 1,000 (e.g. score $21 - 18$). Later on, the number of iterations went down to 550 (e.g. score $5 - 21$). I would have to imagine what the training would be like if the buffer size was 1 million.

Here, we used off-policy learning because using a single model for training and producing output would result in instability in learning. Therefore, two models with the same structure are defined, the target model is completely greedy while the behavioral model is epsilon greedy, the behavioral model is learning toward the target model. In the book, the weights of the target model are updated (i.e. copied from behavioral model) every 5000 iterations. I later suspected that this is too frequent (i.e. update every 5-10 episodes), I increased it significantly. I also threshold it so that later phase of training would update less frequently. In addition, I came up with the idea that makes the update only if the current reward received by the model is higher than the reward received at the time of the last update. This ensures that if the model deteriorates due to overfitting, the target will not suffer from that, which is the key to bringing the overfitted behavioral model back on track. I have not implemented it, but I suspect this could stabilize the training even further, especially in the later phase of the training.

Customized Epsilon Decaying Schedule

The epsilon decaying schedule indicated in the book is the following equation

$$eps = min + (max - min) * e^{-\frac{frames+1}{DECAY}}$$

Where min, max are the hyperparameters defined as the maximum and minimum epsilon value during the training. $frames$ is the number of iterations processed so far (usually, a round takes about 700-900 iterations). $DECAY$ is another hyperparameter specified to control the rate of decay

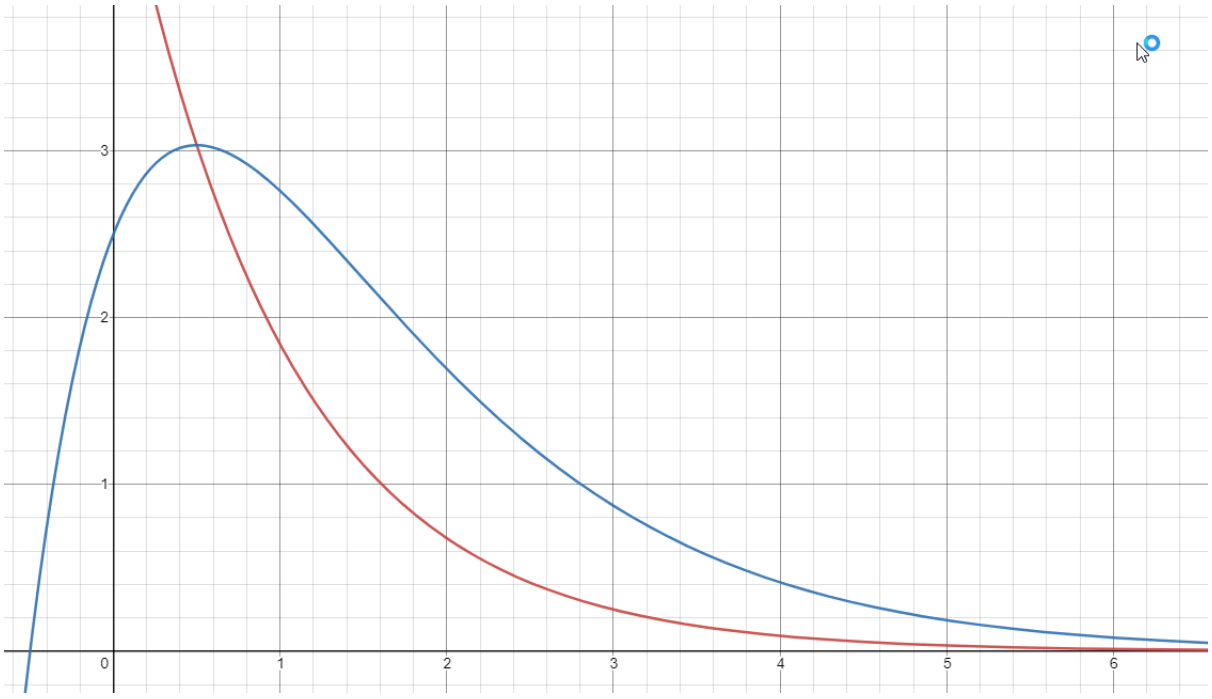
If we consider $\frac{frames+1}{DECAY}$ as variable x, the decaying schedule is similar to the following (I assume $max - min = 5$, which is impossible, but it can illustrate my point)

However, during training, I wish the decaying to be much slower when x is above maybe 1.5 (so that). I therefore considered the following piecewise function

$$eps = min + (max - min) * e^{-\frac{steps+1}{DECAY}}, if \ \frac{steps+1}{DECAY} < k$$

$$eps = min + (max - min) * e^{-\frac{steps+1}{DECAY}}\left(\frac{steps+1}{DECAY} + (1-k)\right), if \ \frac{steps+1}{DECAY} \geq k$$

This would result in



For $\frac{steps+1}{DECAY} < k$, the decaying schedule is as the function in red, and otherwise, the decaying schedule

is as the function in blue. This function is very flexible, for example, if I want the slowing decay happens at a later time, I simply increase k.

Through carefully planning, I could control exactly at what epsilon value would the slowing decay starts, at what episode (approximately) would the slowing decay starts. Note that the fully developed mechanism is in Space Invaders code. Here is an example of my planning

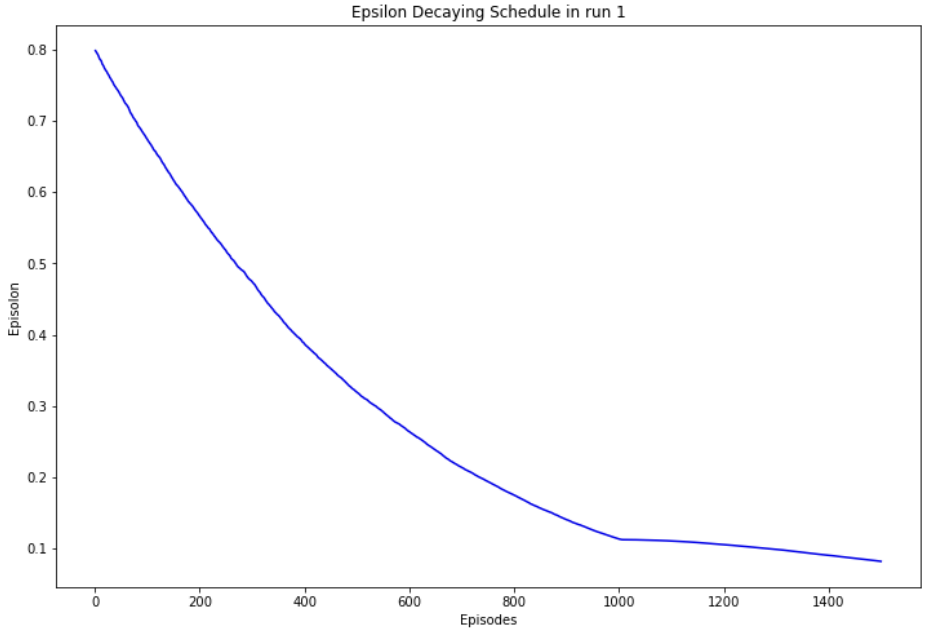
```
EPS_START = 1
EPS_FINL = 0.005

EPS_DECAY = 200 * 500 # epd equals EPS_DECAY approximately 500 episodes, assuming 200 frames per episode, slowing start are 1000 episodes
DECAY_TH = 2 # start slowing decay at rate = 2, eps = 0.14

DIFF = 20
DIFF_P = 0.8
BREAK_TH = 1000
```

The calculation is as follows:

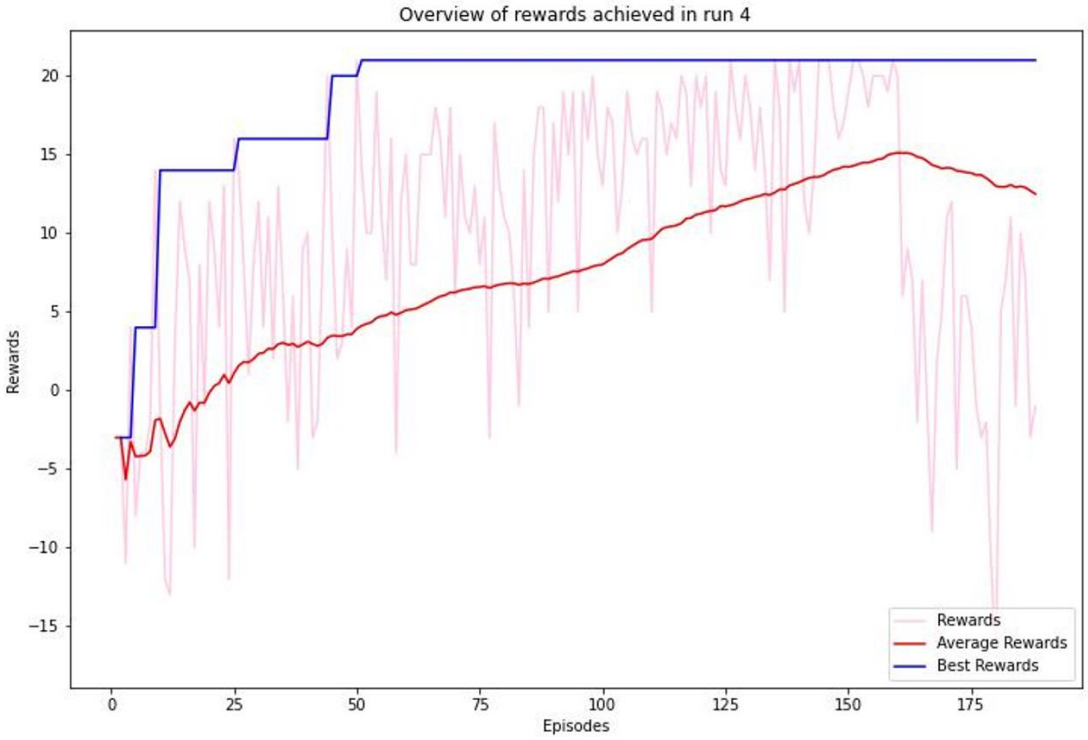
$$eps\ at\ threshold = 0.005 + (1 - 0.005) * e^{-2} \approx 0.14$$

$$\frac{episodes * steps\ per\ episode}{EPS\_DECAY} = \frac{1000 * 200}{500 * 200} = 2$$

And it is working to perfection! (Except the steps per episode is only approximation). The



My initial attempt was to threshold $DECAY$, and reduce it after a certain episode. However, this can cause a sudden rise in epsilon because the second part of the equation certainly increased by a lot. As a result, the running average tumbled and the training was seized (I modified the code so that if the difference between the current running average and the best average, then break the training; I later changed this to be the percentage of best average to account for different scale of rewards in different games). The sudden change in the running average reward is as follows



Value of Epsilon

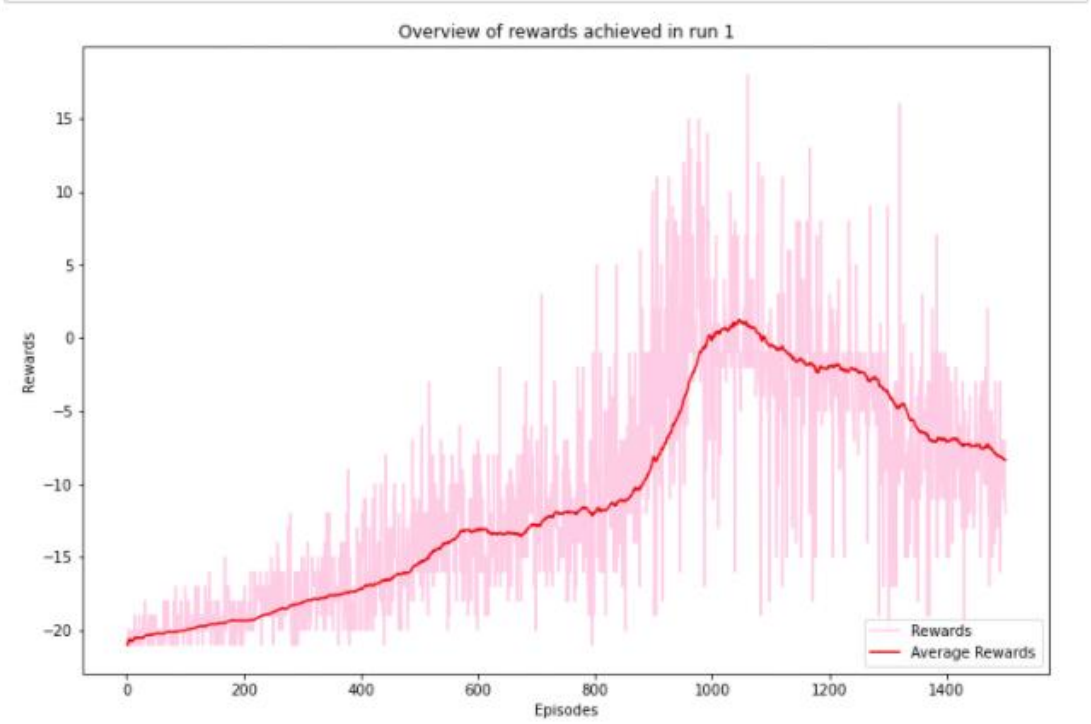
It might seem strange that the epsilon is extremely small, the minimum was set to 0.005 in the first run. This is because there are many iterations in a round. If we assume there are 800 iterations, then the expected number of random actions is 4. Based on my observation, any random actions could result in loss of the round.

Result

I trained in total of 6 runs, each run consists episodes ranging from 1,000 to 1,500. Due to some errors in my code, only 4 of them are saved. The training time for 1 run of 1,000 episodes is approximately 6 hours. Luckily, my computer is equipped with NVIDIA GPU, or the training time would be 15 times slower!

Run 1

The first run is purely done with the code in the book without much modification. The result is as follows:

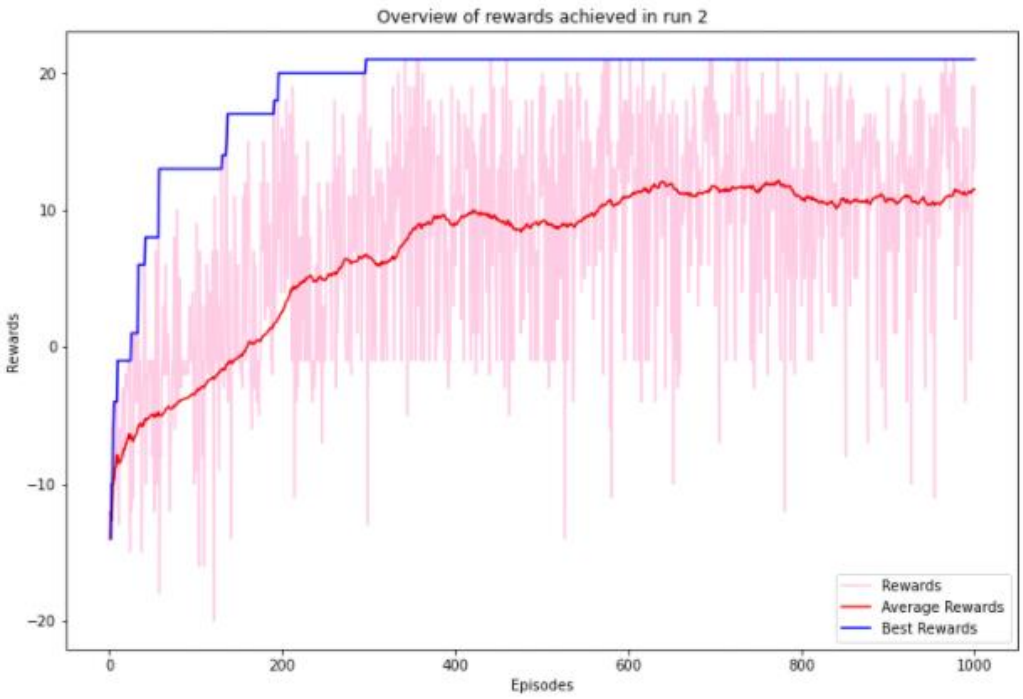


Notice that it crossed the 0 thresholds at around 1020 episodes, this means, in the past 100 episodes, the agent was able to beat the opponent. However, unlike what's shown in the book, the agent's performance started to deteriorate. The agent failed to learn anything from the final 450 episodes, which inspired me to put a threshold in place to seize training if the running average rewards continue to deteriorate.

Run 2

To prevent previous undesired results, I increased the frequency of updating target model weights from 1,000 iterations to 8,000 iterations, and 16,000 iterations for episodes after 300. This updates the target model weights by approximately 10 and 20 episodes respectively. This should stabilize the training. In addition, I changed the parameter from 100,000 to 25,000, which makes epsilon decay 4 times faster. This would improve the agent's ability to score greatly (at the expense of more exploration). I changed the replay buffer size from 10,000 to 40,000 (I wished to increase more, but my hardware doesn't allow it), which would improve the independence and variety of sampled data. I changed the learning rate from 1e-4 to 1e-5, then set it to be 6e-6, 2e-6 after 100 and 300 episodes. This would prevent the model from overfitting (learning more steadily). Finally, I changed max epsilon from 0.8 to 0.1, min epsilon from 0.05 to 0.01.

The result is as follows:



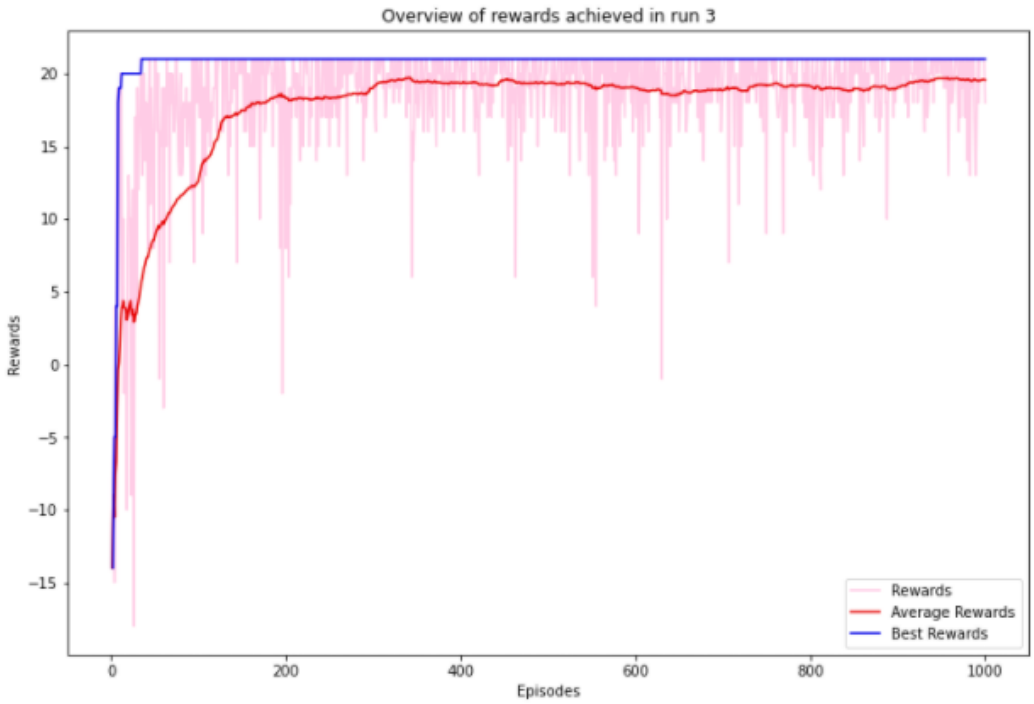
The agent was able to beat the opponent on average just after 200 episodes of training. This is mostly due to the change in epsilons. Now it decays from 0.1 as opposed to 0.8, which enables the agent to take more “optimal” or “suboptimal” actions, which resulted in higher chances of winning.

The reward seems to converge at around 10 points in this training run, I implied that the majority of my analysis was correct. The best practice I believe was to tweet the hyperparameters one at a time and observe their effects (e.g. just change the size of the replay buffer while keeping other hyperparameters constant). However, due to the limited computational power and time window, I just experiment with them all at once.

Run 3

I continued to adjust the hyperparameters by a lesser amount from my previous analysis except 1) I decreased the epsilon decaying rate because I believe it was too fast in the second run (maybe that’s the reason we stuck at an agent with suboptimal behavior) and 2) I increased the minimum epsilon for the same reason. I wished that the decaying rate of epsilon would be slower in the later stage of training, and this was the first attempt at controlling the decay schedule.

In addition, as the professor indicated in class that penalizing large Q values by adding 0.05 * Q_val in the temporal loss calculation could facilitate training, I added the term to the previous mean squared loss. I was able to achieve an extremely well result



It converged at around 19.4 points! The agent was already able to obliviate the opponent. However, I wish to see if I could break the 20-points mark. That would mean the opponent could only score 1 point on average!

During 1000 testing rounds, this agent was able to score an overall average of 19.103 points. The agent scored less than 15 points in 67 of the rounds and lost 4 rounds in total (i.e. scored below 0). It is a decent performance, but I wonder if I can train a better agent

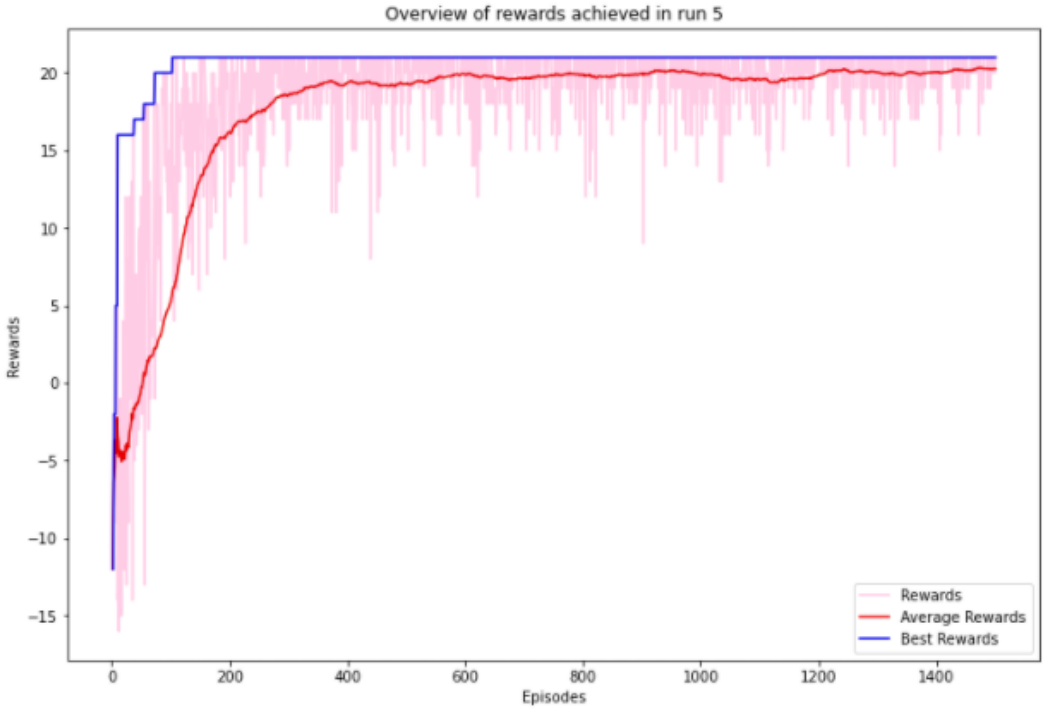
Final Run

Due to the diminishing marginal returns, it seemed particularly hard to outperform the previous performance

After several runs of experimenting, the best average rewards were always approaching 20 but never surpassed it. I finally was able to hit the best-running average of 20.3 points with the average converging at 20.2! The final hyperparameters that I used are as follows:

- Max epsilon: 0.1
- Min epsilon: 0.002
- Epsilon of final 100 rounds: 0
- Learning rate: 1e-5, 4e-6, 6e-7, 1e-7 with thresholds at episodes 100, 300, 1000
- Epsilon decaying factor: 75,000
- Iterations before learning: 4,500
- Frequency of copying and pasting weights to target model: 15,000 and 30,000 iterations with thresholds at episodes 300
- Additional loss function component: 0.1 * Q_val

The result is as follows:



This agent completely crushed the opponent in the test games!

Discussion

My Hypothesis was True!

My previous agent-training experiences involve a single run of training (e.g. one run of 2000 episodes). However, in this case, achieving the best possible involves multiple runs of training. This implies that I have to save the agent so I could continue the training at a later stage. I did some search online and people used to save the environment and the agent. In this case, I didn't instantiate an agent explicitly. I thought about it for a while, my hypothesis was that the behavioral model is the agent because the agent's job is to choose an action given state information, and the model produces the action values for states so that actions can be taken based on those values. Therefore, as long as I save the model weights, the agent is saved, and training can resume at a later stage.

It turns out that I was correct! I was thrilled because otherwise, the first 6 hours of training would be for nothing! And I don't think it is safe to train for 30 hours at once (my computer will melt because the CPU temperature was above 80 degrees Celsius during training!)

Importance of Exploration

One huge advantage of my training methods compared to training all episodes at once is that my methods allowed for exploration each time training starts. I deliberately set the max epsilon value to be well above the epsilon value of the last training episode, so that the agent could explore more. I thought about if

this exploration could make the weights update worse because the target model always produces the best action value, I am comparing that to the non-optimal action value produced by the behavioral model, which could result in updating the weights in the wrong way.

Turns out, I was wrong. What happened was that the behavioral model produces non-optimal action values for the current states while the target model produces optimal action values for the next states. So this is saying, "okay, I am going to take this bad action, and I am going to take the best actions thereafter, if the next state (that this bad action takes me to) has a high optimal action value, that means I should adjust my approximation for this bad action because this could lead to good outcomes (since the optimal action value for the next state is large, I will have a good reward I take the best actions from the next state, so the action value for this 'bad action' should be higher)". This could open to a variety of strategies that goes beyond the obvious. For example, in the game of Pong, instead of trying to catch the ball every time, the agent started learning to control the rhythm and discovered the weakness of the opponent, and was able to crush the opponent by exploiting that weakness.

I have uploaded the weights and code to play the game called "Test Agent – Play Game.ipynb". It will be interesting to test it out

Modify Target Update?

I am thinking that if the behavioral model overfitted, and the weights were copied to target model, it would result in unstable learning and produce worse results. To deal with this problem, I came up with the idea to keep track of the running average rewards when last time target model's weights were updated. Then the next time to update the weights, only perform the update if the current running average is higher than the last running average rewards.

I have not implemented this strategy, but I am interested to see if this helps

More Complex Neural Network Structure

I also wonder if I properly train a more complex neural network to approximate the Q values could approximate the values more accurately so that the agent could completely crush the opponent – average reward of 21 points!

Struggle

Jumping from Pong to Space Invaders

It was both more and less straightforward to go from solving Pong to solving Space Invaders than I had imagined. Firstly, the structure of DQN is not varied at all. However, due to the different rewards systems,

the rewards were too large that they influenced the network's ability to learn. Therefore, I decide to clip the reward. The learning before clipping is indicated below

And here is the learning after I clipped the reward

The change was not significant. However, I believe that there should be a difference because the huge difference in loss calculation would result in large gradients during the weights update phase, which ultimately caused the gradient to explode because the gradient of the loss with respect to the weights in the

earlier layers of the network would be the product of gradients of the loss with respect to the weights in the later layers of the network. If the loss is huge, gradients are large, multiplication of large numbers is even larger! To illustrate, only consider the derivative of the MSE loss function: $2 * (Q_{expected} - Q)$, when rewards are huge, the Q values would be huge (because they are expected future rewards), the loss will be huge, and then the derivative will be huge! Causing the gradient to explode

I originally thought about just clipping it during the loss calculation phase because that would end up with much fewer clipping operations (as opposed to clipping all the rewards). However, this strategy seemed to cause a huge increase in memory usage, which force me to forfeit this approach. I suspected that, when rewards are modified right after they are received from the environment, the first in first out mechanism (during the buffering phase) would prevent them from taking additional memory. However, if I change them during loss calculation (which would be torch tensors), they could take up additional memory which would not be removed.

In addition, I tried to use the weights that crushed the Pong game as weights initialization for the model in this case. However, it failed to learn. I believe that "fine-tunning" must be done with similar tasks

A side note: to ensure that the reward during gradient and loss calculation is indeed clipped, I printed out the reward, Q values, expected Q values and loss, they appeared to be as expected

Here is the reward for a batch

```
tensor([1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000, 0.0000,
        0.0000], device='cuda:0', dtype=torch.float64)
```

And here are the Q values and loss

```
tensor([[35.5779, 38.5486, 26.0717, 32.8696, 40.2007, 54.8405, 34.7168, 32.3054,
         43.1482,  5.9836, 37.5183, 44.9259, 32.3949, 40.3208, 36.2162, 48.8236,
         25.8553, 42.2231, 26.1596, 44.0961, 28.9655, 13.3579, 40.1591, 63.7988,
         31.2917, 25.6186, 33.8181, 39.8007, 32.5357, 38.0560, 44.9019, 28.3564,
         39.1546, 32.6258, 44.3575, 28.0759, 49.8570, 26.2581, 33.0442, 26.4317,
         37.4466, 36.1532, 35.1835, 37.1591, 50.6695, 38.9193, 34.1220, 47.0835,
         34.5338, 26.2522, 27.6360, 31.1637, 35.7556, 29.9022, 47.2489, 27.0568,
         38.7104, 28.7505, 27.9342, 41.1257, 41.8234, 41.2493, 54.9236, 42.0405]],
       device='cuda:0', grad_fn=<SqueezeBackward1>)
tensor([[34.5708, 41.6775, 22.3683, 31.6179, 39.7799, 54.1045, 34.3438, 31.7662,
         45.2799, 10.2197, 36.4348, 44.7517, 36.7117, 41.1146, 36.9112, 49.4232,
         25.3081, 43.1988, 26.2787, 45.6800, 28.8411, 16.8200, 39.9647, 61.3661,
         31.6628, 25.3081, 35.1970, 37.3112, 32.3399, 39.4407, 42.8359, 26.6664,
         39.3005, 32.5808, 44.3759, 30.9778, 51.1429, 26.3102, 30.1370, 28.2064,
         39.1732, 34.8985, 35.8228, 36.3623, 50.0283, 39.6650, 34.7924, 47.7892,
         34.4235, 26.2787, 27.3235, 32.3372, 37.0746, 29.8592, 47.2175, 27.1102,
         37.1808, 22.7750, 28.7864, 43.6087, 40.8605, 42.7296, 55.2809, 41.8012]],
       device='cuda:0', dtype=torch.float64, grad_fn=<AddBackward0>)
tensor(4.7918, device='cuda:0', grad_fn=<AddBackward0>)
```

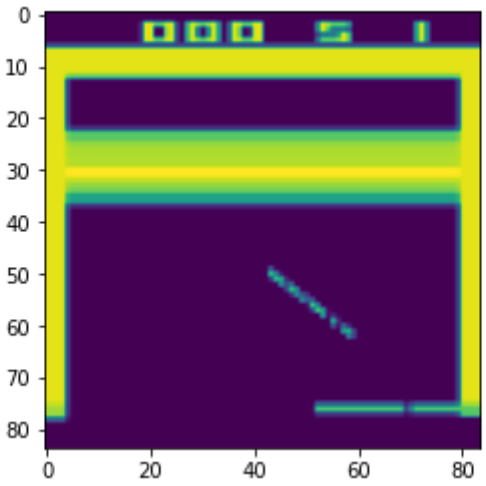
The best running average that I am able to achieve was a little above 200 points

Jumping from Pong to Breakout

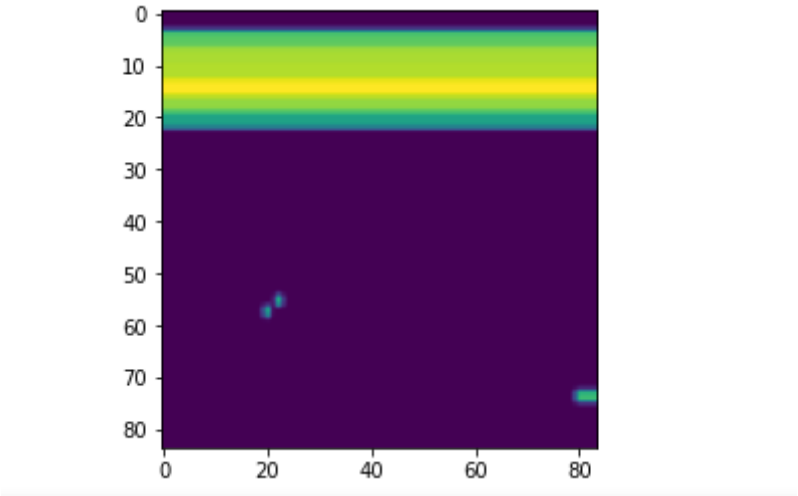
I initially struggle a lot because my model simply can't learn. Here is the first 1500 episodes:

And the following 1000 episodes never managed to pass running average of 5!

Later on, I realized that the image feed into the model is very confusing. It contains the sores and text on top which could interfere and confuse the network when it is trying to find meaningful abstractions of the image. The pre-processed image from their method is as follows: (note that I made a mistake in the code and it produced motion of the ball for 12 frames instead of 2: that's why the trajectory is so long)



After I adjusted the transformation and carefully cropped the image, here is the output



It was simple to exclude the text on top, I just take the bottom 84 pixels horizontally. To exclude the boundaries, I need to take the 84 center pixel values, which I used the formula: $center\ pixel \pm \frac{84}{2}$. Then I carefully find the correct down0sampling size to make sure the boundaries are not included, but all the boxes in the game are not neglected.

Small mistakes led to waste of training runs

One of my biggest struggles is making sure that all the codes associated with my modifications have been properly modified. For example, After I figured out a proper exponentially decaying schedule for epsilons, I wish to plot the schedule, I modified the code so that epsilons for each episode is stored, and the list was returned. However, once I started the training, I realized that I forgot to assign the returned list to an array. That was mid-way through the training (i.e. 3 hours). It made me very upset

In addition, small errors in the code could ultimately cause the loss of carefully stored information. For example, I decided to store the weights of the model at the end of the training run, and the syntax was $torch.save(model.state\_dict(),\ file\_path)$, however, I typed "," instead of "." after $model$. After the training was completed after 5 hours, the error popped, not only did I lost the model weights at the end, I also lost all the information that I wanted the training function to return so that I could make pretty plots visualizing the training (e.g. rewards, average rewards, best rewards received during training). I was extremely upset with this mistake. The mistake is indicated below

```
----> 12          save(main_model,state_dict(),f"Weights/{env}_last.pth")
     13      print(f"EP {ep}, curr_reward: {ep_reward}, best_reward: {log.bes
t_reward}, running_avg_reward: {round(log.average, 3)}, curr_epsilon: {rou
nd(eps, 4)}")

NameError: name 'state_dict' is not defined
```

Getting the tensors to the right data type

I also spent quite some time trying to get all the tensors to the right type. Some of them need to be float, some must be int.

Unsatisfying results

I struggled when the results were not what I was expected. For example, during the first training run, the agent deteriorated for the final 500 episodes, which wasted a lot of training time and computational power

Weird errors

In addition, I struggled with weird errors that just happened out of nowhere (i.e. I couldn't figure out why). A particular example is the error below

```
----> 23         frame = cv2.cvtColor(observation, cv2.COLOR_RGB2GRAY) # turn the im
age to gray
     24         frame = cv2.resize(frame, (self._width, self._height), interpolation=cv
2.INTER_AREA)
     25         return frame[:, :, None]

error: OpenCV(4.5.5) :-1: error: (-5:Bad argument) in function 'cvtColor'
> Overload resolution failed:
>  - src is not a numpy array, neither a scalar
>  - Expected Ptr<cv::UMat> for argument 'src'
```

So for some reason, the grey scaling the image suddenly had an error. After I copied all the code to another notebook, it ran smoothly.

Reference

Jha, A. R., & Pillai, G. (2021). Mastering pytorch: Build powerful neural network architectures using advanced Pytorch 1.x features. Packt Publishing.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013, December 19). Playing Atari with deep reinforcement learning. arXiv.org. Retrieved April 21, 2022, from https://arxiv.org/abs/1312.5602