
 This repository Search Pull requests Issues Gist

mimoralea / king-pong Unwatch 1 Star 0 Fork 0

[Code](#) [Issues 0](#) [Pull requests 0](#) [Wiki](#) [Pulse](#) [Graphs](#) [Settings](#)

Branch: master king-pong / report.md Find file Copy path

 mimoralea doc: Change list indentation 8d80b2b 13 seconds ago

1 contributor

327 lines (201 sloc) 28.1 KB Raw Blame History

King Pong

Deep Reinforcement Learning Agent

Definition

Project Overview

For this project we will be creating an agent that can beat a hard-coded CPU player in the classic Pong game directly from raw pixels. This is particularly challenging because the agent would have to read the game from raw pixels, and this creates a large amount of states that has been the liability of reinforcement learning.

Also, this approach to solving the game of pong is better than the hard-coded version, not only for the potential of better performance, but also for the generality of the solution, meaning the same agent with slight modifications would be able to solve other similar problems.

Problem Statement

The tasks for this project are very challenging. First we need to create a Pong game simulator, then we need to apply Computer Vision to get the raw pixels from the game state and preprocess the images, Deep Learning to reduce the state space of the problem, and Reinforcement Learning to allow our agent to learn an optimal policy from scratch.

The following tasks will have to be completed before this project can be called successful.

- First we need to create a Pong simulator.
- We will need to provided a hard-coded CPU player to use for our base comparison.
- We have to read the raw images from the UI.
- Additionally, we will have to create a Deep Neural Network that can interpret and extrapolate efficiently for unseen game states.
- Lastly, we need to implement the deep reinforcement learning agent to be able to be competitive against a hard coded CPU player.

Metrics

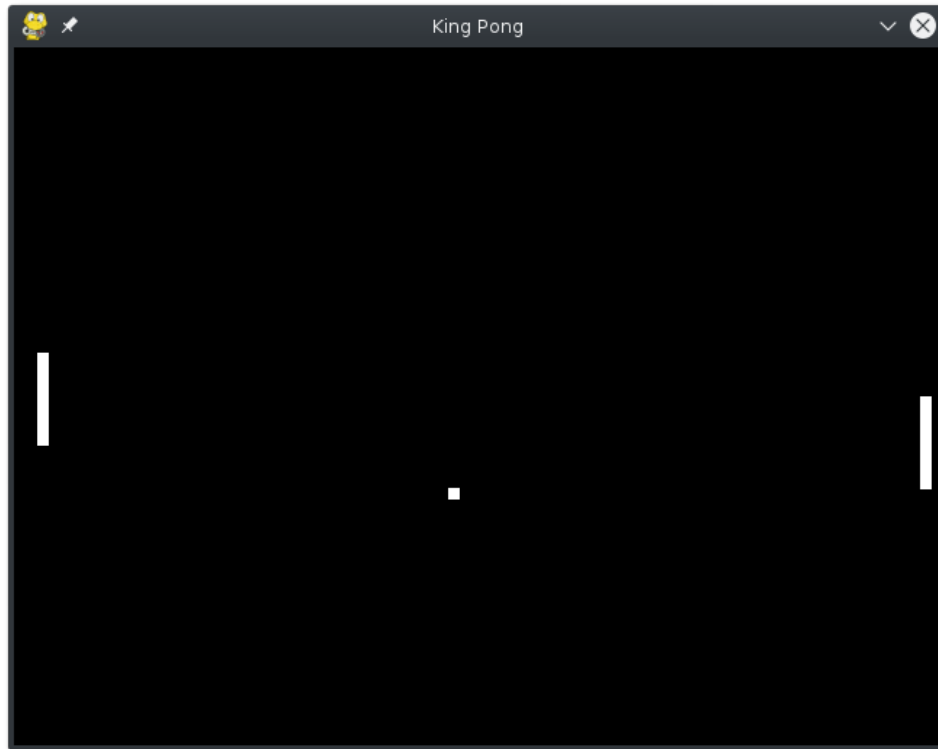
To be more precise we will have to come up with a rating for the agent. Simply, we will match the fully trained player against the hard-coded CPU, stopping the match when **the first player reaches 100 games**. Each game will be 3 points. That is, the first to score 3 points wins a game, the first to win 100 games is the **King**. Our goal is to get to the 100 points first, but hopefully we can keep the CPU player under 70 points.

Note that 100 games are not including training games. The agent has to train for millions of games before it learn a robust policy. The 100 games metric is only used to test a fully trained agent.

Analysis

For this Deep Reinforcement Learning problem there is no data set per se, but there is, obviously, that which the agent will consume.

This is how the game of King Pong looks like:



Data Exploration

The game is a pygame window of 640 by 480. The agent reads the window using OpenCV into an array and then shrinks the image down to a 80 by 80 array. Though the shrinking also causes resolution distortion for a human looking at the images, the agent is able to extrapolate and find patterns that help him generalize into the correct states.

Not only the image is re-sized but also compressed into a gray-scale image which values are 0 or 255. And then the values are clipped to 1. So the input image gets converted from a 640x480x3 with values ranging from 0..255, down to a 80x80x1 with values in the range 0..1.

Now, think about it, although the image will be read with all color channels, it is important to reduce the input space as much as possible. The first way of doing this, is to reduce the width and height of the images to 80x80, the other way is to get reduce the amount of color channels of the image. Additionally, white and black colors are normalized which might further improve the performance of the Agent.

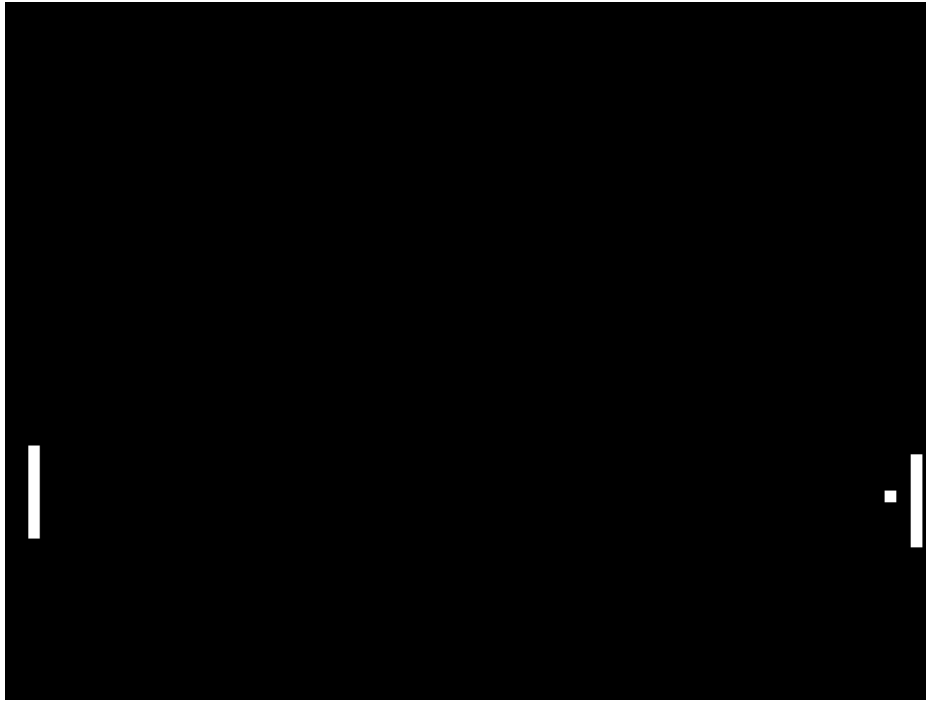
However, you might be wondering, but how does a single image tell me the direction in which the ball is going? In fact, this creates an additional challenge. To solve this problem we expand our input array to contain 4 of this images at a time. The sequence of images does indeed tell you the direction and speed in which the ball is traveling.

One additional feature that this problem may seem to have is translational invariance. In specific, the ball moving in a specific direction and speed towards one area may seem to be the same as the ball moving on, for example, exactly the opposite speed and direction. Although it might seem worthwhile to implement logic for this specific problem, since the space is pseudo continuous, the translation of the images get too complex. Also, the robustness of Deep Learning is such that our best bet is to just trust the network and not add efforts that are specific to this problem but instead try to generalize even further in order to create a much robust Deep Reinforcement Learning Agent.

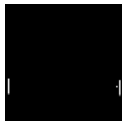
Exploratory Visualization

Here is a visual of how the images get processed through the flow of the app.

First, the image gets captured directly from the screen on 640x480x3 (yeah, this is a color screenshot):



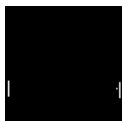
Next, the image gets resized to a 80x80x3:



Then, the color channels get reduced to grayscale:



Then, the image gets thresholded to black and white values:



Finally, the new image gets stacked with the previous 3 preprocessed images:

So the stack which originally looked like this:

TIME:

Beginning -----> Current



~~~~~ Now, the image on front, which is the oldest image gets dropped.

TIME:

Beginning -----> Current



And the new image which we just processed gets pushed to the end ^~~~~^.

This is important because we want the agent to always have the last 4 images in the correct sequence. It should not matter whether the images get pushed into the beginning or the end only if they always get inserted into the data structure in the same order. If they are not, the agent would get confused with the direction and/or magnitude the ball is traveling.

## Algorithms and Techniques

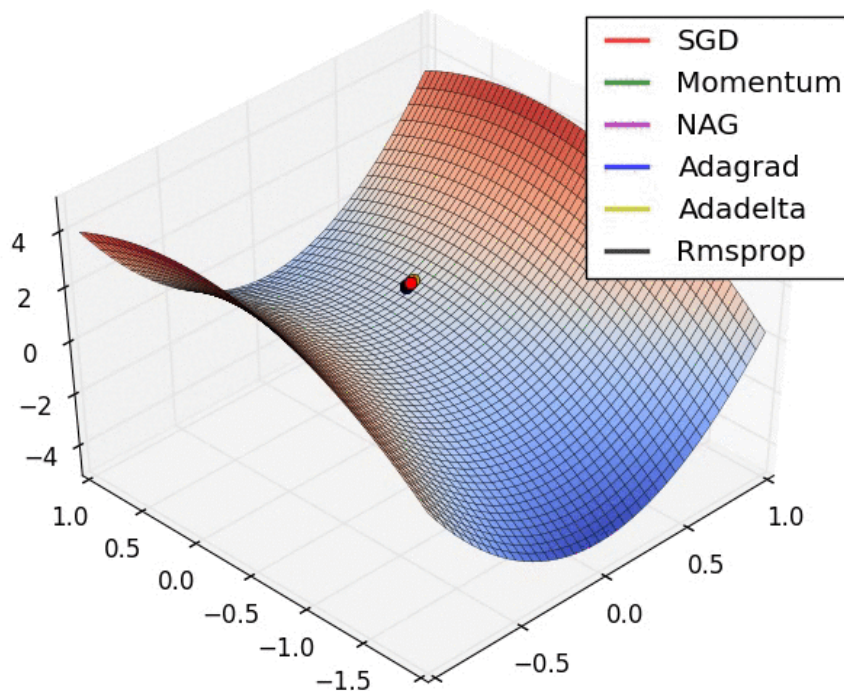
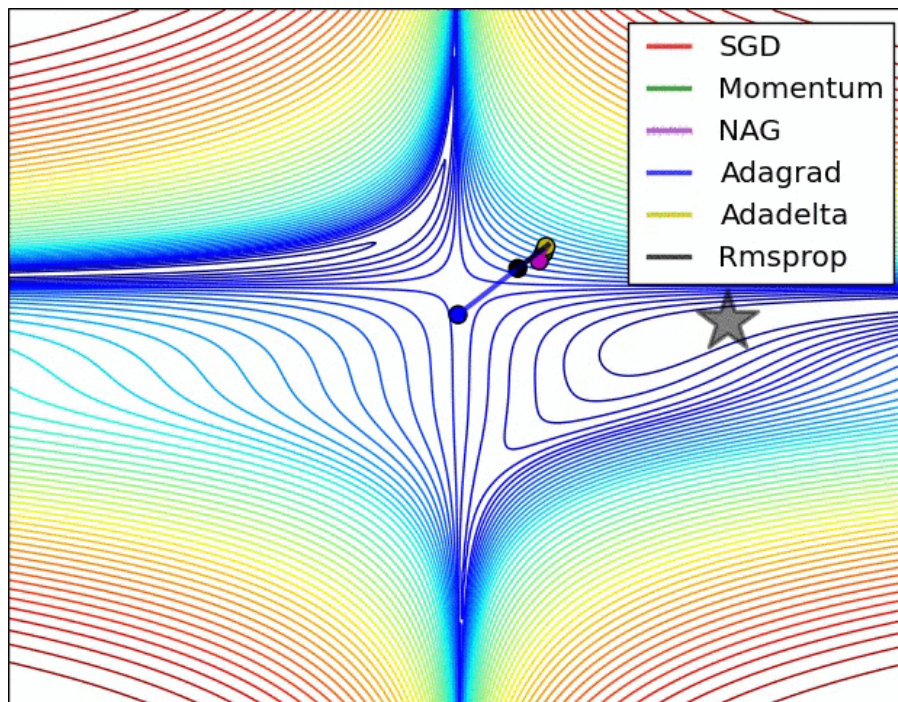
For this project we will implement a variant of the Deep Q-Learning algorithm that the Google Deep Mind team release a couple of years ago. Now, why wouldn't an algorithm like KNN work for this problem? How about Linear Regression? How about clustering. Well, it is very important to clarify that this problem is neither a Supervised nor a Unsupervised Learning problem. The problem at hand is Reinforcement Learning which is "the third" field of Machine Learning.

As I mentioned on the [README](#), Deep Reinforcement Learning is the union of two technologies. On one hand, we use Reinforcement Learning techniques to train a policy for the best action to take. In conventional Supervised Learning, we usually start with a dataset, in Reinforcement Learning the agent generates experiences in the form of state, action, reward, new state tuples, which basically tell a story. The world was on this state, I took this action, and I got this reward, and arrived to a new state of the world. So Linear regression would just not be able to work. On the other hand, we use Deep Learning which is Supervised Learning. Although this is a Reinforcement Learning problem, the state space that this work has is huge. Think about how many locations could you place the ball in, how many different directions and speeds could it be traveling to, and how many different places the paddles could be at. So, in this problem, a lot of states represent technically the same issue and those states should be treated in a similar fashion. This is where Deep Learning comes to play.

Therefore, for the input space we will implement a Deep Learning Neural Network using TensorFlow. This Deep Convolutional Network will take care of receiving the input image stack and extrapolating to find the best of 3 moves (No action, Up or Down.)

The first layer of the network will have 80 by 80 by 4 to be able to get the input of 4 images 80 pixels wide by 80 pixel high. Then, we connect this layer with a convolutional layer that reduces the input further to 8 by 8 by 4, and create a densely connected layer of 1600 connections into 512, and finally we connect these to a readout network with 512 inputs and 3 outputs.

This network cost function will be the square mean error and we will be training the network with an Adam Optimizer which is known to be an algorithm that converges faster than the generic Gradient Descent. The reason, as explained on [this paper](#), is the algorithm uses the momentum of the descent to speed up its course towards minima. Here is a gif from [Alec Radford](#) that helps explain the main differences between Standard Gradient Descent and one with Momentum.



I highly recommend you read over [this post](#) for a more in-depth comparison of the different Gradient Descent method variations, but allow me to briefly explain what you see on the pictures above. First, you can see how the GD algorithms with Momentum tend to "speed-up" when the surface has high inclination. Though, this creates some overshooting effect, it is vital to speed up the algorithm. This speed-up is due to the inclusion of momentum which standard GD doesn't have and it is main reason why the loss update shows half way the trajectory while the others had already converged.

For the Reinforcement Learning agent, we will implement a version of the Q Learning algorithm. That is at it's core, it is just the same we implemented in Project 4, but this time, the learning of the agent will have a period in which it only collects information. That is, the agent doesn't train in order to allow it to collect several samples. After approximately 50,000 steps, then we allow the agent to change the networks values. Here is a pseudo-code of this algorithm as represented on [this wonderful post](#) by [Tambet Matisen](#):

```

initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
    select an action a
        with probability  $\epsilon$  select a random action
        otherwise select  $a = \operatorname{argmax}_{a'} Q(s, a')$ 
    carry out action a
    observe reward r and new state  $s'$ 
    store experience  $\langle s, a, r, s' \rangle$  in replay memory D

    sample random transitions  $\langle ss, aa, rr, ss' \rangle$  from replay memory D
    calculate target for each minibatch transition
        if  $ss'$  is terminal state then  $tt = rr$ 
        otherwise  $tt = rr + \gamma \max_{a'} Q(ss', aa')$ 
    train the Q network using  $(tt - Q(ss, aa))^2$  as loss

     $s = s'$ 
until terminated

```

Also, we will be using an initial epsilon value of 0.6 and decaying it to about 0.01 over the course of several steps. This will allow the agent to explore random moves every time we restart training and converge to an almost optimal policy in the end. There was no specific methodology on picking these values though some common sense and intuition that we need to make the agent take enough random actions to even get a chance of hitting the ball once. And as suggested on [this David Silver's lecture](#), we selected not to decay the epsilon to zero, but instead always keep some space for very small randomness.

## Benchmark

This project doesn't have an obvious benchmark. However, we will be using human performance collected from average players to compare the agent. Additionally, the agent will be playing against the hard-coded CPU player which needless to say, it is a very good player. Want to try? It won't take even 2 minutes.

For a best of 3 games of 2 points each, run:

```
python king_pong.py
```

I bet the CPU wins.

## Methodology

There were several steps in order to make this work. The Deep Network was a challenge, and the learner took a long time to show progress. In fact, we had to source a GTX 980 and install CUDA support on our server in order to iterate over our implementation fast enough.

## Data Preprocessing

As discussed above, the data wasn't preprocessed once, but the implementation includes a preprocessing routine that does the job continuously. Again, from a single image 640x480x3 (w,h,channels) read of the screen down to a stack of 4 grayscale images of 80x80x1. Here is a code snippet of how this is done:

To get the image array from Pygame we use:

```
color_img = pygame.surfarray.array3d(pygame.display.get_surface())
```



The `get_surface()` method returns a reference to the currently set display Surface. The `array3d()` method returns a 3d array of the pixels. So at this point we have an array with height, width and channels with the values for each. One caveat we noticed is that these methods return the array on a 90 degree rotation by default, though it doesn't represent a challenge to the Deep Neural Network since for it, it's consistent to what it sees. If we were to change the rotation of the image, the Network would have to be retrained to learn the new percepts.

After reading these values into a 3d array we preprocess the image to shrink it down to 80x80x3, then collapse the 3 channels to 1, finally clipping the values to 0 or 1:

```
resized_img = cv2.resize(color_img, (80, 80)) # resize down only
greyscale_img = cv2.cvtColor(resized_img, cv2.COLOR_BGR2GRAY) # reduce from 3 to 1 channel
_, binary_img = cv2.threshold(greyscale_img, 1, 255, cv2.THRESH_BINARY) # clip the 'greyscale' to black or wh
```

The last step, we stack up the new image to the input stack which we use to train and query our Deep Network, this code shows how that gets done:

```
img_stack = np.append(binary_img, img_stack[:, :, :3], axis=2)
```

This basically appends the new frame into a subset of the current stack. Exactly, the first 3 images are kept and the new image occupies the place of the last.

In addition to all this pre processing, the scoreboard that can be seen on the version of the game for humans, and when the agent is `not in training mode` has to be removed in the training version for better training time and results. This is because the score creates some noise in the data that is not worth looking at. For example, going down to reach the ball has very little to do with the score going 3-2, we still want the agent to go for the point.

## Implementation

Initially we look for a [current Pong implementation](#) to study how the Pygame framework works. Then we look at some [other sample implementations](#). After grasping the basic Pygame concepts we took the challenge and implemented our own version. Mostly because we wanted to be able to play against the CPU player, as well as programmatically move one frame at a time at the speed the agent needed to learn. So we needed to decouple most of the code and make it functional for different aspects of our problem.

Next we learned how to grab and process images from raw pixels to something we could input into our network. The tutorials in the TensorFlow website show very good basic examples of building a Multilayer Convolutional Network, and surprisingly enough, common operations like Weight Initialization and network architectures are very generic and reusable. For this reason, we started with code from their [MNIST for Expert](#) tutorial. However, we further improved on their sample code and added more utility functions for build weight and biases and convolve relu pools with a single line of code.

At one point we hit a roadblock with using Gradient Descent for this problem. Since the MNIST tutorial referenced above uses a standard Gradient Descent optimizer, for we started to look out for solutions to similar problems, and we came with a codebase that helped us complete the deep learning portion of the agent. [Deep Learning Flappy Bird](#) has a lot of useful code and since they had solve similar problem before, we found about `AdamOptimizer` in there, which is what in the end make a big difference in seeing progress (along with the GTX 980.)

Next, we coded the learning agent separately and attempted to run different learning styles that can be read in literature. For example DynaQ, adds a dreams layer to the agent which basically it tries to generate frames from their current network state. We attempted other methods to probabilistically select frames that are important given the absolute value of the rewards the agent received. We came up with this idea as it is simple a better approach to the mini-batch implementation we currently have. We will discuss this point further later on the "Refinement" section.

As soon as we started coding, we noticed that we needed to build the software using a modular approach. That is, separate the agent, from the game, and further separate the agent from the deep learning code. Some of the samples that you can find on the web, are a single file solution that is hard to read and much harder to modify. Our project is thus separated as follows.

```
.
├── agent.py      # agent code with the learning logic. It calls the game module to receive percepts, uses th
├── Dockerfile
├── imgs/
├── king_pong.py  # game code handling frames and movements. It also as the environment by giving rewards whe
├── LICENSE
├── logs/
├── multinet.py  # deep learning network code. Here we build the network, process the images, do the actual
├── networks/
├── percepts/
├── README.md
└── report.md
```

This was a much cleaner approach and it allows to more easily improve the code as we see fit.

Overall, the implementation took several weeks, and we found immense help with previous solutions to similar problems, as well as prominent papers, documentation and tutorials on the diverse technologies. Though very difficult, it was worth the work.

## Refinement

Several refinements had to be made. First of all, the agent had to train for several days to reach a good performance. Also, having many hyper-parameters, we iteratively chose values that made sense for the problem. Clearly not the best approach, as it would have been to do `GridSearch` for parameter tuning, but the greatest amount of time had already been spent in other areas of the project. The `AdamOptimizer`, for example, when set with a learning rate too large ( $10e-4$ ) will learn sub-optimal policies, in the end, a value of  $10e-10$  was selected for the longest portion of training and then  $10e-8$  seem to further improve the agent.

The Gamma value we found most useful was 0.90, we tried higher values mostly. The problem with the discount factor being too small is that closer to 0, it would make the agent too short-sighted. So one of the policies that it would quickly learn is to go to one of the corners, either upper or lower and stay there waiting for the ball. The thing is that since the corner is closer to the upper and bottom walls, there is a greater change of the ball ending in one of those places, than somewhere near the middle. Also, since the reward system was set to give 0.1 points are given to impact the ball, and 1.0 to score a point (-1.0 to give a point), intrinsically just hitting the ball is much more valuable to the agent. So, a too low value of gamma would create an unwanted behavior on the agent.

Epsilon was set to start with a 60%. This would give the agent, while in training mode, a high chance of acting randomly yet not totally random. This in really was a convenient method to quickly see how the agent was reacting to the different hyper-parameters. For example, if we started with a 100% chance randomness and the decay was throughout 500,000 iteration, we wouldn't really be able to see how good were the actions the agent was selecting early on.

## Results

The results of this project are remarkable. Not only the agent was able to beat the benchmark but in fact the agent was able to beat the CPU player on a 100 games match-up consistently. By very little, but still it did win.

## Model Evaluation and Validation

For an observation of our results, please, enjoy the agent on a 3 points per game 10 games match up against the CPU player.

```
python agent.py -g 10 -m 3 -vv
```

This should only take about a few minutes.

Concretely, we run the above command 3 times; the agent was able to win:

- 10 of 19 games
- 8 of 18 games
- 10 of 17 games



The human player, on the other hand won:

- 0 of 3 games
- 2 of 5 games
- 3 of 5 games

The tournament against the human player was limited to 3 games because on a 10 games the performance would drop considerably. Further comparison is shown next.

## Justification

We argue the agent is a very solid solution to the pong game. Though, we weren't able to limit the CPU player to our desired 70 games on the "first to 100" tournament, the fully-trained agent was able to win 100 out of 189. Also, when playing against the CPU, the average human player gets less than 50% the games against the CPU on a 10 games match. On the other hand, the agent often wins the most amount of games.

To show numbers we ran 3 times 100 games of 3 points, the results were as follows:

- First iteration CPU won 100 to 98
- Second iteration Agent won 100 to 93
- Third iteration Agent won 100 to 94

This can be considered close victories, and they are, but this is also related to the training time. Remember, this is an agent that got his first game after the 5,000,000th training timestep. The improvement would just take some extra training time on the NVIDIA GTX 980. And it is in fact training as I write these lines.

## Conclusion

Here is a gif of the fully-trained agent playing:



On this project we explored how to train an agent play the game of Pong from raw pixels. For this we use many different cutting edge technologies. First, we created a game simulator of the game of pong. This was challenging due to the world dynamics, for example, detecting ball collisions. Then we had to use Computer Vision techniques to read the raw pixels from the screen, resize and remove channels from the percepts. Additionally, we had to implement Deep Learning to reduce the state space of the

problem into something manageable. And finally implement Reinforcement Learning to allow an agent that only knows about states, actions rewards and new states into a competitive Pong player.

Along the way we found many challenges. The most difficult part of the project is to use all of these technologies to work well. To find a solution to this problem we designed a modular implementation on the code, in which the game simulator is separate from the agent and from the neural network code along with the computer vision utility functions. This not only made the project possible, but also the code reusable, as it would be much simpler to use the same agent code to solve other games. Also, it would be simple to remove the TensorFlow dependencies and implement Deep Learning using something else like Theano, or Caffe, Torch, etc.

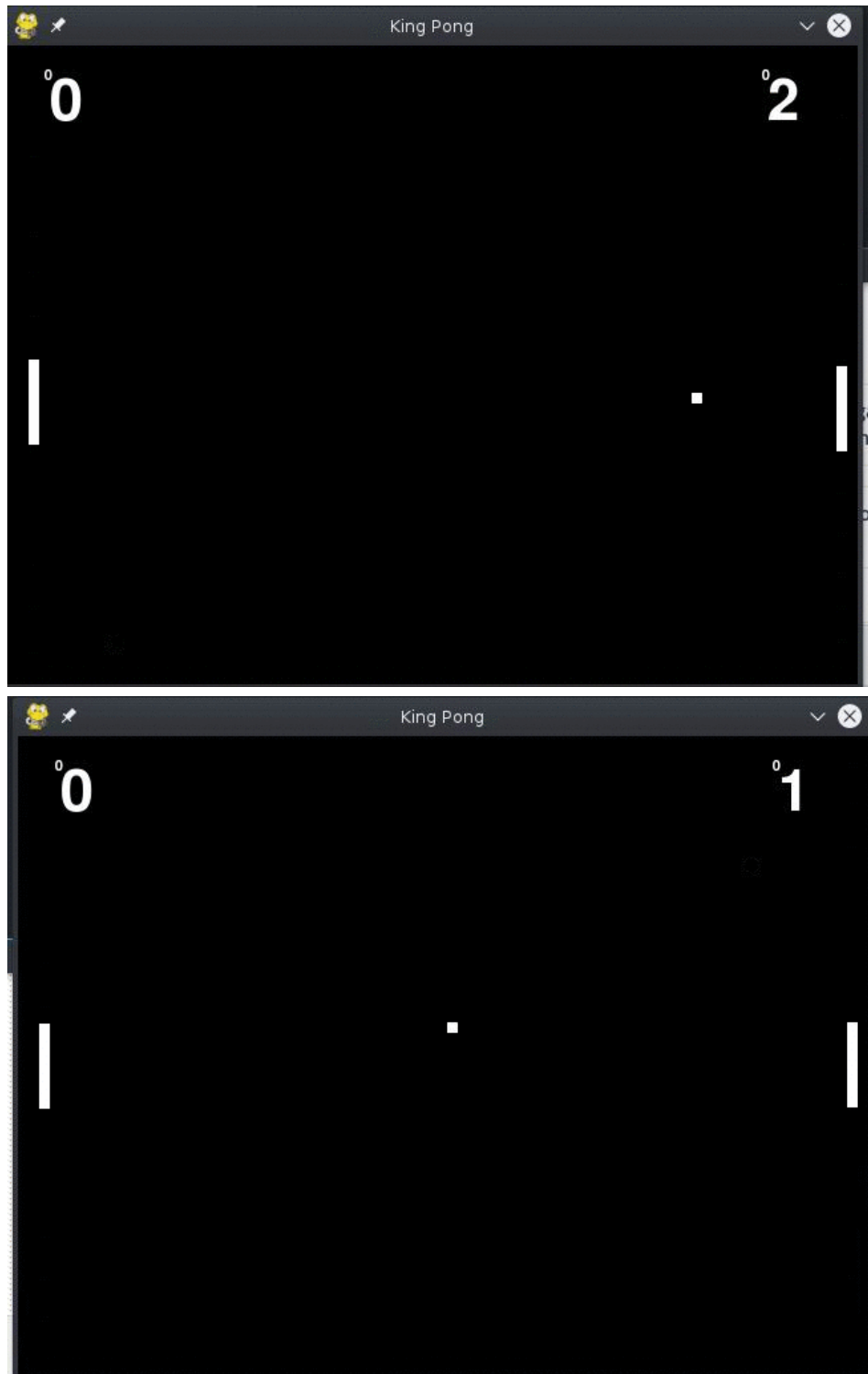
This was a very interesting project we got into. Not only we applied a Machine Learning technology, but in fact 2 of the most prominent technologies as of 2016. We proved how hard it is to do Deep Reinforcement Learning, but also how satisfactory it is to get great results. We would be doing more of these kinds of work in the near future.

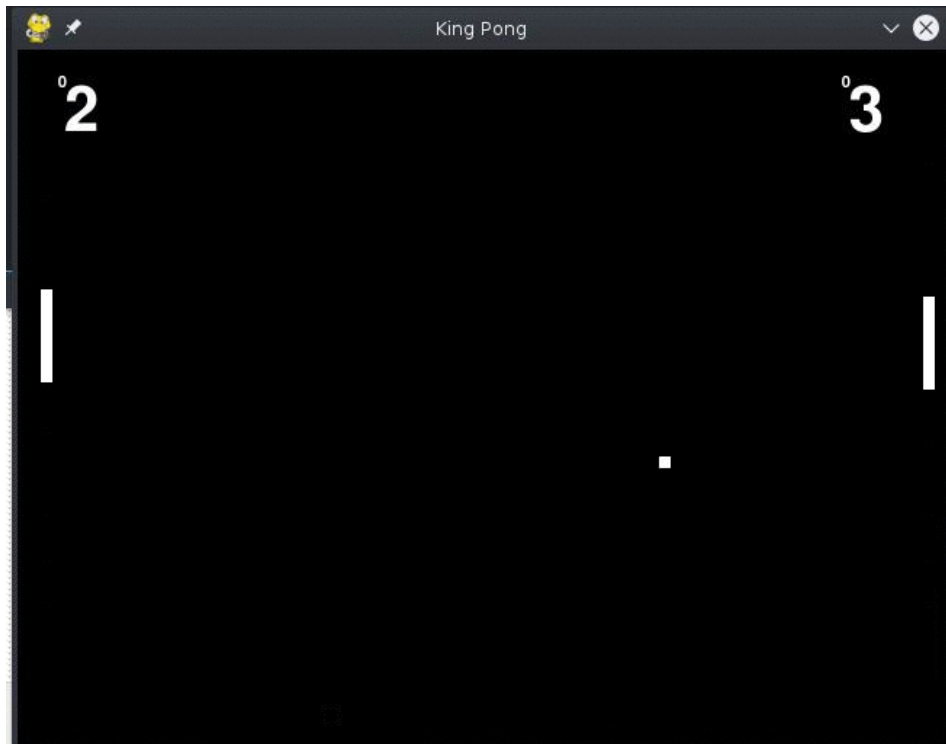
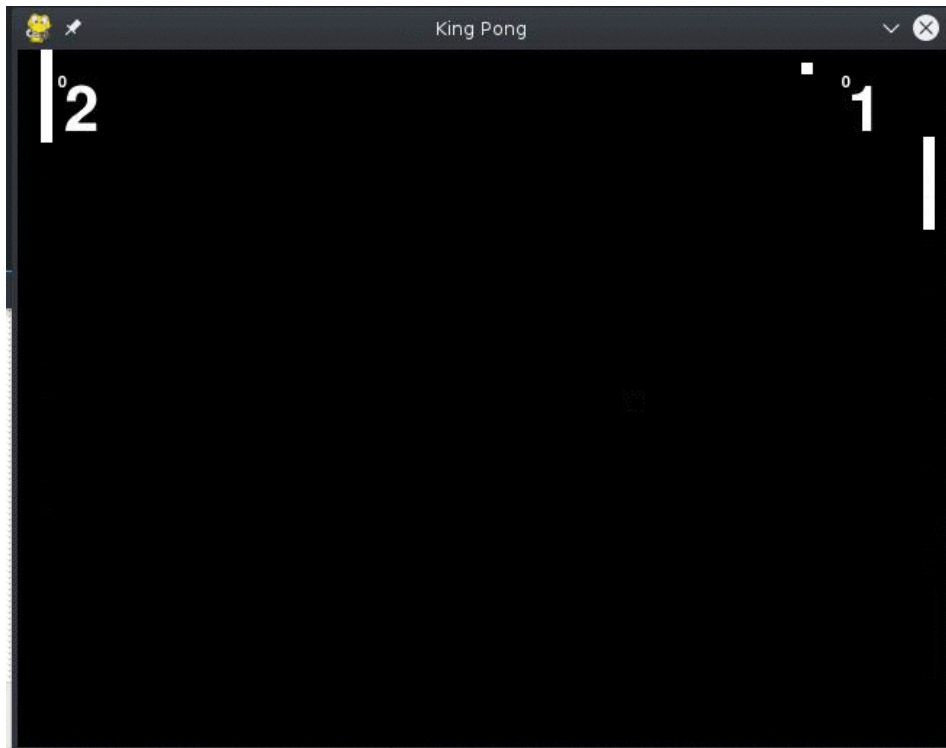
## Reflection

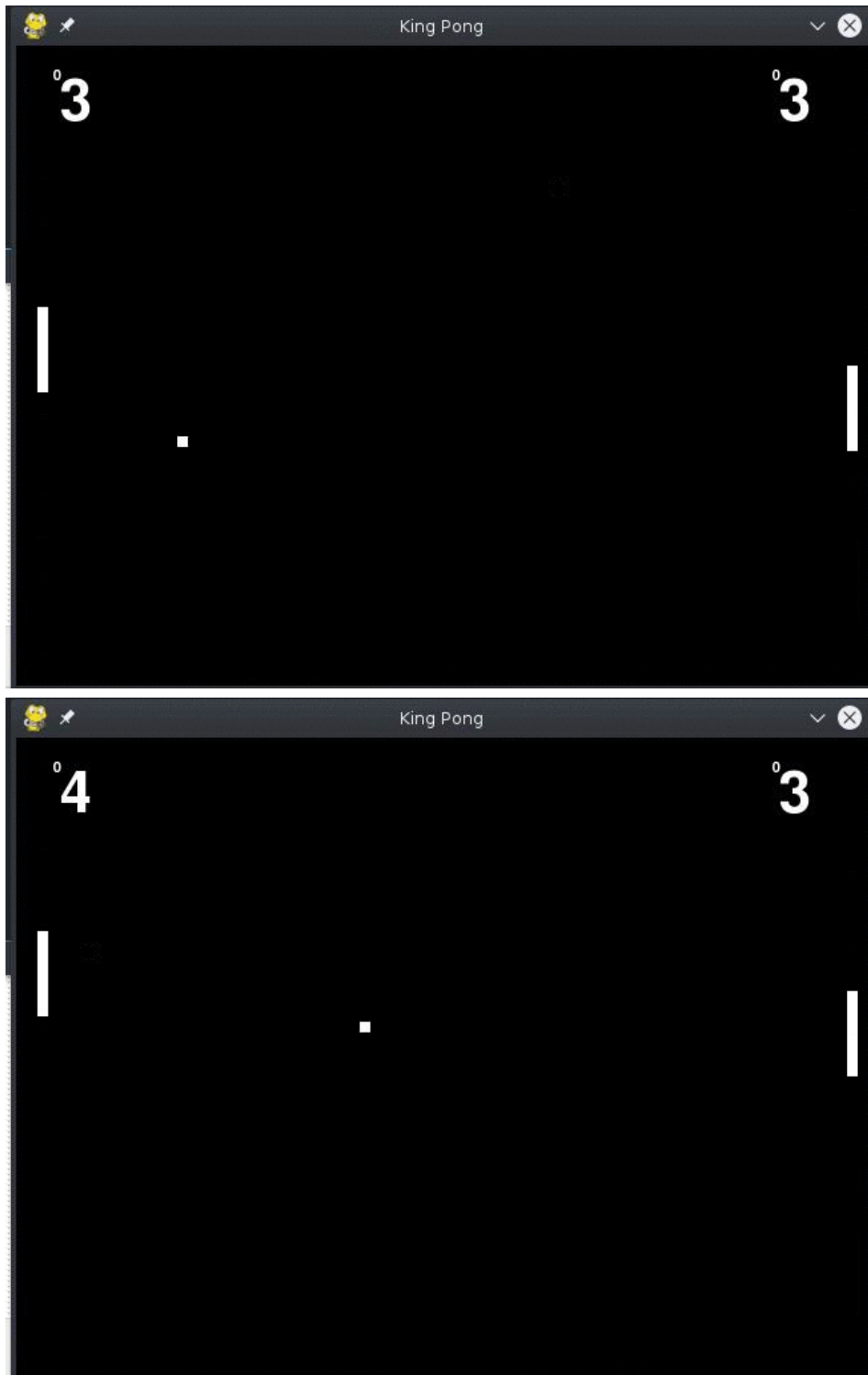
Since we used cutting edge technologies, some of the steps we had to take in order to make this project work were completely in the dark. We had very little guidance that if we had picked some other technology, it would had been much easier to follow examples. For example, deep learning is a very new technology, with the emerging tools such as TensorFlow there is enough material in how to process images, but little in how to process a series of images as to model the motion of the ball. This challenge was resolved when reading [very useful posts](#) about deep reinforcement learning, but the practical examples were not the norm.

## Improvement

Short GIFs of the moments when the agent loses:







Although completely satisfied with the results, the agent can definitely be improved. Here are some of the weak spots of the agent:

- It doesn't have a large visibility of the ball trajectory when updating  $q$  values. For example, the agent uses only 4 frames to update  $q$  values. The problem with this is that if it had more frames, the values would be able to propagate further. Imagine you being able to only see the last 4 frames of the ball, your anticipation to the plays are much more delayed. The same happens with the agent. Though, this is expensive and it might not improve performance greatly. Though, more explanation is required.
- Discount factor could be improved. As we discussed before, some of the hyper parameters were selected on a trial and error way. One good improvement would be to use some randomized algorithms and pick the best parameters.

We can think of some improvements the agent would benefit from. For example, currently we only sample 60 images from the 50,000 images in the memory queue. This is a rather small portion of what could be sampled, even if performed on every timestep.

For a next iteration of this agent, we will be implementing a more complex learning routine that would take into account those features and probabilistically select frames that are to be of meaning to the agent. In specific we think about something along these lines:

- From the memory batch create a probabilities array for selecting a frame.
  - The absolute value of the reward would be a good metric. That is our rewards would give weights of  $(-1, 1, 0.1) \rightarrow (1, 1, 0.1)$
  - Then we normalize the array by dividing each value by the sum.
- Select 100 memories with replacement (Just as humans potentially replay same dreams over and over.)
- Add padding to each memory.
  - Randomly select a beginning frame ~50 frames earlier than the selected frame.
  - Do the same for the last frame; ~50 frames after the selected.
- Train using these frames sequences.

We believe that this is a much intuitive way of selecting frames to train on. As the current algorithm doesn't prioritize frames and it could take longer to actually select a frame that changes the qvalues on the network due to not having any reward at all. The current implementation would work best if the reward system was much more noisy. For example, lets say we want to reduce the jitter on the agent, so we add a -0.001 reward just giving for moving. Then the random selection of the frames would convey meaning to the agent most of the time. Still though, prioritizing frames seems like a great idea and very straight-forward to implement.

Another improvement that could be easily implemented is allow the agent to learn from a human playing. That would be another simple addition, we just would have to bypass the actions that the agent is taking and use the key presses the user sends. Perhaps, even just enabling the actions to be overridden by the human player at any time. We just would have to make sure we use the actions sent by the user not only for controlling the game, but also for storing it on the memory queue.

## References

---

Other references are spread out throughout the project, but these were particularly useful.

- [Deep Learning Flappy](#)
- [Deep Learning Tutorial](#)
- [Pong PyGame](#)
- [Udacity's Reinforcement Learning Course](#)

