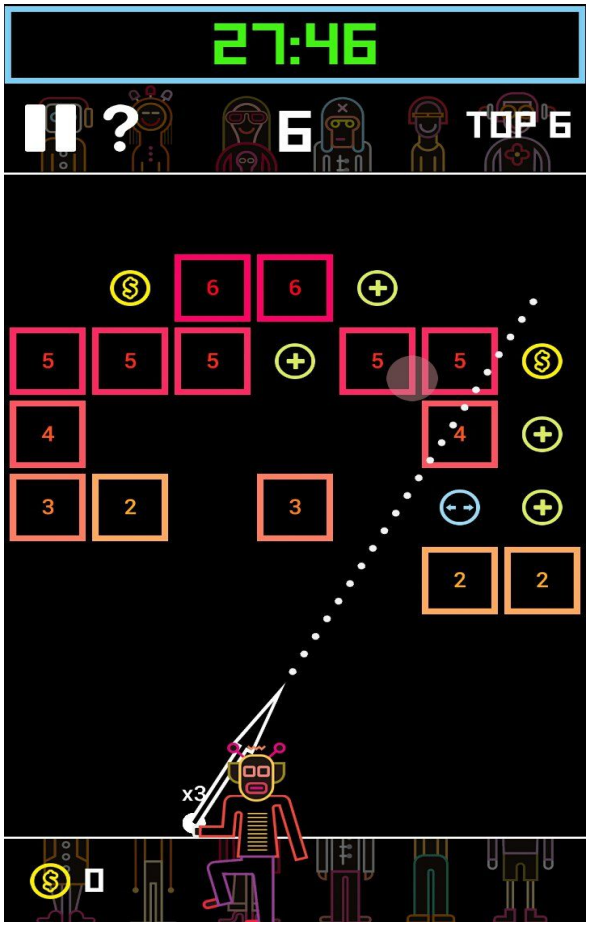
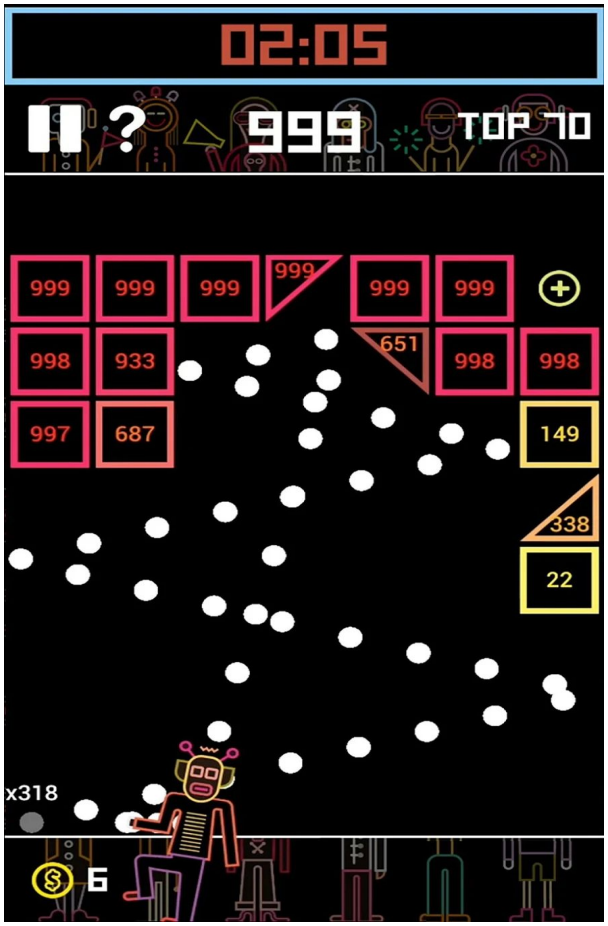
# Mastering BBTAN with Deep Reinforcement Learning

Brandon Tan (tjiansin), Irvin Lim (ilim5), Xiangyu Li (xli148), Chong Wang (cwang147)

## Introduction

BBTAN is a mobile game that gained popularity in late 2018. As of 10 November 2019 it has garnered over 10,000,000 mobile installs worldwide. BBTAN provides a refreshing twist and is reminiscent of brick breaker games that many of us enjoyed during our childhood. For this project, we attempted to create a bot (built on top of deep reinforcement learning techniques) that can master the game and consistently produce highscores.



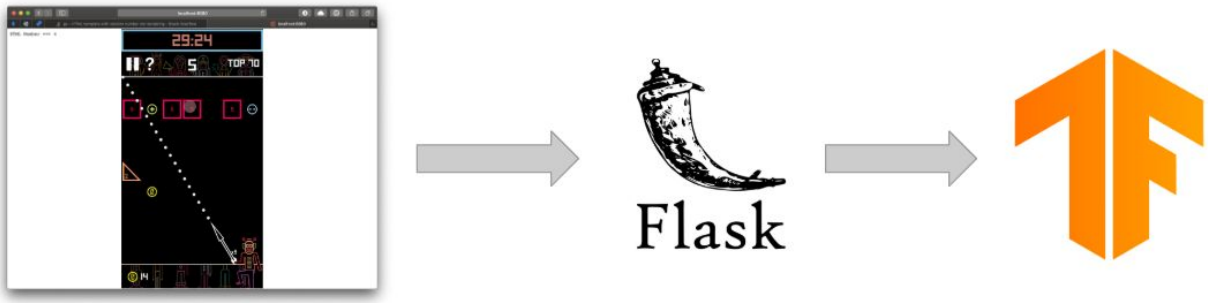



## Setting up the game for Deep RL

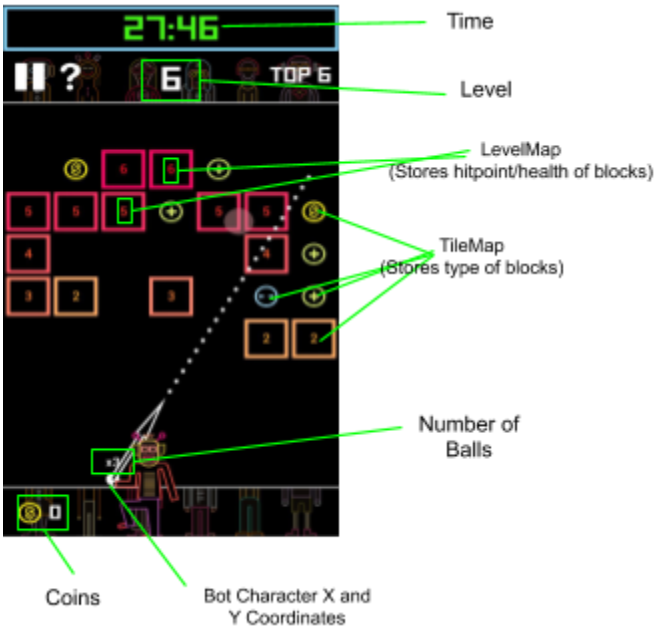
As BBTAN is a mobile only game, there were initial concerns of having to rely on emulators to programatically interact with the game. Emulators are notorious for being CPU/GPU hungry and performance is usually far from ideal. Our worries are compounded by the realisation that we would also need to speed up the game by factors of tens or hundreds to realistically train our model within the time that we have. Unfortunately, the various public emulators that we looked at does not support a speed up feature nor is it trivial to implement one. Moreover, even if they do support speeding up the game, this also means that emulator would need even more CPU/GPU which would likely greatly exceed the specifications of the machines that we have access to.

An alternative is hence required and a quick search on Google revealed a few open source clones of the game built in various game engines for multiple different platforms. Notably, there were 2 prime contenders for the project; one built using Javascript, the other using Allegro Gaming Library (Windows only) in C. For compatibility reasons, the Javascript clone was eventually chosen.

However, using a Javascript clone meant that we could not directly access or interact with the game like we would for a native game since a browser is required to run it. In order to overcome this limitation, we modified the game source to hook onto specific events (e.g. pending action event or game over event) such that a http request is sent whenever these events are fired. Consequently, we deployed a local web server (built using Flask framework) to capture these requests. Subsequently, the web server will handle these requests by calling our model and output from our model will be used as the response. This setup essentially allows us to use the web server as a bridge between our game and our model.




When a game event is fired, the http request is sent along with a comprehensive list of all the current game state variables.

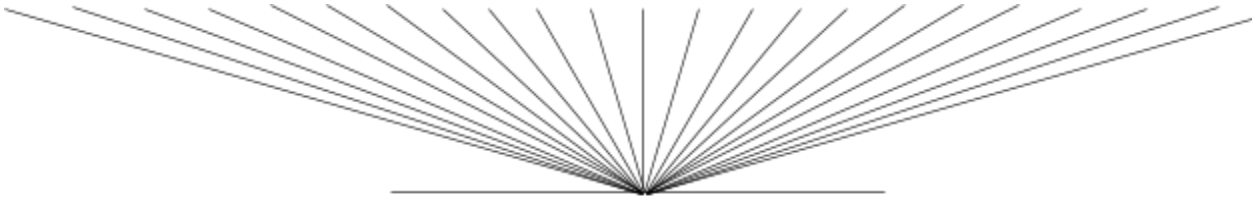



```
{
      level: game.level,
      balls: game.totalBalls,
      coin: game.coin,
      bot_x: game.bbtanGameBot.x,
      bot_y: game.bbtanGameBot.y,
      tileMap: game.tileMap,
      levelMap: game.levelMap,
      gameTime: game.gameTime,
      gameStatus: game.gameStatus
 }
```

While having verbose game state data is definitely convenient and helpful, many of these variables are not essential when it comes to helping the model make a decision. More specifically, our model is used to predict the best angle given the current game state. Therefore, we deemed certain variables like coin, level, bot_y, gameTime, gameStatus as unnecessary.

Hence, our state space for our model is the following flattened array:
[normalized_balls normalized_bot_x normalized_tileMap levelMap]


while our action space is defined as an integer from 0-24, representing 25 evenly spaced angles

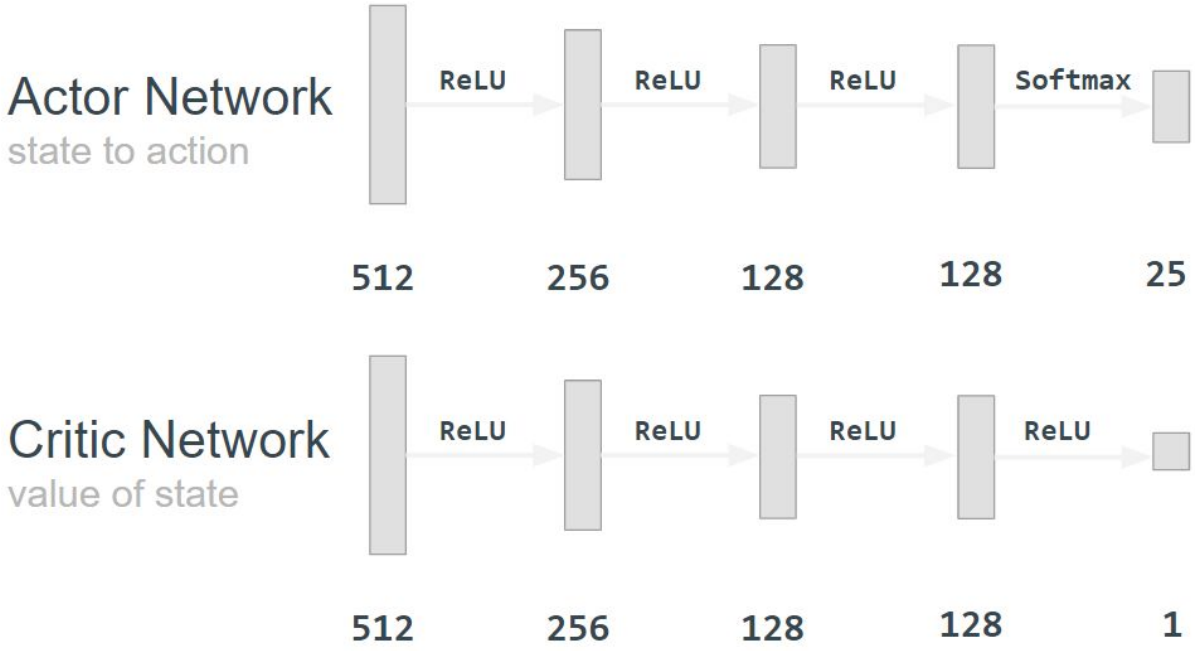


*Not drawn to scale

# Our Model

Various models were considered during initial discussions. Some models that were brought up include DQN, A2C and A3C. Each of these models were researched and we settled on A2C as our model. This is mainly due to our familiarity of A2C as well as the results that A2C seemingly is able to achieve. Moreover, certain models like A3C, which requires parallelization of the game environment, is not something our game code base could trivially support and hence was eliminated from consideration.

For our project, the following architecture is used:




- All layers are initialized from a normal distribution with a mean of 0 and standard deviation of 0.02
- Adam optimizer with learning rate of 0.0001
- $Loss_{actor} = -\sum_{0}^{n} log(P_{a_i})(D_i - V(s_i))$

  $Loss_{critic} = \sum_{0}^{n}(D_i - V(s_i))^2$

  $Loss_{total} = L_{action} + 0.5 * L_{critic}$

  where $P_{a_i}$ is the probability of the action at index $i$,
  $D_i$ is the discounted reward of state at index $i$,
  $V(s_i)$ is the value of state at index $i$

# Overcoming Challenges

Over the span of time working on the project, 2 main challenges stood in our way.

## Slow Training

Excited as we were to start training our model when we first have the architectures set up, it quickly became very apparent that each epoch or a single game play was taking excruciatingly long. Each gameplay was taking anywhere from 10 seconds to 2 minutes to complete. Within 24 hours, barely over a thousand gameplays could be completed. This was a really big problem for a couple of reasons. Firstly, reinforcement learning is notoriously known for needing a lot of training epochs. Secondly, CPU resources of the host machine is not efficiently utilized. Many CPU cycles are wasted idling as most of the time taken in a gameplay is attributed to waiting for the balls to complete their animation sequence. Lastly, by looking at the training epochs in the original A2C/A3C paper, a simple game like Atari Breakout (Breakout's game concept is similar to BBTAN) requires tens of millions of training frames corresponding to tens of thousands of games to be proficient[1]. In comparison, BBTAN is much more complex with a much larger action and state space. This means that it is highly likely that our model would require a lot more training to be proficient at BBTAN.

Hence, we set out to do many experiments and research in an attempt to speed up the game. At one point we even considered building a native version of the game ourselves. Fortunately, some of our experiments paid off and we progressively increased our game's frames-per-second (FPS).

The first major boost to FPS came from switching out the requestAnimationFrame function which the game used with the setTimeout function. The requestAnimationFrame function is a method that signals the browser to re-render a page or update an animation[2]. The problem with it is that the frequency of this re-render operation is fixed at 60 FPS and can only be controlled solely by the browser. Hence, the switch to setTimeout. The setTimeout function allows us to specify an arbitrary interval (4ms lower bound) between re-renders and thus allows us to reduce the interval delay, increasing FPS to the range of 200 to 300.

Despite the 5x FPS improvement, training speed is still objectively too slow. In order to further boost FPS, we implemented batch rendering. Instead of rendering a single frame and then waiting for the interval to expire, multiple frames are rendered within the same interval. In our project, we used a batch size of 24  which seem to be optimal for the hardware that we are

---

[1] "Asynchronous Methods for Deep Reinforcement ... - arXiv." 17 Jun. 2016, https://arxiv.org/pdf/1602.01783. Accessed 14 Dec. 2019.
[2] "Window.requestAnimationFrame() - Web APIs | MDN." 12 Oct. 2019, https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame. Accessed 14 Dec. 2019.

using. With batch rendering, we managed to achieve a FPS range of 2200 to 3000, a further 10x improvement and a 500x improvement over the initial 60 FPS.

### Not converging

With the FPS now in acceptable numbers, we began model training. The first training session ran for approximately 10 hours, completing slightly less than 10,000 gameplays. We were glad to be able to complete that many gameplays in a reasonable amount of time but when inspecting the model results (e.g loss charts, reward charts, etc.), it was far from ideal. The losses and reward values were fluctuating wildly, no signs of any learning could be observed. There was no downward/upward trend nor is there any trend. Actions predicted by the model were pretty much random. Nothing was seemingly converging.

This sparked a flurry of more research, questioning of initial assumptions and intense scrutiny of the code we've implemented. The accumulation of all those trials and errors led to the implementation of 5 techniques/design decisions which significantly improved our results.

Firstly, normalization of game state. In our first training session, the input to our model were the raw values of the game state. This includes the x coordinates of our character, number of balls, the types of block represented using 13 different integers and the hitpoint/health of each block. According to some literature we've found in our research, due to the different ranges of values for each of these components, gradients showed a tendency to oscillate back and forth for a long time before convergence[3]. In addition, categorical variables like our block types although already integer encoded artificially presented a natural ordering relationship between the types which should not exist[4]. Hence, to combat these issues, each component is normalized. . The number of balls and block health were divided by the level (number of balls and health scale with level). Block types on the other hand, being a categorical variable, was converted into one-hot vectors. Similarly, x coordinates of the character were also normalized by converting it into a one hot vector (game is fixed width).

Secondly, reduction of the action space. The initial action space had a size of 350*724. Each action corresponds to a pixel in the game. Each pixel gives rise to a different shooting angle when clicked on. Such a large action space contributes to the convergence issue and our team decided to reduce it by fixing the y coordinates, leaving only 350 possible actions. Subsequently, we felt 350 actions is still too large and further reduced it by limiting it to only 25 action which corresponds to 25 evenly spaced angles/x coordinates.

---

[3] "Why Data Normalization is necessary for Machine Learning ...." 7 Oct. 2018, https://medium.com/@urvashilluniya/why-data-normalization-is-necessary-for-machine-learning-models-681b65a05029. Accessed 14 Dec. 2019.

[4] "Why One-Hot Encode Data in Machine Learning?." 28 Jul. 2017, https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/. Accessed 14 Dec. 2019.

Thirdly, batch gradient updates. To help combat the wild fluctuations that we've observed, a switch between stochastic gradient updates and batch gradient updates was made. While SGD allows models to converge faster, it also introduces more noise per update[5]. On the flip side, batch gradient updates takes a longer time to converge but provides probabilistically more optimal and less noisy steps per update. Hence, a delicate balance is required and for our project we utilized a batch size of 32. In other words, gradient updates are only done once every 32 gameplay by averaging the gradients accumulated across them.

Fourthly, building a deeper model. With the normalization of the input game state, the state space got significantly larger. From a state space of size 128, it increased to size 1234. As such, the initial model architecture that we have, 3 layers (including input and output layers) each for actor and critic networks with 128, 128, num_actions/1 hidden weights respectively, was likely too small and too shallow for our normalized state[6]. However, we are also weary of the risks involved in having excessive hidden weights and eventually settled with 5 layers (including input and output layers) each for actor and critic networks with 512,256,128,128, num_actions/1 hidden weights respectively

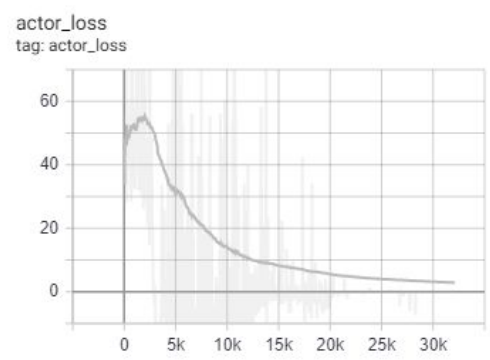
Lastly, reward shaping. To give a better chance for the model to converge and perform decently, we partake in reward shaping by injecting some of our basic knowledge of the game into the reward function to encourage certain behaviours. The base reward for each level survived is 2. However, disincentives were handed out for each remaining block in the game state and disincentives are scaled proportionally to the position of the blocks. Blocks in a lower position are given a higher penalty while blocks in a higher position are given a lower penalty. Additionally, we also give out a small incentive for each empty space in the game states.

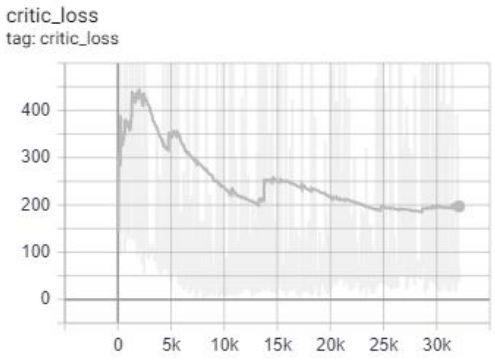
$$\text{Reward} = 2 - \sum_{0}^{\textit{num remaining blocks}} 0.01*\text{block\_position} + \sum_{0}^{\textit{num empty spaces}} 0.01$$

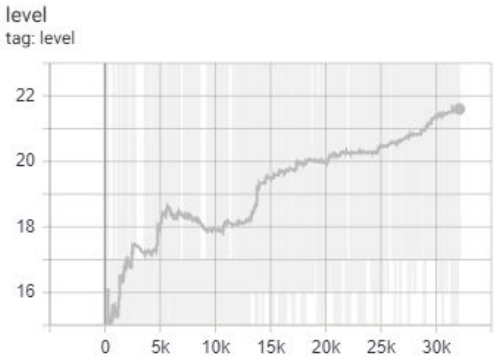
[5] "An overview of gradient descent optimization algorithms." 15 Jun. 2017, https://arxiv.org/pdf/1609.04747. Accessed 14 Dec. 2019.
[6] "The Number of Hidden Layers | Heaton Research." 1 Jun. 2017, https://www.heatonresearch.com/2017/06/01/hidden-layers.html. Accessed 14 Dec. 2019.

# Results

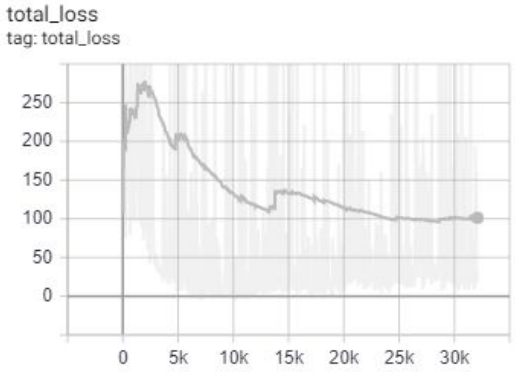
The combination of all these techniques, tricks and hacks turned out favourably. The described challenges and issues were largely mitigated. Below are the results after 30,000 gameplays.



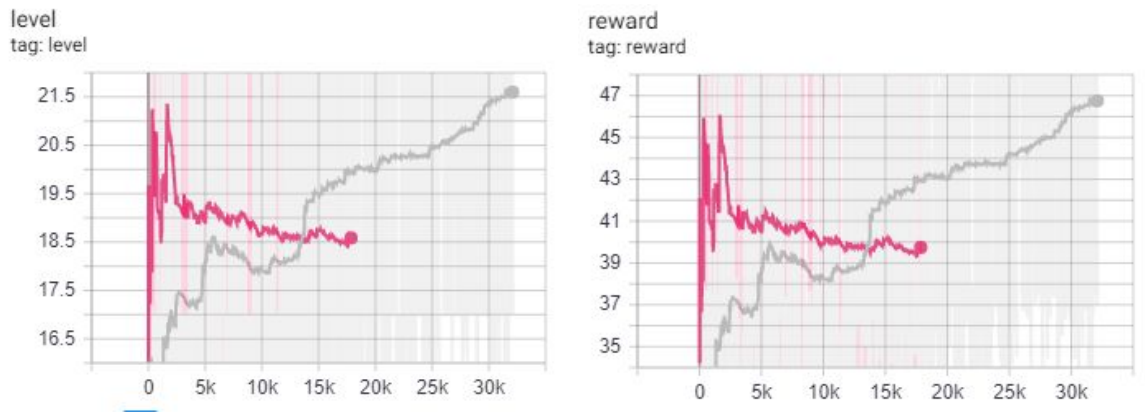








As seen from the charts, our actor and critic loss progressively get lesser while our level and reward charts produces a really nice upward trend. To give more perspective, we benchmarked our model we a random agent.




On average our model outperforms the random agent (pink line). As can be observed from the charts, the random agent averages around level 18 to 19 while our model steadily improves and averages around level 21 to 22 (after 30,000 gameplays).

# Reflection

Our team is happy that the project concluded with decent success but in hindsight there are still areas in need of improvements.

### Critic Loss Stagnation

As seen from the result charts above, the critic loss stagnants at a certain point. We believe that in order to further improve the performance of our model, we have to work on reducing the critic loss as much as possible. Currently, it is not very clear as to why the critic loss is stagnating, more research and experimentation is required.

### Native Game Code Base

In addition, even with the FPS boost described above, the browser is still not able to fully utilize the resources that the host machine is able to offer. Going forward, it might be a wise decision to look for alternative game code bases, preferably native ones such that the host machine can be more efficiently utilized, speeding up training speeds. Moreover, native code bases would also likely be able to support parallelization of the game which opens up the potential to use other models like A3C.

### Final Thoughts

Overall the project displayed the importance of thoroughly understanding the internals of deep learning models while also dispelling the myth that deep learning is very much like a black box that comes plug-and-play. From hyperparameter tuning to architectural decisions, a lot of work is done in the background to fit the environment or problem that the model is learning to solve.

Our team definitely learned a lot throughout the semester and we will continue to add more layers when life throws us more lemons[7].

## Source Code

https://github.com/BrandonTJS/CSCI2470-BBTan

[7] "Assignments | CS147 - Deep Learning | Brown ... - Brown CS." http://cs.brown.edu/courses/csci1470/assignments.html. Accessed 14 Dec. 2019.