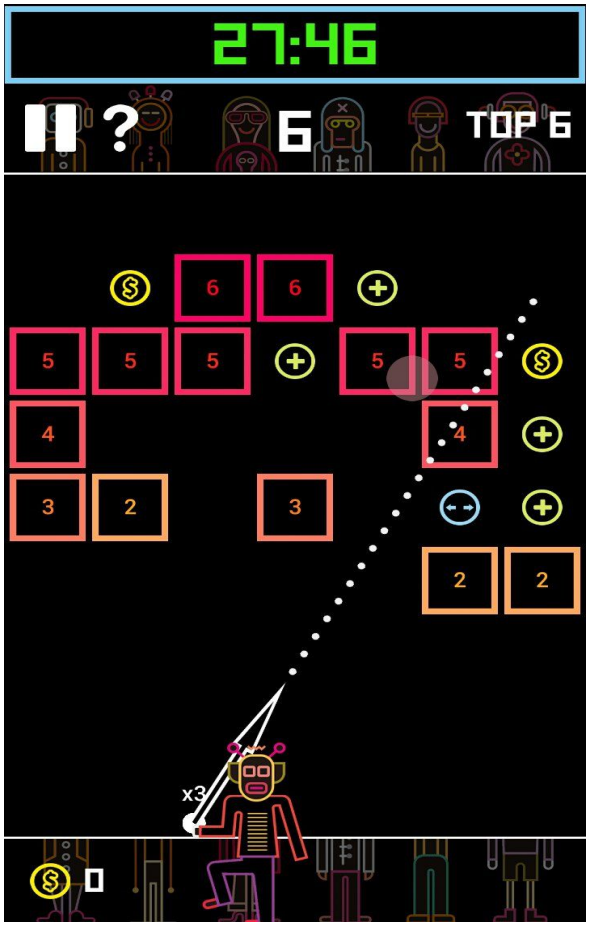
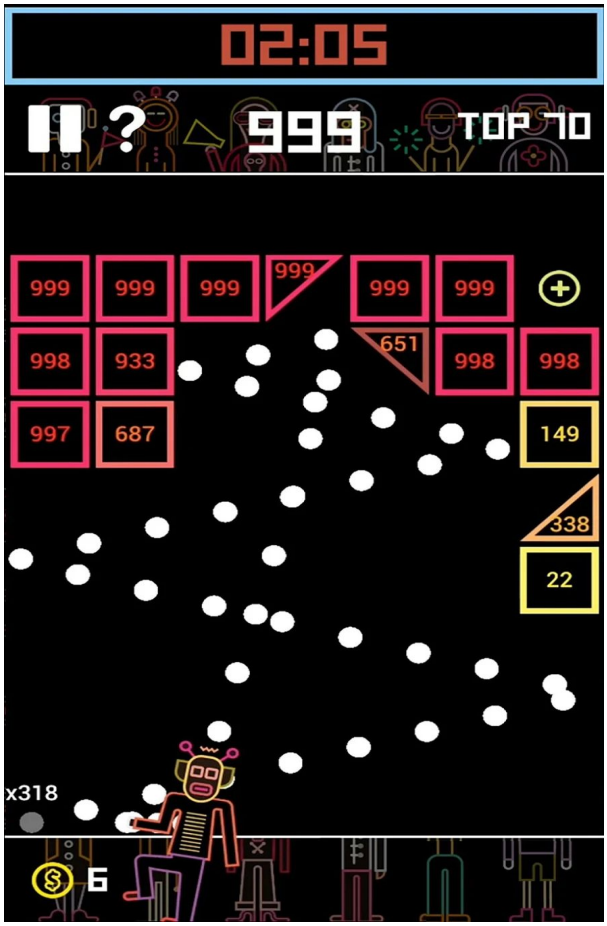
Mastering BBTAN with Deep Reinforcement Learning

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 train a reinforcement learning model in order to create a bot 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 web search 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. This also allowed us to inspect and retrofit the game code to make it more amenable for faster training.

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.

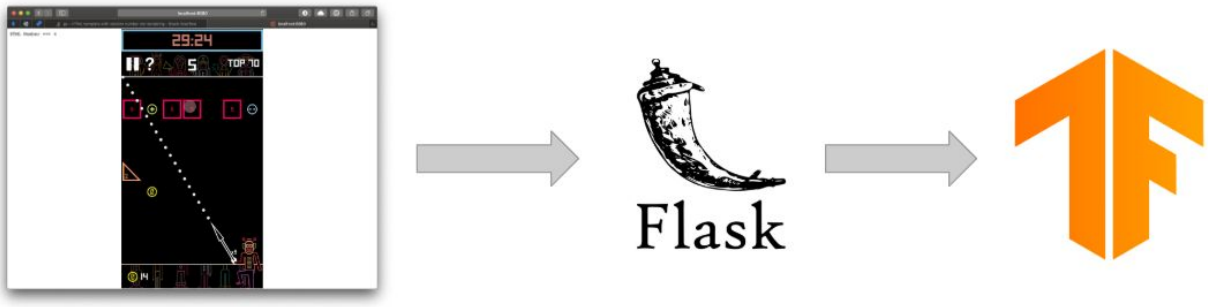
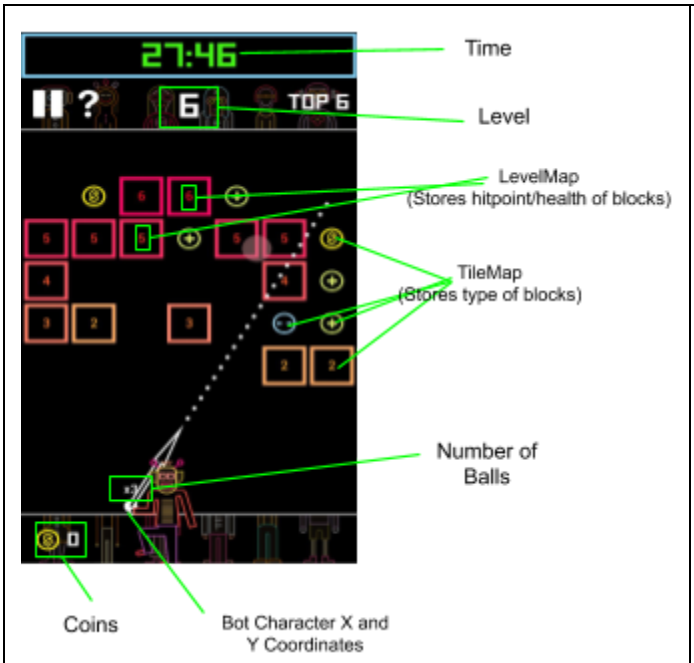


Figure 1: High-level architecture of training environment

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
 }
```

Figure 2: Outline of all variables in the game's state space.

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, the bot's y-coordinates (it remains constant), remaining game time (we disabled the time limit) and game status as unnecessary states.

Hence, our state space for our model is the following flattened array:

Figure 3: State space in an array form.

Our action space is the set of angles to shoot the balls at. Since angles are a continuous variable, we discretized the angles so that we would have a well-defined action space. We defined our action space as an integer from 0-24, representing 25 evenly spaced angles as follows:

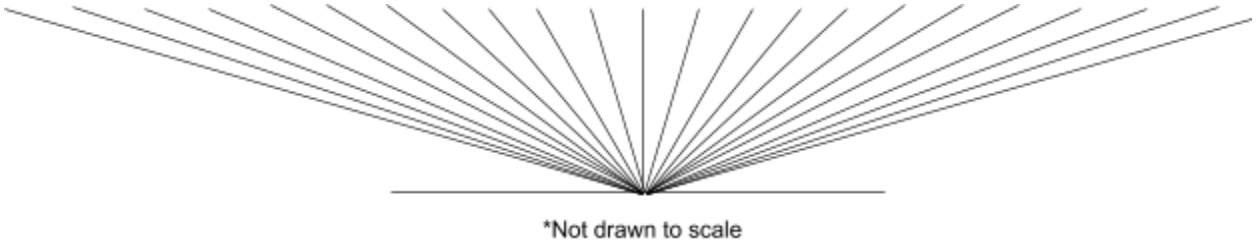


Figure 4: Ball trajectory angles, corresponding to the action space.

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:

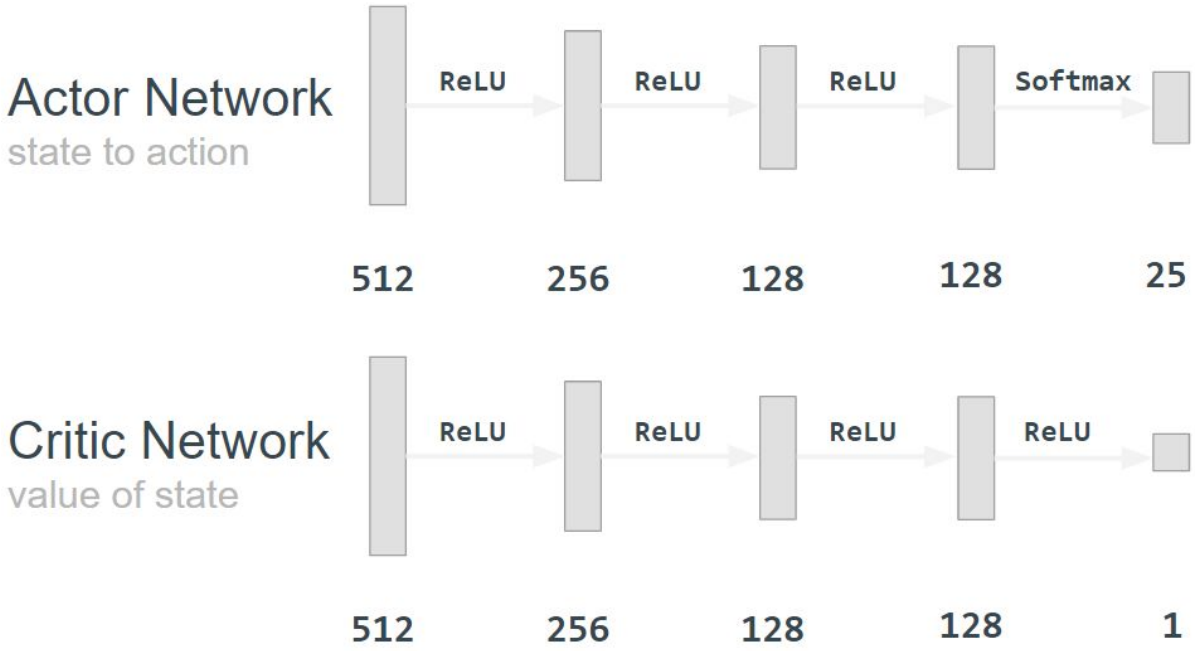


Figure 5: Model architecture used in the A2C network.

- 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
- Hardware configuration used for training:
  - CPU: Intel i7-9750H (12M Cache, 2.60GHz, up to 4.50GHz)
  - GPU: NVIDIA RTX 2060
  - RAM: 16GB

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, as many CPU cycles are wasted idling waiting for the balls to complete their simulation in the game. 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 training time. Since the game runs on JavaScript but is rendered in the browser, we initially thought of simulating the game in Node.js (a server-side JavaScript runtime) instead of the browser, since there would be much overhead from updating the Document Object Model (DOM) in the browser. On the other hand, if we reduced the amount of time taken to simulate an action, we could essentially increase the number of episodes trained per unit time. We chose to adopt the latter approach, as we could essentially visualize the training process in the browser.

---

[1] "Asynchronous Methods for Deep Reinforcement ... - arXiv." 17 Jun. 2016, https://arxiv.org/pdf/1602.01783. Accessed 14 Dec. 2019.

One possible method was to increase the position delta of objects in the simulation. Like most physics simulations and game engines, movement of objects are modeled by an (x, y) vector for its position as well as a $(dx,\ dy)$ vector for its velocity, where the position of any object at time t is given by

$$(x_t, y_t) = (x_{t-1} + dx, y_{t-1} + dy).$$

If we scaled the values for $(dx,\ dy)$ by a larger constant velocity factor, a ball object would move over a longer distance per frame, which could potentially reduce the time taken for the balls to complete its entire path. However, if we increased this value too liberally, the collision detection mechanisms used in the game engine would fail to work as well. For example, a ball could be moving towards a block, but in the next frame it could "phase" through the block if the velocity is too high, and thus the block will not be broken. As such, we chose not to adjust this value in the game, since it greatly compromises the accuracy of the real game.

Instead, we decided to increase the frame rate (or FPS) of the game, which is essentially the number of simulated state updates performed per second. Most game engines constrain the framerate to 30 or 60fps, because the human eye can only perceive distinct frames between 25-30fps, and 60fps is the standard for most displays today. Because the game uses the `Window.requestAnimationFrame()`[2] method to throttle and run frame animations at a constant rate, by scheduling a frame every 1/60th of a second, or an interval of 16ms per frame. There was no native way to change the framerate used by the browser, and thus we decided to modify the game code and design our own frame scheduler to prevent throttling of the frame rate.

We first tried to run the frame updates endlessly in an infinite `while` loop, but quickly realised that because the JavaScript runtime is single-threaded, this would block all asynchronous operations such as the sending and receiving of XMLHttpRequests, which is used by the game to communicate with the bridge (and hence the model). This method also quickly crashes the browser.

Instead, we used the `setTimeout()` function which allows us to perform a frame update with a 4ms minimum delay, a lower bound specified by the HTML specification[3]. Specifically, we could schedule the next invocation of the frame update only after 4ms later. This allowed us to improve the frame rate from a previous value 60fps, to a maximum of 240fps.

Despite the 4x frame rate improvement, training speed is still objectively too slow. In order to further boost FPS, we implemented batch rendering in each `setTimeout()` invocation. Instead

---

[2] "Window.requestAnimationFrame() - Web APIs | MDN." 12 Oct. 2019, https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame. Accessed 14 Dec. 2019.

[3] "WindowOrWorkerGlobalScope.setTimeout() - Web APIs | MDN." 22 Aug. 2019, https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout. Accessed 14 Dec. 2019.

of rendering a single frame and then waiting 4ms for the timeout to expire, a batch of frames are rendered and updated consecutively before waiting for the 4ms timeout. This is essentially a hybrid between the infinite loop method mentioned previously and `setTimeout()`, in order to allow other asynchronous events to fire but yet drastically improve the framerate. In our project, we found that a batch size of 24 was optimal for the hardware that we used for training our model, in order to finally make the simulation CPU-bound to maximize the hardware.

We also implemented a frame rate counter that is displayed within the browser on each frame redraw, which allowed us to benchmark and monitor the frame rate during training. This was implemented using a circular buffer that stores the latest 50 timestamps when each frame was rendered, and returns the reciprocal of the average time taken between each frame. With the asynchronous batch rendering method described above, we managed to achieve a final frame rate between 2200-3000fps, a total of a 500x improvement over the initial frame rate of 60fps, which allowed us to run 10,000 episodes in around 10 hours of training.

Non-Convergence

With the framerate now in acceptable numbers we began model training, but found that the model results (e.g loss charts, reward charts, etc.) were far from ideal. The losses and reward values were fluctuating wildly, and 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, and nothing was seemingly converging.

We used 5 different strategies which helped improve our results significantly.

Firstly, we normalized the game state. Initially, the inputs 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. We found that 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[4]. 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[5]. Hence, to combat these issues, each component is normalized by an appropriate mechanism. The number of balls and block health were divided by the level (number of balls and health scale with level). Additionally, since block types were categorical integer variables, they were encoded into one-hot vectors instead. Similarly, the x-coordinates of the character was also normalized by converting them into a one hot vector (since the game is fixed in width).

---

[4] "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.

[5] "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.

Secondly, we reduced the action space significantly. The initial action space had a size of 350*724, where each action corresponds to a pixel on the game screen (originally for mobile phones). Each pixel would give rise to a different shooting angle when tapped/clicked on, with the bot's position being the origin of the directional vector. A large action space would take a much longer time for the model to explore and evaluate every single action, even though the difference between each action/angle is marginal. We first tried limiting it to 350 possible actions, with a fixed y-value of 470. We subsequently felt that 350 actions is still too large of an action space given our limited training time, and further reduced it by limiting it to only 25 evenly spaced angles/x coordinates. This reduction of action space allows the model to converge easier, but the model may not have picked the "correct" action due to the loss in granularity, i.e. the best action might have been in between two angles.

Thirdly, we also performed batch gradient updates. To help combat the wild fluctuations that we've observed, we switched from stochastic gradient updates to batch gradient updates. While SGD allows models to converge faster, it also introduces more noise per update[6]. 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 gameplays by averaging the gradients accumulated across them.

Fourthly, we added more layers to give rise to a deeper model. With the normalization of the input game state, the state space got significantly larger due to the encoding to one-hot vectors. Our initial state space of size 128 increased to a size of 1234. As such, the initial model architecture that we used consisted of 3 layers, with sizes of 128/128/1 for the actor and 128/128/numActions for the critic network. This was likely too small and shallow for our normalized state[7]. However, being wary of the risks involved in having excessive hidden weights, we eventually settled on a 5 layer architecture for our A2C model, with sizes of 512/256/128/128/25 for the actor and 512/256/128/128/1 for the critic network.

Finally, we also performed reward shaping. To give a better chance for the model to converge and perform decently, we shaped the reward function using our intuition and knowledge of the game to incentivize certain behaviors. 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.

---

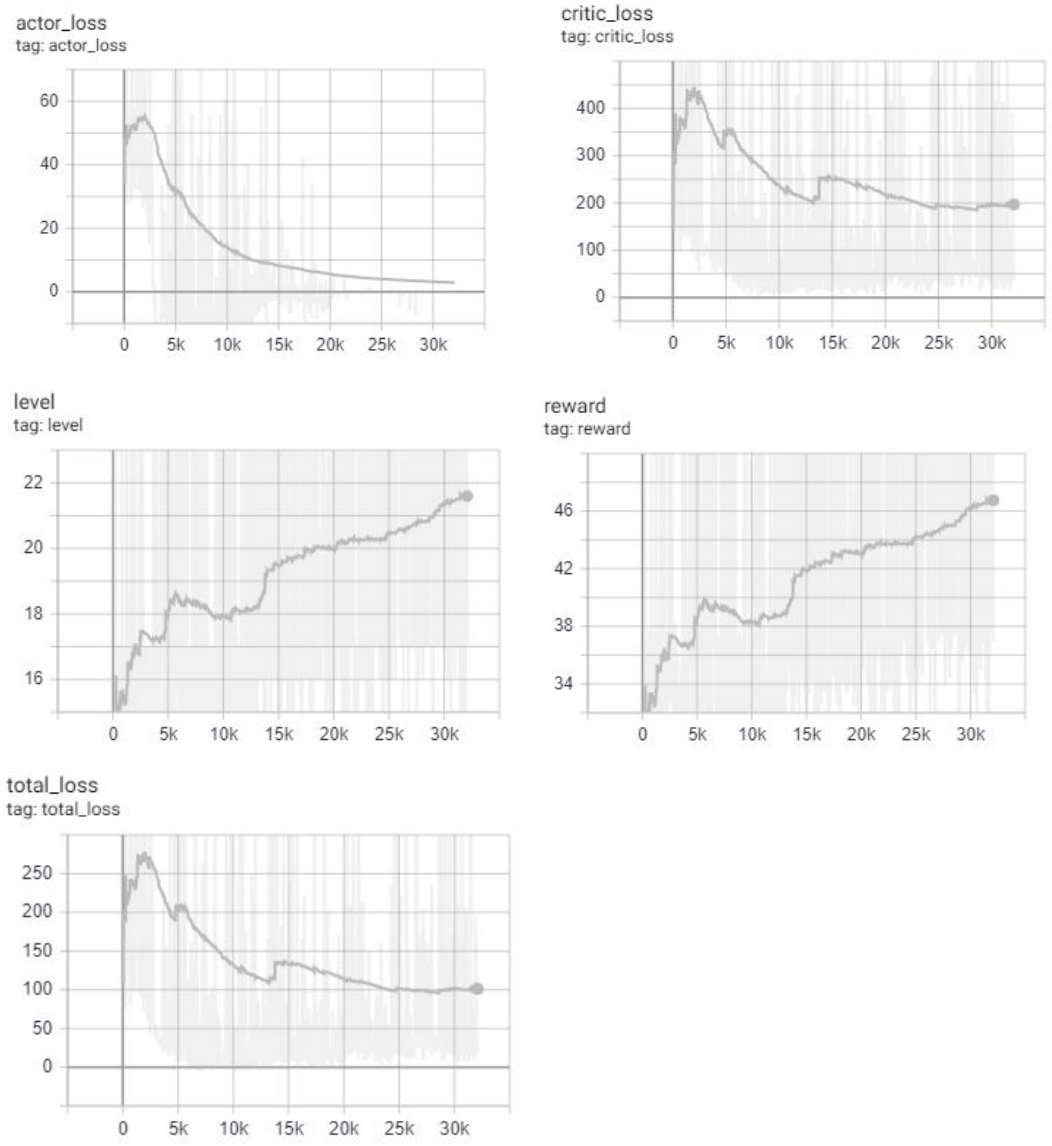
[6] "An overview of gradient descent optimization algorithms." 15 Jun. 2017, https://arxiv.org/pdf/1609.04747. Accessed 14 Dec. 2019.
[7] "The Number of Hidden Layers | Heaton Research." 1 Jun. 2017, https://www.heatonresearch.com/2017/06/01/hidden-layers.html. Accessed 14 Dec. 2019.

Our reward function is as follows, where b^i represents the i^{th} block out of a total of $B = 63$ blocks within the level map, and b^i_{row} represents the row of the block (an integer from 0 to 8, where row 0 is the topmost row) and E represents the number of empty spaces:

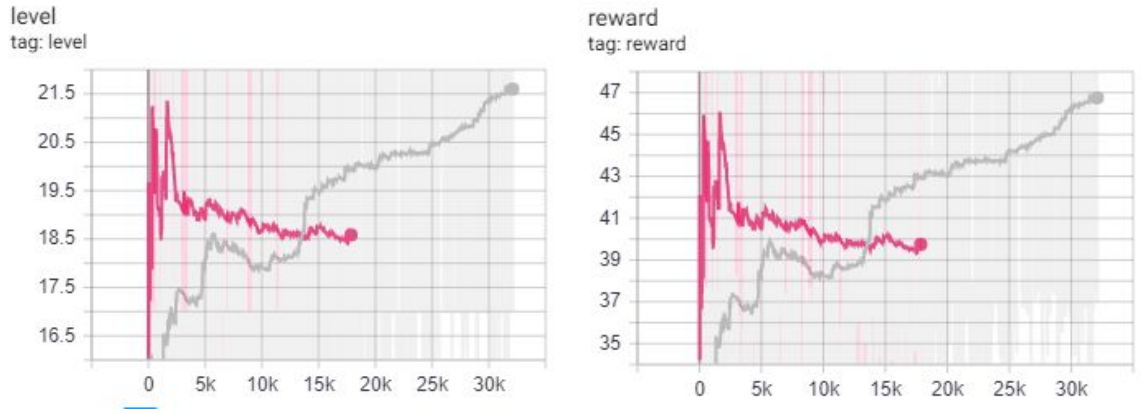
$$Reward = 2 - \sum_{i=0}^{B} 0.01 * b^i_{row} + \sum_{0}^{E} 0.01$$

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 lower while our level and reward charts produces an upward trend. To put the results into perspective, we benchmarked our model against 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 improvement.

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, thus more research and experimentation is required.

Native Game Code Base

In addition, even with the frame rate 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[8].

Source Code

The source code for the game, the Flask bridge and our model can be found at https://github.com/BrandonTJS/CSCI2470-BBTan.

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