Improved Reinforcement Learning in Multiple Environments

By Kaleo Ha'o

Definition

Overview:

Although reinforcement learning is the science of how a software agent learns to act properly in a given environment, its roots are found in behavioral psychology[1]. Consequently a simple way to understand reinforcement learning is to look at psychology's use of reinforcement to raise children[2]. When raising a child, if the child does something good like sharing his toys with a sibling, rewarding this behavior would encourage the child to share again in a similar situation. However, if the child instead does something bad like stealing a toy from a sibling, punishing this behavior would diminish the chances of the child stealing again in the future. If you take this example and replace the child with a software agent, you arrive at the definition of reinforcement learning. Here the goal is to design an agent that can master the decision making process in a given environment such that it acts optimally by maximizing rewards received from the environment. Formally, this is the science of the agent determining what action to take in a given state (or scenario), by determining the values of all possible actions in a given state.

Until recently reinforcement learning was limited to rather simple environments because reinforcement learning agents lacked the human ability to filter out information and intuitively observe the world at different levels of abstraction. In mathematical terms, reinforcement learning agents could not efficiently learn environments with high dimensional state spaces. An example of an environment with a high dimensional state space is a visual input. When confronted with a visual input a human filters out most of the information that comes in through the eyes[3]. For instance while driving a human might disregard a bird flying across the top of her visual field. A reinforcement learning agent, on the other hand, would treat the presence of the bird with the same importance as the presence of the cars on the road. High dimensional state spaces are not limited to the presence or absence of objects but also include levels of abstraction when seeing those objects[4]. When a human sees a cup on a table, she can seamlessly interpret that cup at different levels of abstraction. The cup could be a receptacle for quenching thirst, a mere decoration, a sentimental gift from a child, a weapon in a dire situation, or effectively non-existent if she were focusing on something else. When an agent has the visual input of a cup, it only sees one thing: a

---

[1] Niv, Y. Reinforcement learning in the brain. *Princeton University,* Psychology Department & Princeton Neuroscience Institute, https://www.princeton.edu/~yael/Publications/Niv2009.pdf

[2] CC BY-NC-SA. Univ. of Minnesota. Changing Behavior Through Reinforcement and Punishment: Operant Conditioning. *Introduction to Psychology,* 7.2.

[3] Haas, D. This is How the Brain Filters Out Unimportant Details. *Psychology Today* 11 Feb 2015, https://www.psychologytoday.com/blog/brain-babble/201502/is-how-the-brain-filters-out-unimportant-details

[4] Peterson, J. Three Forms of Meaning and the Management of Complexity. *University of Toronto*, Department of Psychology, http://www.jordanbpeterson.com/docs/434/Assigned_Papers/Peterson%20JB%20Three%20Kinds%20of%20Meaning%20Final%203.pdf

grid of pixels. Effectively, when in a complex environment a human is able to learn how to act while being bombarded by an infinite amount of information, but a reinforcement learning agent drowns in a sea of an infinite state space. This was the case until Google DeepMind combined Reinforcement Learning with Deep Learning to design an agent that mastered the Atari game system[5].

Problem Statement:

DeepMind's work inspired an Open AI research request to develop an RL (reinforcement learning) model that could solve all Open AI Gym Environments (with continuous action spaces) without changing hyperparameters[6]. The experiment documented in this paper tackled a reduced scope of that Open AI research request, by setting a goal to create a robust Q-Learning implementation that could solve two different Open AI Gym Environments (Flappy Bird and Monster Kong) without changing hyperparameters. Quantifiably and mathematically speaking, this means that the goal was to create a model that could accurately approximate Q-values given any state in two different learning environments consistently across multiple episodes (i.e. the model had to display evidence of learning multiple times in each environment). The inputs for this experiment were the live still images (or video frames) of the two game environments during simulation, reward values provided by the environment per agent action, and a 'Done' indicator which signaled when a game episode had reached a terminal state (i.e. agent dies or saves the princess). These inputs were used because they encompass the necessary components for reinforcement learning, which revolves around determining optimal action policies given occupied states. The input images provided observations to determine occupied states. The rewards provided by the environment helped determine optimal policies. The inputs were provided through Open AI's gym environments. These two environments were originally from the PyGame Learning Environment[7] and were adapted to be used through Open AI by Luis Sobrecueva[8].

A CNN (Convolutional Neural Network) was used as the model for approximating Q-values. CNNs have been typically used in image classification problems because of their ability to extract visual patterns in image data. This ability to extract visual patterns is effectively a form of focusing, where the CNN learns what to focus on and what to filter out based on a loss function. Since this experiment used images as input, it only made sense to make use of the CNN's pattern recognition ability to learn what to visually focus on during gameplay. However, unlike most image classifiers, the proposed model forewent the use of pooling layers to preserve spatial information received from gameplay[9].

Metrics:

The Flappy Bird environment is a side-scrolling game that has the agent navigating through gaps in pipes, where the agent dies if it hits a pipe, the ground, or the top of the screen. Everytime the agent passes a pipe, it receives a reward of +1. Everytime the agent dies, it receives a negative reward of -1. A simple

---

[5] Mnih, V. Playing Atari with Deep Reinforcement Learning. *arXiv*: 1312.5602v1 [cs.LG] 19 Dec 2013.
[6] https://openai.com/requests-for-research/#improved-q-learning-with-continuous-actions
[7] Tasfi, Norman. PyGame Learning Environment. http://pygame-learning-environment.readthedocs.io/en
[8] Sobrecueva, Luis, @lusbo, https://github.com/lusob
[9] Raval, S. Deep Q Learning for Video Games. *Math of Intelligence* #9, https://www.youtube.com/watch?v=79pmNdyxEGo

way to measure the effectiveness of the RL model in this environment was to track score (per episode). Here the lowest the RL model could score was -1 by dying before passing the first pipe. Passing pipes resulted in higher scores.

The Monster Kong environment is a knockoff of the original Donkey Kong arcade game and tasks the agent with avoiding fireballs while collecting coins and rescuing the princess at the top of the screen. The agent starts each game with three lives. Collecting a coin rewards the agent with +5. Rescuing the princess at the top of the screen rewards the agent with +50. If the agent gets hit by a fireball, the agent loses a life and receives a negative reward of -25. If the agent loses all three lives, the current game episode ends. A simple way to measure the effectiveness of the RL model in this environment was to track score (per episode). Here getting hit by three fireballs without collecting any coins resulted in the lowest possible score of -75. Collecting coins and avoiding fireballs resulted in higher scores.

Analysis

Data Exploration:

As mentioned above, the input for the RL model was the gameplay images. Flappy Bird produced gameplay images that were 288 x 512 pixels. The RL model took control of an agent (bird) which started toward the left of the screen. The pipes to be avoided were introduced from the right and scrolled across the screen toward the agent. At any given gameplay frame, the RL model could choose between two actions: 'Up' or 'None'. One of the tricky aspects of this game was the fact that gravity constantly acted on the agent. So consistently performing the 'None' action, caused the agent to fall instead of remaining static. Another problem was the fact that everything changed color between gameplay episodes. In one game, the agent was blue while in the next, the agent was red. In addition to all of this, another source of complexity was the background; one of the things the RL model needed to learn was to ignore the background. Luckily this environment had one very positive aspect that facilitated learning: the environment gave feedback quickly. In other words, the agent received a positive or negative reward within five seconds of gameplay by dying or passing a pipe. The usefulness of this fact became more apparent when analyzing the Monster Kong environment. Monster Kong produced gameplay images that were 465 x 500 pixels. The RL model took control of an agent (unidentifiable creature) at the bottom of the screen. The prized princess was always at the top of the screen. Near her was always a monster which spat fireballs that traveled down the screen. Gold coins were always scattered randomly throughout the screen. The agent had to traverse a ladder system to progress upward toward the princess. At any given gameplay frame, the agent could perform one of six actions: 'Left', 'Right', 'Up', 'Down', 'Jump', or 'None'. While this game didn't change colors or have a complex background, it had a rather sizable obstacle for reinforcement learning: long periods of time without receiving any rewards. Since the fireballs started at the top of the screen, the agent had to wait for the fireballs to traverse the entire screen before it had the chance to receive a negative reward. Worse yet, there were times when the agent started far away from any coins. This resulted in the agent going for periods of 20 or 30 seconds without receiving any form of reward.

Understanding the state spaces in these environments was tricky yet simple. Technically each environment had a high dimensional state space where a change in any pixel from gameplay would have constituted a different state. Luckily using a CNN to extract visual patterns made the states more humanly intuitive. This effectively turned gameplay states into simple descriptions. For instance, when playing Flappy Bird, a human wouldn't take every pixel into mathematical consideration. Instead, she'd focus on the agent's current vector velocity and the agent's location in relation to the approaching pipe gaps. As a result, the state witnessed in the Flappy Bird screenshot in Figure 1 would simply be a description: "Two upcoming pipe gaps are directly in front of the agent." Likewise in Monster Kong, a human would focus on the agent's location in relation to coins, fireballs, and the princess. As such, the state witnessed in the Monster Kong screenshot in Figure 1 would also be a simple description: "There are no fireballs approaching the agent, there are six coins somewhat close to the agent, and the princess is four terraces above the agent."

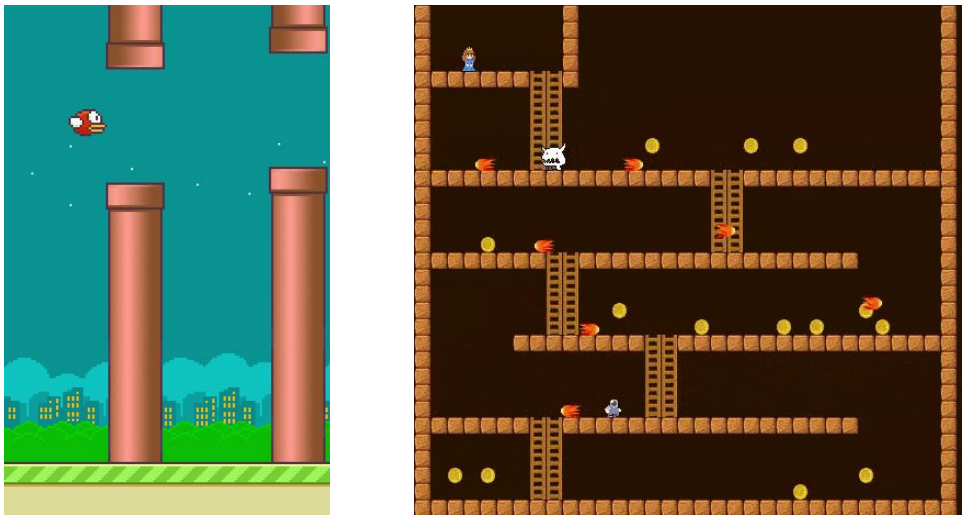

Figure 1 | Exploratory Visualization. Left: An example of a gameplay frame from the Flappy Bird environment. Here the agent is a red bird and is about to receive a positive reward of +1 by navigating through the gap in the red pipes. The pipes are moving right-to-left across the screen. **Right:** An example of a gameplay frame from the Monster Kong environment. Here the agent is on the second terrace and has just avoided the orange fireball to the left. The prized princess is at the top of the screen. The white, horned, fireball-spitting monster is also toward the top screen. When spat out, fireballs must traverse the entire screen to get to the agent's starting point (bottom terrace). Given the distance between the agent and fireballs or coins, it is very easy for the agent to receive zero rewards for long periods of time during training.

Algorithms and Techniques:

Since reinforcement learning is the science of mapping values to actions in a given state, the foundation for this experiment was the Bellman Equation[10]

$$\widehat{Q}(s,a) = r + \gamma max_{a'} \widehat{Q}(s',a')$$

which estimates the value of the RL agent taking action *a* given the RL agent is acting from state *s*. Here, the value of an action is known as an action's Q-value, hence the alternate name Q function. This Q-value of action *a* is the immediately observed reward *r* and the maximum possible Q-value of the next state *s'*, discounted by γ (where γ is between 0 and 1). Rolling out the second half of the Q function

$$\widehat{Q}(s,a) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \ldots$$

shows that the Q function (or Bellman Equation) estimates the maximum sum of future rewards discounted by γ at timestep *t*. Once Q-values are accurately estimated for all possible actions in a given state, a policy $\pi(s)$ is determined, which tells the agent which possible action has the highest Q-value (i.e. what to do next). The genius of the Bellman Equation is that it provides a recursive strategy for estimating Q-values by a process called Value-Iteration. However, this experiment replaced the use of Value-Iteration with the strategy of using a CNN (convolutional neural network) to directly approximate the Q function.

As discussed earlier, CNNs have shown lots of promise with learning visual data. In a similar fashion, the CNN in this experiment took gameplay frames as input and produced vectors of estimated Q-values with the same size as the number of possible actions in the given environment. In Flappy Bird, the vector output of the CNN had two values for the two possible actions in any given state. Likewise in Monster Kong, the vector output of the CNN had six values. Since estimating Q-values can be considered a form of regression, a Mean Squared Error loss function was used to drive the CNN's learning. However, the problem with this notion was that the true Q-value of any given state-action pair was unknown a priori, making the error of the CNN technically incalculable. To get around this the Bellman Equation was reintroduced in order to estimate true Q-values. These estimated true Q-values were run against the CNN's output to calculate error, as in the following equation:

$$E(s,a) = \widehat{Q}(s,a) - Q(s,a)$$
$$E(s,a) = [r + \gamma max_{a'} Q(s',a')] - Q(s,a)$$

[10] Alzantot, M. Deep Reinforcement Learning Demystified (Episode 2) - Policy Iteration, Value Iteration, and Q-Learning. *Medium*, 8 Jul 2017, https://medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-978f9e89ddaa

Here Q is the CNN which estimates Q-values, and error E, given state s and action a, is the Bellman Equation subtracted by Q. This equation is technically only an estimation of the error since the Bellman Equation also uses Q to estimate future rewards, but it worked in practice due to the fact that the Bellman Equation is closer to the true Q-value than is Q[11]. What made this equation work for the RL model was the fact that all variables were observable from gameplay, notated as transitions $\{s, a, r, s'\}$ (where s' is the subsequent state following the agent's taking action a from the original state s to receive reward r). To prevent the CNN's parameters (weights) from changing too rapidly and causing the RL model to crash, an additional change was made to the error function. The Q model in the Bellman Equation section of the error function was made to be a separate CNN, which updated its own parameters less frequently. To make this distinction, Q^* is used in the following updated error function:

$$E(s,a) = [r + \gamma max_{a'} Q^*(s',a')] - Q(s,a)$$

Updating Q^* at a slower frequency than Q, prevented the Bellman Equation section of the error function from changing too rapidly and made the RL model more stable.

To create training batches for Q, the RL model stored transitions observed from gameplay in a dataset as a form of Replay Memory. From this Replay Memory, the RL model randomly sampled transitions to create training batches. Q then trained on these batches using the prescribed loss function (which encapsulates E)[12].

In order to keep the model from becoming too unstable, reward clipping was implemented. Here, all rewards were clipped at -1 and +1. This mostly applied to the Monster Kong environment, which originally produced rewards of +5, +50, or -25. With reward clipping, getting hit by a fireball resulted in a -1 reward while collecting a coin and rescuing the princess each resulted in a +1 reward.

Although not implemented in the initial iteration of the RL model, Priority Sweeping was eventually introduced to accelerate learning. Priority Sweeping is a technique where a model's learning puts a higher emphasis on "higher priority" transitions[13]. This was useful because not all transitions are created equally. For instance in Monster Kong, an agent can learn more from getting hit by a fireball than jumping when there is nothing around it. One results in a negative reward and possibly a terminal state, while the other results in nothing. However, the random sampling of transitions treated both of these states with the same priority. Priority Sweeping addressed this lack of efficiency. In this experiment, the RL model used a rudimentary form of Priority Sweeping by isolating and repeatedly training on "higher priority" transitions. The RL model considered two scenarios a high priority: a terminal state and a transition sequence resulting in an episode score higher than the worst possible score (-1 in Flappy Bird and -3 in Monster Kong). So every time an agent transitioned into a terminal state, the RL model repeatedly trained on that single transition, according to a Terminal Repeat hyperparameter. Then it

[11] Suresh, H. Deep Reinforcement Learning. *Introduction to Deep Learning.* MIT 6.S191, https://www.youtube.com/watch?v=xWe58WGWmlk&t=1985s

[12] To clarify, the actual loss function is a Mean Squared Error function that uses $E(s,a)$. E is not the loss function in it of itself.

[13] Andre, D. Generalized Prioritized Sweeping. *U.C. Berkeley*, Computer Science Division.

looked at the episode's score and if the score was higher than the worst possible score, the RL model isolated all of the episode's transitions from the Replay Memory and repeatedly trained on those transitions, according to a Priority Factor hyperparameter. This Priority Factor hyperparameter was also a multiplier which got larger in correlation to performance. For example in Flappy Bird, the RL model trained on a game episode with score +3 more times than it trained on a game episode with score +1. Repeatedly training on these higher priority transitions effectively mimicked these higher priority transitions being added to the Replay Memory multiple times. However, repeatedly training on these transitions when they occured was computationally less expensive than actually adding them to the Replay Memory multiple times.

Priority was given to terminal states because the error calculated from a terminal transition is absolute. This is because Q no longer needs to estimate a future reward. As such, the error function at a terminal transition is

$$E(s,a) = r - Q(s,a)$$

This is radically different from all other transitions where Q^* is necessary to estimate future long-term reward and $E(s,a)$ is only an estimation of the error. Taking this into consideration and placing a high priority on terminal transitions accelerated learning and anchored the RL model to absolute Q-values instead of estimated true Q-values. Priority was also given to high performance episodes (where the agent scored higher than the worst possible score) to offset how easy it is to perform poorly in a game. For instance in Flappy Bird, there are more ways to die than there are to survive. As a result, Replay Memory was dominated by low scoring experiences, especially toward the beginning of learning. This resulted in the RL model quickly learning the Q-values of unfavorable actions and postponing the learning of Q-values for favorable actions. Placing a high priority on high performance essentially balanced out the Replay Memory between low score experiences and high score experiences, enabling the RL model to learn both favorable and unfavorable actions at a more similar rate.

Model Construction:

The construction of the RL model was broken down into two major sections: constructing the CNN architecture and tuning hyperparameters. Once the CNN was built, hyperparameters were tuned according to variance in model metrics (score). All hyperparameters with their initial values are as follows:

Hyperparameter	**Initial Value**	**Details**
Input Stack Size	4 frames	To facilitate learning, stacks of gameplay frames served as the states which were fed into the RL model's CNN. In other words instead of having a single gameplay frame entering the CNN, each input state contained the memory of the last few frames with the current frame, according to this hyperparameter. The details of this process are given in the **Data Preprocessing** section, below.

Hyperparameter	**Initial Value**	**Details**
Resize Target	80 x 80 pixels	Every gameplay frame was resized to the Resize Target in order to keep input size the same between both environments and to cut down on computation costs.
Batch Size	35	Size of training batch sampled from Replay Memory for CNN training
Future Reward Discount (γ)	0.99	Discount factor applied to all future rewards in the Bellman Equation (See above)
Starting Epsilon (ε)	0.99	The starting exploration rate or probability with which the agent performed a random action during gameplay.
Final Epsilon	0.1	The exploration rate linearly decayed until it reached this value.
Epsilon Decay Episodes	1,000,000	The number of episodes it took for Epsilon to decay to the Final Epsilon value
Replay Memory Size	500,000	Maximum number of transition saved in memory according to a Last-In, Last-Out policy
Training Frequency	1	Number of actions taken between batch training epochs: E.g. When set to '1', batch training would have occurred every time agent took an action. When set to '3', batch training would have only occurred every three times agent took an action. Etc...
Q* Update Frequency	1,000 epochs	Number of training epochs between each Q* update.
Random Actions to Start	100	Number of random actions at the start of gameplay to populate Replay Memory
Action Repeat	1	Number of times a chosen action is repeated, to control action frequency and reduce computation time, according to the idea that an agent may not have to decide an action at every single gameplay frame in order to learn the environment. E.g. When set to 10, RL model would have only chosen an action every 10 frames.
Optimizer	RMSprop	Optimizer used by the RL model's CNN.
Learning Rate	0.001	Learning rate used by the RL model's CNN.
Error Clipping	[-1,1]	Range at which error was clipped during loss calculation (to keep model stable): E.g. $E(s,a) = 1{,}000$ would have been clipped to '1' when set to [-1,1].

Hyperparameter	Initial Value	Details
Terminal Repeat	N/A	Used for Priority Sweeping, this number determined how many times the agent trained on a terminal transition. (Not included in initial iteration of RL model.)
Priority Factor	N/A	Used for Priority Sweeping, this was a multiplier used to determine the number of times an agent trained on a high performance game episode. (Not included in initial iteration of RL model.)

Benchmark:

The benchmark model for this experiment was a simple random agent. The random agent represents the antithesis of learning in any given reinforcement learning environment and is an agent that chooses a random action in any given state. Formally, the random agent played each game environment for 50 consecutive game episodes. In Flappy Bird, the random agent attained a -1 score for every episode, meaning that the agent never made it passed the first pipe (Figure 2). Since it is very possible for gameplay to become arbitrarily long in Monster Kong, the agent was given a five minute playing limit for each episode in Monster Kong. Here the random agent's scores ranged from -3 to 3, rarely went above +1, and hovered around the -1 to 0 area. As such, the RL model had to outperform the random agent benchmark in both environments in order to be considered successful for the problem statement.

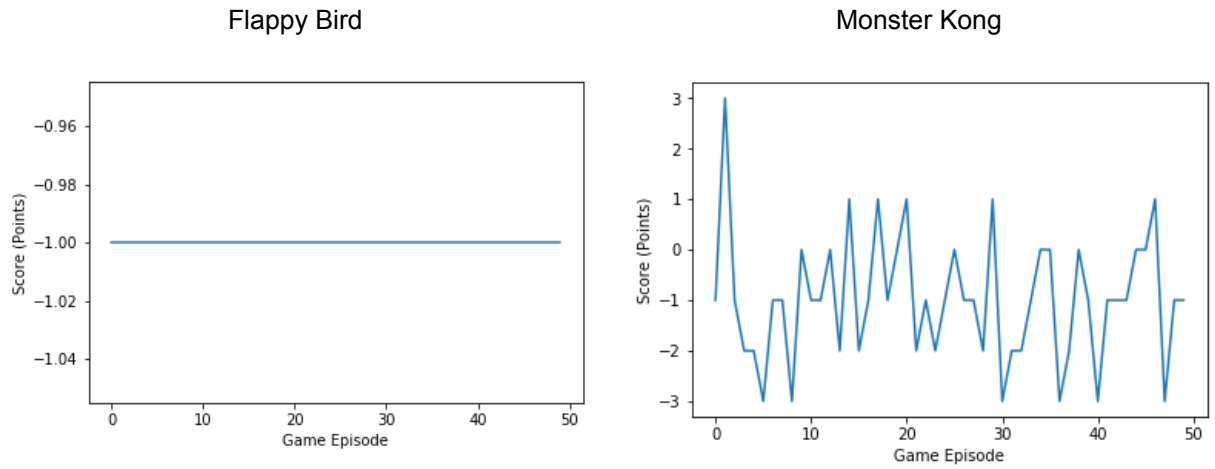

Figure 2 | Random Agent Simulation Results. Left: The random agent never makes it past the first pipe in Flappy Bird. **Right:** The random agent only scored above '1' once, but it achieved the lowest possible score of -3 six times.

Methodology

Data Preprocessing:

While implementing a CNN that directly viewed the raw gameplay frames may have been feasible, two preprocessing steps were taken to improve learning and minimize computation costs. The first preprocessing step turned the incoming raw gameplay frames to grayscale images and resized them to 80 x 80 pixels (final Resize Target changed to 100 x 100 pixels). This simultaneously reduced computation costs and made the RL model's CNN robust enough to handle the different sized game screens between both environments. The second preprocessing step stacked the current gameplay frame with the previous few frames to form a stack of gameplay frames, which served as the actual state of the RL model during gameplay. For example, if the Stack Size hyperparameter were set to 3, the state fed into the RL model's CNN would have been a stack of the current frame and two previous frames. This allowed the RL model to understand more about the current state. For instance in Flappy Bird, it's impossible to determine the current direction and velocity of the bird by observing a single frame. But, looking at multiple consecutive frames shows if the bird is currently falling or already moving upwards. At the beginning of gameplay, black screens were used to fill a frame stack to the intended Stack Size.

Implementation:

The RL model's exact implementation is given in the accompanying Jupyter Notebook, but it is summarized as follows:

Algorithm:

- Set training time and additional training parameters
- Initialize two identical CNNs: Q and Q^*
- Commence Gameplay:
    - **Random Actions:** Model performs random actions to start populating the Replay Memory with gameplay experiences, $\{s, a, r, s', t\}$ (where t notes whether s' is a terminal state)
    - **Learning (*LOOP*):**
        - With probability ε model performs a random action, *ELSE* model performs action chosen by Q.
        - Model saves experience in Replay Memory.
        - *IF Training Frequency* is met, training batch is sampled from Replay Memory, and Q trains on batch:
            - *IF* experience results in a terminal state, loss function uses $E(s,a) = r - Q(s,a)$
            - *ELSE* loss function uses $E(s,a) = [r + \gamma max_{a'} Q^*(s',a')] - Q(s,a)$
        - *IF Update Frequency* is met, update Q^* to match Q*'s* parameters
        - *IF* agent enters terminal state, commence **Priority Sweeping** before starting next game episode:
            - Train Q on terminal experience repeatedly according to *Terminal Repeat* hyperparameter
            - *IF* model scores higher than worst possible score, train Q on experiences from episode repeatedly according to *Priority Factor* hyperparameter
            - Start next game episode
    - When training time eclipses or epsilon reaches its final value, *END*

At times training required 12 hours before the slightest evidence of learning became apparent during the RL model's first few iterations. As such, "Pickup Training" functionality was added to the model. This saved Q's parameters, Q^*'s parameters, and the Replay Memory, enabling subsequent training sessions to pick up where a previous training session had ended.

Code:

While most of the code (see accompanying Jupyter Notebook) was easy to implement, there was a complication worth noting. CNNs are usually easy to implement once they are built because input data and target data are usually readily available. However, this experiment did not have any target data to feed into the RL model's CNN a priori. In other words, the target data had to be calculated in real time as

the RL model trained its CNN. To accomplish this, target Q-values were calculated using the Bellman Equation immediately after a training batch was sampled, but that led to another complication. The Bellman Equation could only calculate the target Q-value for a single state-action pair. The CNN, on the other hand, predicted a vector of all possible Q-values for a given state. For instance in Monster Kong, the Bellman Equation calculated a single target Q-value for a single transition, but the CNN required a vector of six target Q-values to perform training on that same transition. To get around this, the additional target Q-values were calculated by using the CNN itself as the target calculator. In other other words, the CNN's own predictions were used as targets in conjunction with the target produced by the Bellman Equation in order to complete target vectors . These faux targets naturally produced a zero loss on all predicted Q-values except for the Q-value correlating to state-action pair being trained on. Details of this implementation are given in the **Additional Functions** section of the accompanying Jupyter Notebook.

CNN Architecture:
The final CNN architecture (for both Q and Q^*) was inspired by Google DeepMind's work with Atari[14] and consisted of an input layer (gameplay frame stacks), four hidden layers, and an output layer. Details of these layers follow:

Input Layer	Frame Stack	Stack of current gameplay frame and recent frames, serving as the model's state
Hidden Layers	Convolutional	32 Filters, Kernel Size 8, Stride 4, ReLu Activation
	Convolutional	64 Filters, Kernel Size 4, Stride 2, ReLu Activation
	Convolutional	64 Filters, Kernel Size 3, Stride 1, ReLu Activation
	Fully Connected, Dense	515 Nodes, ReLu Activation
Output Layer	Fully Connected, Dense	Output matches number of possible actions for each environment and uses a Linear Activation.

Refinement:

With the CNN and Algorithm successfully built, the refinement process focused on fine tuning hyperparameters through a tedious trial and error process. Details of each hyperparameter's fine tuning are as follows:

[14] Mnih, V. Human-level control through deep reinforcement learning. *Nature* 518, 529 - 533 (2015).

Hyperparameter	Initial Value	Final Value	Details
Input Stack Size	4 frames	4 frames	The stack size had a huge impact on the memory size (GB) of the Replay Memory (which was the largest limiting factor for preserving RAM) . E.g. Going from one frame to two frames instantly doubled the Replay Memory's size. So while larger stack sizes showed promise for improving learning, 4 frames turned out to be a workable number for keeping the model stable.
Resize Target	80 x 80 pixels	100 x 100 pixels	Increasing the Resize Target improved learning while at the same time increasing the Replay Memory's size.
Batch Size	35	15	Larger batch sizes seemed to dilute the loss calculations during training. This could be due to the Replay Memory consisting of very similar experiences toward the beginning of learning. In the end, lowering the batch size proved effective.
Future Reward Discount (γ)	0.99	0.9	Lowering the Future Reward discount allowed the RL model to be more short-term focused. This proved effective since the RL model did not need to think far ahead in order to learn how to survive and perform better than the benchmark.
Starting Epsilon (ε)	0.99	0.99	This was kept at 0.99 as common with Reinforcement Learning practices.
Final Epsilon	0.1	0.25	Increasing the Final Epsilon actually improved learning and reduced overfitting. The reason for this is a mystery at this point.
Epsilon Decay Episodes	1,000,000	1,000	This number was lowered in order to reduce the time required to train the RL model.
Replay Memory Size	500,000	4,000	This was the primary limiting factor for this experiment. As noted before, the initial Replay Memory size caused the RL model's computer to run out of RAM. In the end, 4,000 was both effective for learning without crashing the computer. It is worth noting, that better hardware would reduce, if not, eliminate this limiting factor.
Training Frequency	1	4	Raising this number from 1 to 4 (reducing training frequency) improved model stability, albeit marginally.
Q* Update Frequency	1,000 epochs	100 epochs	Increasing how often Q* was updated (lowering the Update Frequency) both accelerated learning and made the model more unstable, causing estimated Q-values to jump around more often. 100 epochs ended up being an effective and stable (enough) number.

Hyperparameter	Initial Value	Final Value	Details
Random Actions to Start	100	1,000	Increasing this number while decreasing batch size improved learning.
Action Repeat	1	5	The initial value of '1' proved to be overkill. Considering the agent averaged 50 frames before reaching the first pipe in Flappy Bird, an Action Repeat of '1' made the long-term reward values so small that learning was virtually impossible at the beginning of gameplay.
Optimizer	RMSprop	Adam	Adam performed better than RMSprop
Learning Rate	0.001	0.00025	Decreasing the learning rate made the RL model more stable.
Error Clipping	[-1,1]	Unclipped	At first error clipping caused the RL model to be very stable. However with the introduction of Priority Sweeping, error clipping proved very detrimental. Priority Sweeping accelerated learning, but it also caused the model's CNN weights to explode at times, producing large Q-values (positive or negative). Upon the weights exploding, a clipped error made it virtually impossible for the model to self correct its weights. In the end, implementing Priority Sweeping and disabling error clipping proved extremely effective, albeit more chaotic, for learning.
Terminal Repeat	N/A	2	This number needed to be kept small as large numbers made the RL model very unstable.
Priority Factor	N/A	3	As with Terminal Repeat, this number needed to be kept small as large numbers made the RL model very unstable.

Results

Model Evaluation & Validation:

The final structure and final hyperparameter settings, noted above, were chosen as an answer to the experiment's proposed problem because they enabled the RL model to show evidence of learning in both game environments during both training and testing. To track learning, estimated Q-values were tracked from a set of randomly held out state-action pairs during training. Since the mathematical goal of this experiment was to accurately estimate Q-values, tracking these held out state-action pairs showed whether the RL model was converging or not. In this case, convergence would imply that the RL model was

learning a state-action pair's actual Q-value, while a lack of convergence would imply the opposite. While initial tracking of Q-values showed lots of divergence or oscillation, the final tracking displayed convergence. In fact in Flappy Bird, Q-values converged fairly quickly for the held out state-action pairs (Figure 3).

The final iteration of the RL model was also chosen because of its level of robustness. While this is in a sense built into the nature of the proposed problem, it is still worth noting. Accurately estimating Q-values with a high dimensional state space is hard enough, but estimating Q-values in two different

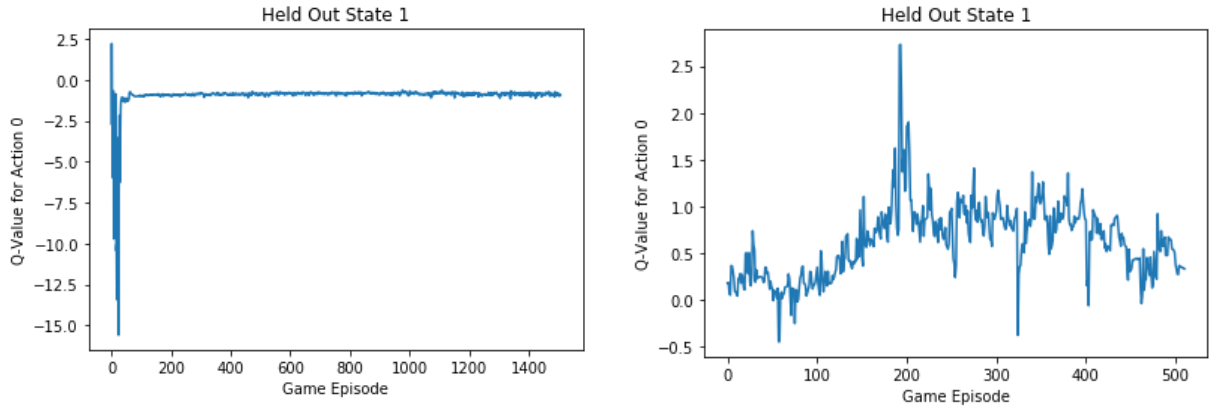

Figure 3 | Training Metrics. Left: Estimated Q-values for one of three held out state-action pairs tracked from the Flappy Bird environment. Here convergence happened fairly quickly. Although not included here, all three held out states tracked similarly to this one. **Right:** Estimated Q-values for one of three held out state-action pairs tracked from the Monster Kong environment. As expected with a more complicated environment, convergence seemed to occur more slowly and in a more chaotic fashion.

environments by definition requires a high level of robustness. This necessity for robustness is compounded by the fact that the Flappy Bird and Monster Kong environments are extremely different from one another. Flappy Bird is a side-scrolling game while Monster Kong isn't. Flappy Bird only has two possible actions while Monster Kong has six. Flappy Bird tasks the agent with flying through pipes while Monster Kong tasks the agent with collecting coins, avoiding fireballs, and rescuing a princess. Flappy Bird is fast paced while Monster Kong is slow paced. They even have different game screen sizes and different color schemes. In the end, the fact that the final RL model learned both environments is a testament to its robustness.

Justification:

The final RL model performs much better than the benchmark model (random agent) in both environments. To test the final RL model, the RL model was run through the same testing simulation as the benchmark model following training. Here, the RL model played 50 consecutive game episodes in each environment (with a five minute episode time limit for Monster Kong) while episode scores were tracked for final evaluation. To prevent overfitting, the RL model also operated with a 0.05 exploration factor (ε). This means that during testing, the RL model had a 5% chance that it would act randomly each

time it chose an action. This also effectively turned the game environments into stochastic environments, making them harder to operate in. Every time the RL model's actions didn't go as planned, it would have to readjust its actions on the fly in order to succeed. The final RL model performed much better than the benchmark model in Flappy Bird. While the benchmark model never even made it passed the first pipe, the final RL model flew passed the first pipe four times. Even better, the final RL model once made it passed three pipes. The final RL model also performed much better than the benchmark model in

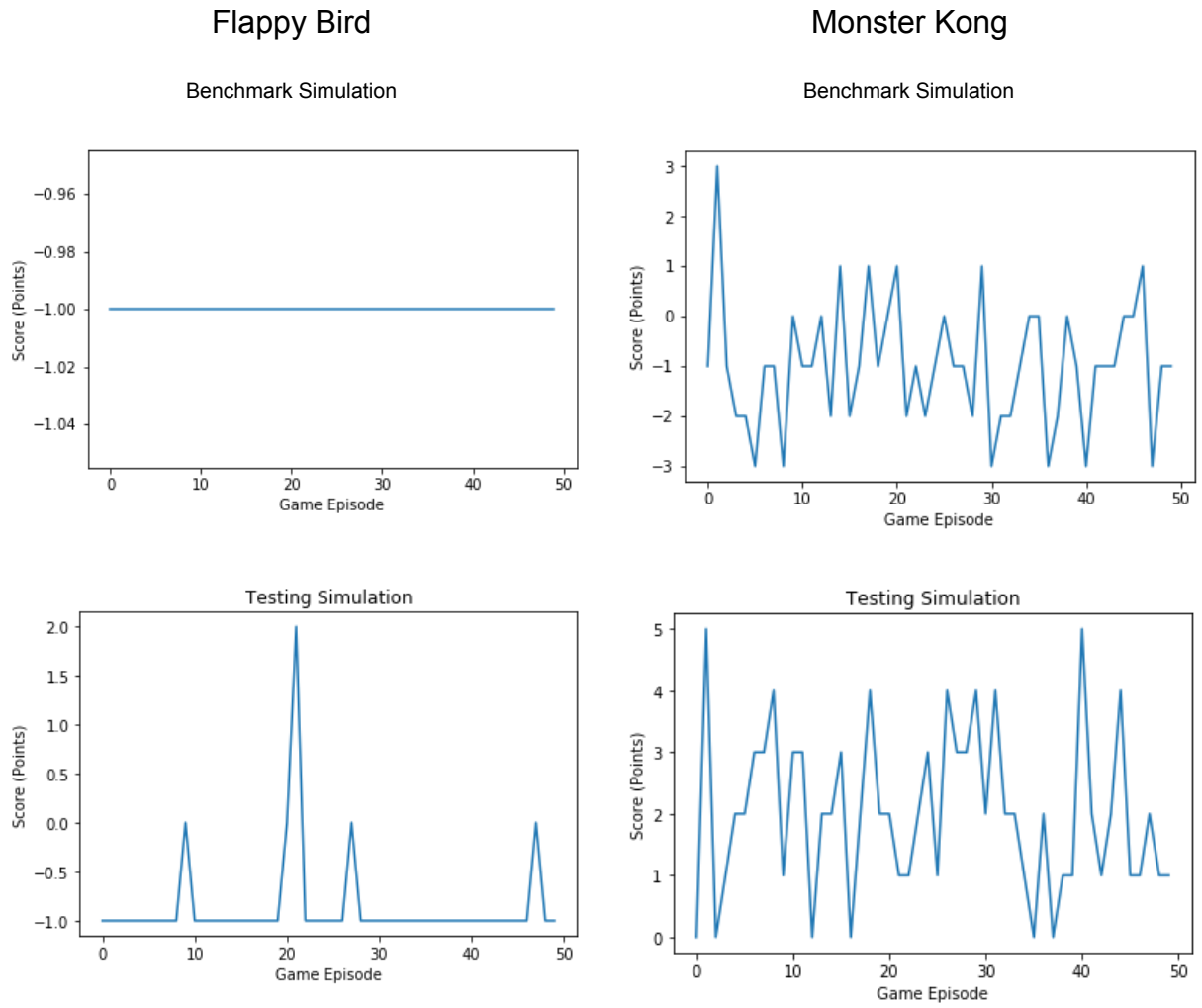

Figure 4 | Testing Metrics. Top-Left: Benchmark model's performance in Flappy Bird, using random agent. **Bottom-Left:** Final RL model's performance in Flappy Bird. **Top-Right:** Benchmark model's performance in Monster Kong, using random agent. **Bottom-Right:** Final RL model's performance in Monster Kong.

Monster Kong. Unlike the benchmark model, the RL model not once received the lowest score possible of -3. While the benchmark model rarely scored higher than +1, the final RL model scored mostly higher than +1. Additionally the final RL model attained a high score of +5, whereas the benchmark model only managed a high score of +3. All in all, the final RL model was able to show evidence of learning across both game environments by performing better than the benchmark model.

Conclusion

Reflection:

Although the experiment turned out as expected, implementing this experiment proved difficult and time consuming. The overall plan was simple: test the random agent, build the CNNs, build the algorithm, and then tune hyperparameters. Aside from debugging, everything went according to plan until hyperparameters needed to be tuned. Here the two biggest obstacles were time requirement and computation cost. The RL model crashed during its first training session when the model's computer ran out of RAM. The main limiting factor for RAM turned out to be the Replay Memory. To fix this, the Replay Memory's capacity was lowered, but this caused the RL model to forget gameplay more quickly. This put the entire experiment in a difficult spot because a shorter memory span required the RL model to learn more quickly from its experiences, which was in opposite to the initial learning strategy. Initially learning was supposed to be slow in order to prevent the RL model from becoming unstable. This is why the loss function was clipped to the range [-1, +1] during the RL model's first several iterations. In fact, the first several iterations of the RL model required 12 hours of training, before learning could be deemed effective or non-existent. Luckily everything changed when Priority Sweeping was introduced with the sole purpose of accelerating learning. This made the RL model more unstable and prone to exploding weights. In order to self-correct the instability, the clipped loss function was eventually unclipped, resulting in the RL model only requiring half an hour of training before learning could be deemed effective or non-existent. At that point, hyperparameter tuning became a game of balancing learning speed and model stability. Eventually this resulted in the final RL model only needing five hours of training in order to perform better than the benchmark model in each environment.

Looking back, the game of building the final RL model can be surmised by looking at the Monster Kong simulation's held out state-action pairs' Q-values (Figure 5). Here the Q-values are complex, straining to converge yet jumping around every so often. Even though the Q-values of all three state-action pairs follow a similar shape, they are subtly unique from one another. Although these Q-values do not converge as well as possible, getting these Q-values to this point took lots of time, trial, and error. Initial iterations showed no convergence whatsoever. The next batch of iterations showed Q-values that incessantly oscillated between positive and negative values. When Priority Sweeping was introduced, Q-values at times exploded to values nearing one million. This was hilarious and devastating since the maximum Q-value in the given environment had a limit of 11 (with future reward discount $\gamma = 0.9$)[15]. Luckily after unclipping the loss function, the Q-values converged enough to enable the RL model to perform well. In a poetic sense, the graphs in Figure 5 represent the overall process of completing this experiment: a chaotic dance of trial and error, straining to meager success.

[15] Since rewards are clipped to [-1,+1] and future rewards are discounted, using a geometric sequence with the maximum possible reward of +1 calculates an upper bound of 11. However, this is highly unlikely since it would require receiving a +1 reward from every future action.

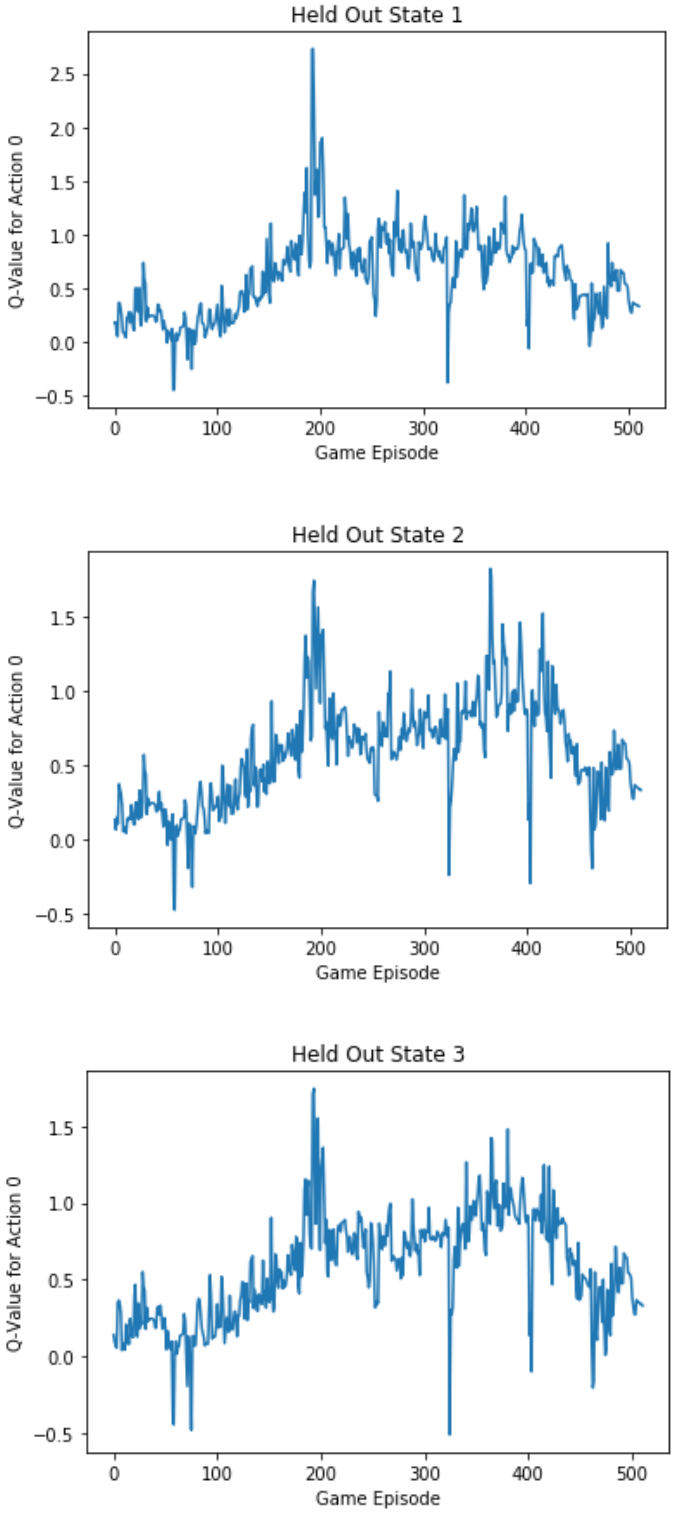

Figure 5 | Free-Form Visualization: Estimated Q-values for all three held out state-action pairs from the Monster Kong environment.

Improvement:

Even though this experiment's solution solves the proposed problem statement, it is far from perfect and can be improved in a number of ways. The first and most important way to improve the experiment is to widen the scope of the problem statement to include more environments. This would necessitate an even higher level of robustness and help the advancement of general artificial intelligence. Another way to improve this experiment would be to allow for longer training times. Even though the final RL model performs better than the benchmark model, allowing for longer training times (30+ days) may lead to human (or possibly superhuman) level performance. Another way to improve this experiment would be to use better training/testing metrics for the Monster Kong environment. This could be easily achieved by tracking how many times the agent collects a coin or makes it to the end of an episode without dying. Such metrics would have allowed for more qualitative comparisons between the benchmark model and final RL model. Finally, another simple way of improving this experiment would be to use a stronger computer. Although this improvement goes without saying for any experiment, it should be specially noted in this scenario since RAM became the primary limiting factor for the RL model's ability to remember past experiences.