Learning to Play Super Smash Bros. Melee with Delayed Actions

Yash Sharma
The Cooper Union
ysharma1126@gmail.com

Eli Friedman
The Cooper Union
eliyahufriedman@gmail.com

Abstract

We used recurrent neural networks to teach a computer to play Super Smash Bros. Melee in a more humanlike way. Previously, neural network agents have been taught to play Super Smash Bros. on a competitive level with humans, but they were "too good"–unrealistically so. The computers could react much faster than a human ever could. However, when these agents were given human-like reaction speeds, they performed much worse than humans. We propose a solution to this problem by adding recurrence to the architecture, so that the computer could "remember" what it saw a few frames ago and act appropriately even though its actions are delayed. Our solution stabilizes the training of competitive agents under human-level delay, as evidenced by qualitative and quantitative results against both the built-in AI and other trained agents. With stronger training partners to play against, these agents should eventually be able to beat the world's best.

1 Introduction

Deep Learning has achieved wide success in many applications, particularly in utilizing convolutional architectures for image classification. Neural networks have also had wide application in game-playing, particularly in recent times. Games provide excellent testbeds for RL/AI algorithms. Even as far back as the 1990s, agents have been trained to successfully play board games–the TD-Gammon Tesauro [1994] agent plays Backgammon by using a neural network to approximate the value function, yielding human-level performance. Most notably, AlphaGo Silver et al. [2016], in March 2016, was able to defeat Lee Sedol, an 18-time world champion Go player, with a combination of deep neural networks and tree search. For further information on applications of deep reinforcement learning in games, please refer to Li [2017]

In previous work, Firoiu et al. [2017], deep reinforcement learning techniques were applied to Super Smash Bros. Melee (SSBM). Released in 2001, SSBM still sports an active tournament scene and fanbase. Players continue to increase in skill, in order to keep up with the evolving metagame. The trained agents set the state-of-the-art in this environment, surpassing the abilities of ten highly-ranked human players.

However, many who played against the agent felt as if the agent was taking advantage of its unrealistic reaction speed: 2 frames (33ms), compared to over 200ms for humans. Despite it being quite apparent that the AI had learned how to effectively utilize the game's mechanics, its strategy did seem to be dependent on successfully reacting in order to counter the opponent's approach. Hence, by training an agent to function under human-like delay, the playing field would be evened, and the model's performance could be more fairly evaluated.

Therefore, in an attempt to resolve the issue, the authors trained a network which took in observations at previous frames as input, along with the actions taken on those frames. Employing this "stacking frames" technique was sufficient to train fairly strong agents with delay 2 or 4, but performance dropped off sharply at 6-10 frames. The authors suspected that the cause for this handicap is the

further separation of actions from rewards with delay, making it harder to tell which actions were really responsible for the already sparse rewards.

In this paper, we propose an alternative to the "stacking frames" approach, by re-exploring the task of training a competent recurrent agent. RNNs are not capable of fully remedying the issue with separation, but can handle this better by factoring in temporal dependencies. We demonstrate much improved training of an agent under significant delay, by evaluating the performance against the built-in AI.

2 Recurrent Neural Networks

Before delving into the application, we felt it would be helpful to overview the particular neural network architecture utilized here. Much of this section is derived from the *DL4J* guide to Recurrent Networks and LSTMs. Please refer for further detail.

2.1 Recurrent Networks

In order to adequately understand recurrent networks, one first needs to understand the basics of feedforward networks. In the case of feedforward networks, input examples are fed to the network and transformed into an output.

Recurrent networks, on the other hand, take their input from both the actual input and the previous output. The output of a recurrent network at time *t* is a function of both the input at time *t* and the output of time *t-1*. See Figure 1 for two representations of a recurrent network.

Recurrent networks can be distinguished from feedforward networks by their feedback loop. By ingesting their own outputs moment after moment as input, they can learn to remember past inputs. This allows for RNNs to utilize contextual information in the sequence itself. The sequential information is preserved in the recurrent network's hidden state, which manages to span many time steps as it cascades forward to affect the processing of each new example.

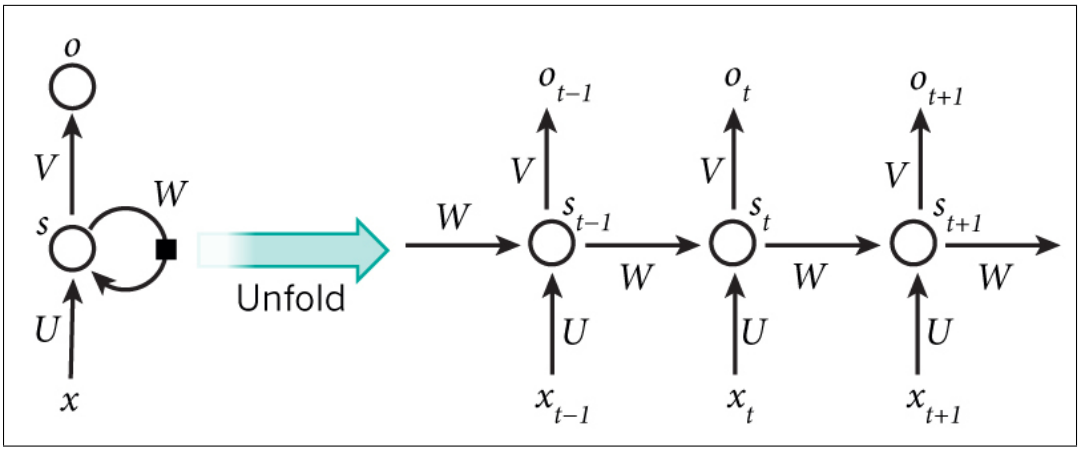

Figure 1: Unfolding of Recurrent Neural Network involved in Forward Computation

2.2 Vanishing (and Exploding) Gradients

Like most neural network architectures, recurrent networks are not a novel concept. By the early 1990s, the vanishing gradient problem emerged as a major obstacle to recurrent network performance. When one feeds the gradient back through a recurrent neural network for many time steps, the gradient is multiplied by the same weight matrix multiple times. If the singular values of the weight matrix are less than one, then the gradient will tend to die out as it travels back through the network. If the singular values are greater than one, the gradient will tend to rapidly increase. Either way, learning becomes very difficult for recurrent networks unrolled over many time steps.

2.3 LSTMs and GRUs

In the mid-90s, a variation of recurrent net with so-called Long Short-Term Memory units, or LSTMs, was proposed as a solution to the vanishing gradient problem Hochreiter and Schmidhuber [1997].

LSTMs propagate the internal state forward using a unit weight connection, so that the state neither vanishes nor explodes as it propogates forward. LSTM units also feature learnable gates, so that the network can learn whether to forget the internal state, or allow it to be influenced by the input. The gates allow the LSTM unit to act as a memory cell that the network can write to, read from, or simply remember a value.

The diagram provided (See Figure 2) illustrates how data flows through a memory cell and is controlled by its gates.

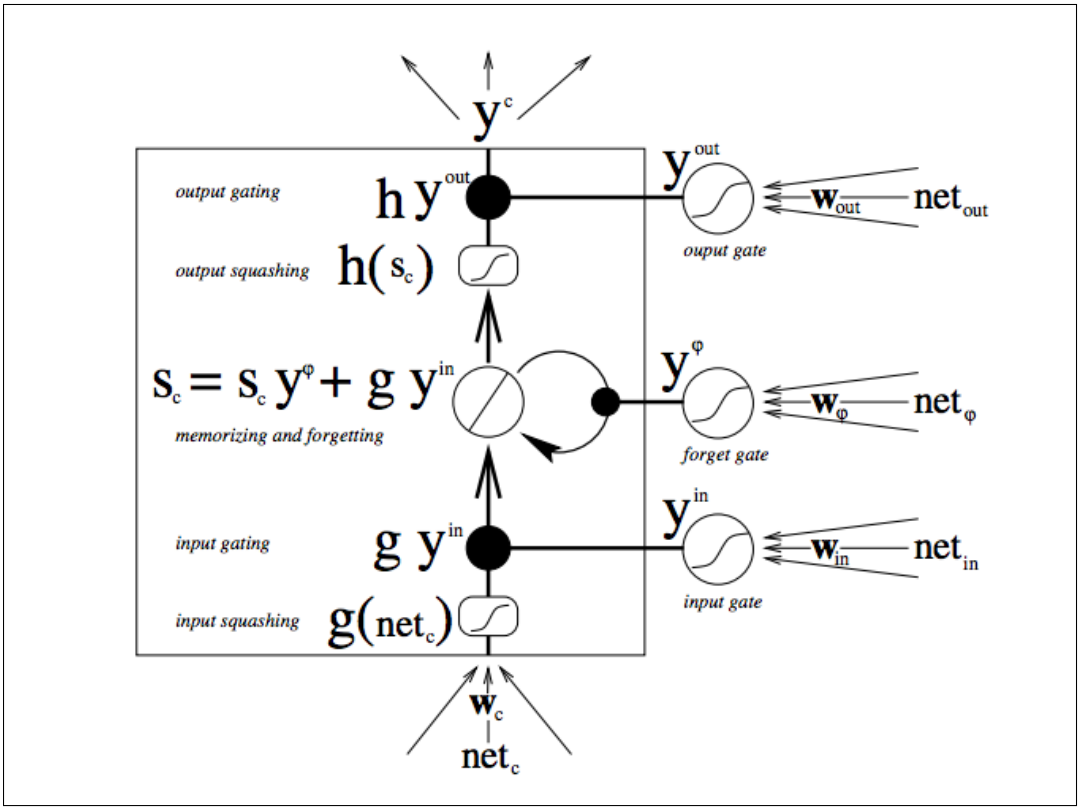

Figure 2: LSTM Diagram

Starting from the bottom of the cell, the triple arrows indicate that information flows into the cell from multiple points. The combination of present input and past state is fed not only to the cell itself, but also to each of its three gates, which will decide how the input will be handled.

The black dots are the gates themselves, which determine whether to let new input in (input gate), erase the present cell state (forget gate), and/or let that state impact the network's output at the present time step (output gate).S_c is the current state of the memory cell, and gy_in is the current input to it. Each gate can be open or shut, and each gate will recombine their open and shut states at every timestep. The cell can forget its state, or not; be written to, or not; and be read from, or not, at each time step, and those flows are represented here.

A Gated Recurrent Unit, or GRU, is basically an LSTM cell without an output gate, therefore fully writing the contents from its memory cell to the larger net at each time step. There have been studies, such as Steckelmacher and Vrancx [2015], which have indicated that for Reinforcement Learning applications, GRUs might perform better, which is why they were utilized in our implementation.

3 Resolving the Action Delay Problem

We delayed our agent's actions by d time steps, so that at time t, the agent would choose an action that would be executed at time $t+d$. The addition of recurrence allows the agent to learn about the temporal dependency between observations and actions at different times. This helps it learn what action it should perform d time steps from now.

We set the agent so that it would only choose a new action every 3 frames. This value was estimated based upon the length of the character's short hop, which is a game mechanic absolutely essential for high-level competitive play. We set d to be 12 frames, since human-level delay is approximately between 12-15 frames.

In theory, a recurrent neural network should be able to handle any amount of delay. The challenge however is to stabilize the training process of the network, so choosing the right hyperparameters becomes a crucial task.

4 Actor Critic vs Q-Learning

In Firoiu et al. [2017], the authors used two main classes of model-free RL algorithms, Q-learning and policy gradients, in order to learn the agent's policy (mapping of states to actions). In Q-learning, one attempts to learn a function mapping state-action pairs to expected future rewards, and uses that to construct a policy which always takes the best action under the learned Q. Policy gradient methods instead directly update the policy based on experience. Please refer to Sutton and Barto [1998] for further detail.

When evaluating performance results, the authors found that Q-learners did not perform well when learning from self-play, or in general when playing against other networks that are themselves training. However, they found that Q-learners perform reasonably well against fixed opponents, such as the in-game AI and a set of benchmark agents. This makes sense because Q-networks do not learn well when faced with a non-stationary target–hence the experience replay dataset in Mnih et al. [2015]. When playing against a learning agent, even an experience replay dataset will not help, since the opponent is still learning.

As our initial goal was to consistently succeed against the built-in AI under any amount of pertinent delay, the Q-learner's failings against learning agents were not particularly relevant. Therefore, due to the model's relative interpretability and availability of literature, we decided on training a successful recurrent Deep Q-network. However, this failing proved to be an issue when attempting to bolster the performance.

We decided to not only quantitatively, but to also qualitatively evaluate the policy that the network learned. The authors found that against the in-game AI, Q-learners would consistently find the unintuitive strategy of tricking the in-game AI into killing itself. Despite the ingenuity in the strategy, we would expect this policy to not prove to be as successful against competent players, due to their lack of predictability. Hence, we made it a point to not only evaluate on the quantitative reward metric, but also on the qualitative "eye test."

5 Hyperparameter Tuning

5.1 Exploration vs Exploitation

A common problem in reinforcement learning is finding a balance between exploration and exploitation. To what extent should the agent take advantage of what it has already learned (exploitation) and to what extent should it explore new possibilities (exploration)?

In each state (except a terminal state), an agent must select an action. The simplest way in which to decide on an action to take is greedy selection: the agent always selects the action with the highest state-action value. This method is pure exploitation. More sophisticated methods aim to achieve a balance between exploration and exploitation.

ϵ-Greedy selection balances the pure exploitation approach with some exploration by introducing an ϵ parameter that defines the small probability that the agent will choose uniformly between its actions. With probability $1 - \epsilon$, the agent will choose the greedy action.

An alternative is Boltzmann selection, which takes into account the relative value of the state-action values. The probability of an agent selecting an action depends on how high that action's value is compared to the other state-action values. So, if one action's value is much higher, that action is most likely to be taken, but if there are two actions with high values, both are almost equally likely. The hyperparameter here is the temperature parameter, which can be increased to increase the exploration rate. Ideally, the temperature should be chosen to match the scale of the Q-values.

As the SSBM environment has such a large state space, it is prohibitive to exhaustively explore the entire space. Therefore, both ϵ-Greedy selection and Boltzmann selection were used to further explore promising actions, and tuning these parameters proved to be pivotal for converging to a successful policy.

5.2 Discount Factor

In reinforcement learning, the discount factor controls how much the agent should consider future rewards while learning. A very large discount factor will force the agent to only consider the effects of its actions with respect to the reward one time step ahead, whereas a very small discount factor will force the agent to consider how its current action will affect all future rewards. As SSBM is a very fast-paced game, we decided that a larger discount–setting rewards 2 seconds into the future to be worth half as much as rewards in the present–was more helpful than a smaller discount.

With the discount factor set so high, the discounted reward summation did not have to be prematurely truncated for computational feasibility, and we were able to utilize rewards from the entire episode.

5.3 Weight Normalization

In Firoiu et al. [2017], the authors noticed that the practical success of first-order gradient based optimization is highly dependent on the curvature of the objective that is optimized. If the error surface is relatively flat, a larger step should be taken; if it is very curved, then a smaller step should be taken. If the condition number of the Hessian matrix of the objective at the optimum is low, the problem is said to exhibit pathological curvature, and first-order gradient descent will have trouble making progress Sutskever et al. [2013].

There are several approaches for improving the conditioning of the cost gradient for general neural network architectures. One is to explicitly left multiply the cost gradient with an approximate inverse of the Fisher information matrix, thereby obtaining an approximately whitened natural gradient. This approach was not taken as utilizing natural gradients significantly slows down unrolling the network. Rather than dynamically constructing the graph at execution time, an unrolled length for a fixed length needs to be created.

Alternatively, standard first order gradient descent can be used without preconditioning, but the model parameterization can be changed to give gradients which approximate natural gradients. Batch Normalization Ioffe and Szegedy [2015], a method where the output of each neuron (before application of the nonlinearity) is normalized by the mean and standard deviation of the outputs calculated over the examples in the minibatch, reduces covariate shift of the neuron outputs. The authors suggest it also brings the Fisher matrix closer to the identity matrix. However, this method introduces dependencies between examples in a mini-batch, making it unsafe to use in recurrent deep reinforcement learning models.

Inspired by this approach, weight normalization Salimans and Kingma [2016] is a method which reparameterizes the weight vectors in a neural network by decoupling the length of those weight vectors from their direction. By reparameterizing the weights in this way, the conditioning of the optimization problem and convergence of gradient descent is improved. Unlike batch normalization, this reparameterization does not introduce any dependencies between the examples in a mini-batch.

We utilized weight normalization to lessen the significance of our initialization procedure. This implementation vastly improved the learning process, by greatly speeding up convergence.

5.4 Recurrent Architecture

In Firoiu et al. [2017], the authors used two fully-connected hidden layers of size 128 for the neural network approximating the Q function. Inspired by choices made in Hausknecht and Stone [2017], to isolate the effects of recurrency, the architecture of the DQN was minimally modified, appending a recurrent GRU cell to the last fully-connected hidden layer, prior to the output layer, which outputs a Q-value for each action. With this update, learning under delay was stabilized.

5.5 Recurrent Updates

Again, inspired by Hausknecht and Stone [2017], bootstrapped random updates are performed. Episodes are selected randomly from the replay memory and updates begin at random points in the episode and proceed for a certain number of timesteps.

Sequential updates have the advantage of allowing the recurrent network to learn temporal dependencies, but violate the DQN's random sampling policy. In order to adhere to the policy, the GRU's hidden state is zeroed at the start of every update.

6 Performance Results

6.1 Quantitative

We trained our network at various amounts of delay against the built-in Level 9 (highest difficulty) CPU. (See Figure 3). The rewards plateauing earlier as delay increases is expected, due to the further separation of actions from rewards. However, more significantly, the training process of the recurrent network is stable, despite the increase in delay, which is a vast improvement on the attempt to condition on previous actions as done in Firoiu et al. [2017].

As generating experiences was a major bottleneck, many different emulators were run in parallel, typically 50 or more per experiment, sending their agent's experiences to the trainer. Under human-like delay, 12 frames, training took a little over 16k episodes to reach positive rewards. [1]

6.2 Qualitative

We viewed the agent trained under human-like delay play against the Level 9 CPU. Confirming our interpretation of positive rewards, the agent performed well against the built-in AI. It was prepared to react to the AI's attacks, and once sent off-stage, was able to recover adequately well. Furthermore, realizing that the AI was prone to approaching, the agent deployed slower, stronger attacks in order to increase damage and ultimately knock out (KO) the opponent by sending them out of bounds. It is possible, though, that the agent only performed slow attacks that took longer than the delay because it could more easily predict the result of the attacks.

Despite being highly effective against the CPU, this strategy does not translate well to human-level combat. As humans are much less predictable, the agent struggled. In order to rectify this, the agent would have to be trained against a fixed set of strong enemies.

6.3 Strong Enemies

The goal of using strong enemies was to discourage the agent from employing degenerate strategies, which would certainly not be optimal in human-level play. Ideally, the agent would generate experiences against a competent human player, however obtaining the sufficient amount of experience needed to learn through playing against one is practically infeasible.

Therefore, a compromise needs to be made between enemies. The agent should not train against weak enemies, as its unoptimal strategies would continue to be encouraged. However, the agent might not learn against the strongest of enemies, as the agent will be overpowered and not have the opportunity to learn.

Therefore, an iterative training process was conducted (See Figure 4). The delayed agents trained by the authors were surveyed, and were ranked based upon qualitative strength. It should be noted that

[1] Computing resources were provided by the Mass. Green High- Performance Computing Center.

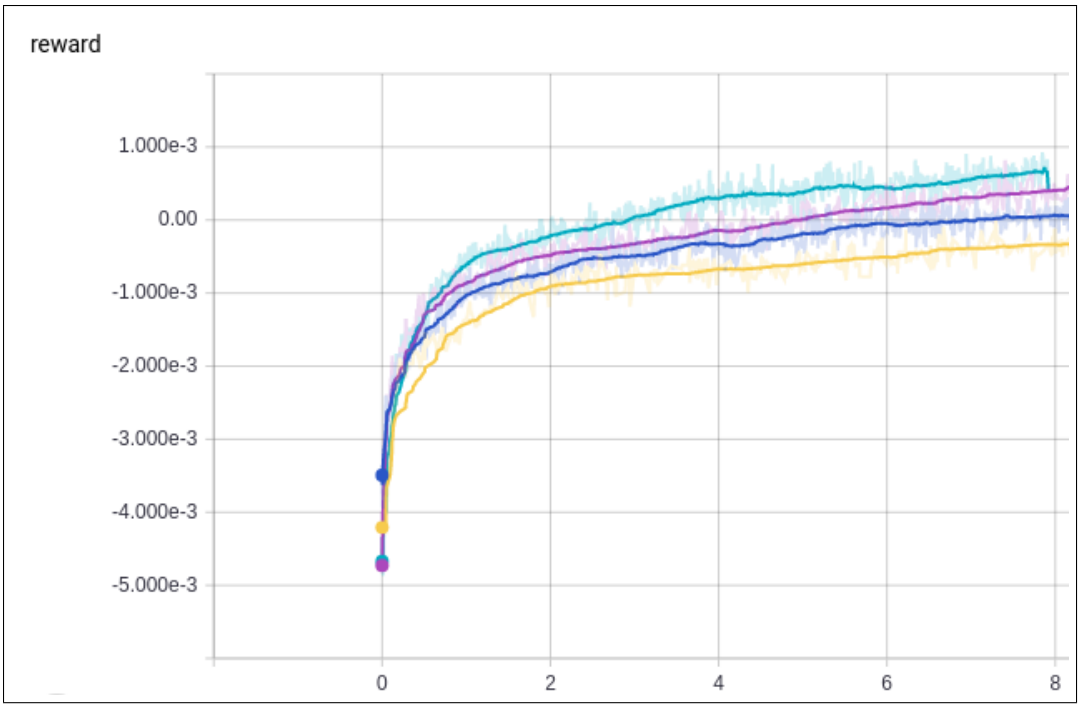

Figure 3: Comparison of Recurrent DQN trained under various amounts of delay against the level 9 CPU: 3 frames (light blue), 6 frames (purple), 9 frames (dark blue), and 12 frames (yellow). Plotting reward vs. training batches (in thousands).

these agents were only delayed by 2-4 frames. An agent playing the character "Peach" was deemed to be the weakest, thus the agent was trained from scratch for many iterations against that agent, until the rewards sufficiently plateaued. After that, we trained the agent against a relatively strong agent, playing the character "Marth", and the rate of reward increase was staggering.

However, after observing our agent play against the "stronger" agent, it was clear that the degenerate strategy of continuously delivering slow, powerful attacks was still being used. Surprisingly, even with this policy, the agent was able to hold its own against the "stronger" agent, who employed a more conventional competitive strategy.

It was found that for nearly all of the delayed agents trained by the authors, this low-level strategy worked, as the agents appeared to not have seen that strategy in their training experiences. Only the strongest agent, which was the AI capable of beating the "world's best" with its unrealistic reaction speed, was able to successfully counter this approach. However, this AI appeared to be too strong for the agent to train against.

Nonetheless, an attempt was made at training against that AI, and despite the large initial negative rewards, and the slow training, the reward continues to increase. Further work would involve building upon the output of this training process, in order to hopefully train a competitive recurrent agent. In addition, we would like to investigate whether it would be possible for the delayed agent to learn faster moves.

7 Discussion

Deep Reinforcement Learning has had widespread success in developing game-playing AIs. In Firoiu et al. [2017], a Super Smash Bros. Melee (SSBM) AI was developed which succeeded at beating the world's best in the competitive scene. We advanced this work by stabilizing the training of the agent under human-like delay. Through tuning a large set of Deep Recurrent Q-network (DRQN) hyperparameters, particularly the exploration vs exploitation trade-off parameters, the discount factor, the weight normalization method, and the recurrent architecture and update procedure, we were able to successfully train under human-like delay. A recurrent agent was able to achieve positive rewards

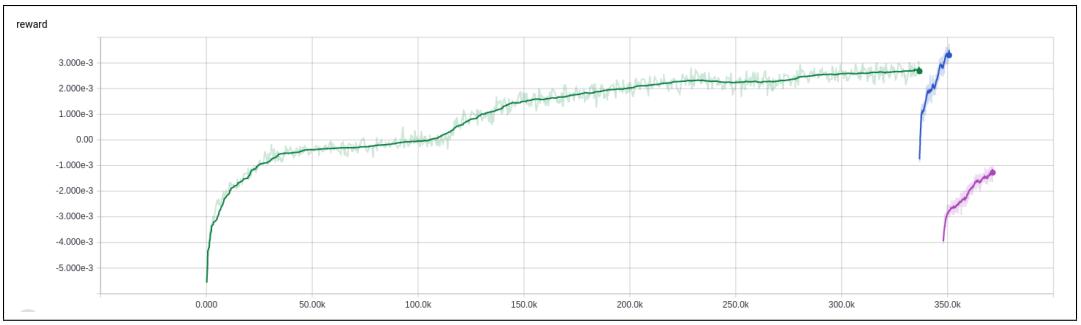

Figure 4: Iterative training process of RDQN against strong enemies: Weakest (green), Strong (blue), Strongest (purple) Plotting reward vs. iterations.

under human-like delay against the built-in AI, and when viewed, clearly succeeded in defeating the AI. However, the strategy learned when playing against the AI was not successful against competent human players. Further work would involve continuing the iterative training process against stronger enemies.

The iterative training approach was necessary for improving the agent due to the inability of the Q-learner to learn from self-play. Policy gradient methods were capable of learning from self-play, but we were unable to successfully introduce recurrence to the system. We believe this is due to the fact that the asynchronous experience generation training process violates the on-policy assumption of the REINFORCE rule, and the effects of this are only exacerbated when previous memories are stored. Resolving this issue and thus enabling training with self-play is an area of future work.

Furthermore, two other classes of approaches can be explored for resolving the action delay problem. A model-based RL approach might be helpful. The agent could learn to predict future observations, so that it would have a better sense of the environment at the time its delayed action would be executed. Another approach would be to learn from demonstrations by obtaining samples of human play from online communities, and training a model to accurately emulate their behavior.

Action Delay, and human-like play in general, is an issue that still has not been addressed by the deep RL community, and remains an interesting and challenging open problem. By stabilizing the training process for a recurrent agent in the SSBM environment, we hope to have contributed to the pursuit of a solution.

Acknowledgments

We would like to thank Christopher Curro for advising the project, and Vlad Firoiu for providing both computational resources and advice regarding the AI's implementation.

References

V. Firoiu, W.F. Whitney, and J.B. Tenenbaum. Beating the World's Best at Super Smash Bros. Melee with Deep Reinforcement Learning. 2017.

M. Hausknecht and P. Stone. Deep Recurrent Q-Learning for Partially Observable MDPs. 2017.

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. 1997.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ICML*, 2015.

Y. Li. Deep Reinforcement Learning: An Overview. 2017.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

T. Salimans and D.P. Kingma. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. 2016.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V an Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

D. Steckelmacher and P. Vrancx. An Empirical Comparison of Neural Architectures for Reinforcement Learning in Partially Observable Environments. 2015.

I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. *ICML*, pages 1139–1147, 2013.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.

G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.