Presentation: https://youtu.be/Mtt3y5O0ebw

Deep Beating: Using Deep Reinforcement Learning to Win at Street Fighter III

Stanford University

Johnson Jia johnjia@stanford.edu

Overview

- **Objective:** Train an agent using reinforcement learning to play Street Fighter III: 3rd Strike.
- **Deep Q-Learning:** One approach to training such an agent is to use a deep neural network to represent the Q-value function and train this neural network through Q-learning. See [1] for an implementation of such an agent.

Game Environment

- **MAME RL Toolkit:** The MAME RL Toolkit [2] provides a simulation environment to train an agent.
- The toolkit takes in a *move action* and an *attack action*, and return the next set of frames of the game state after executing the move and attack actions.

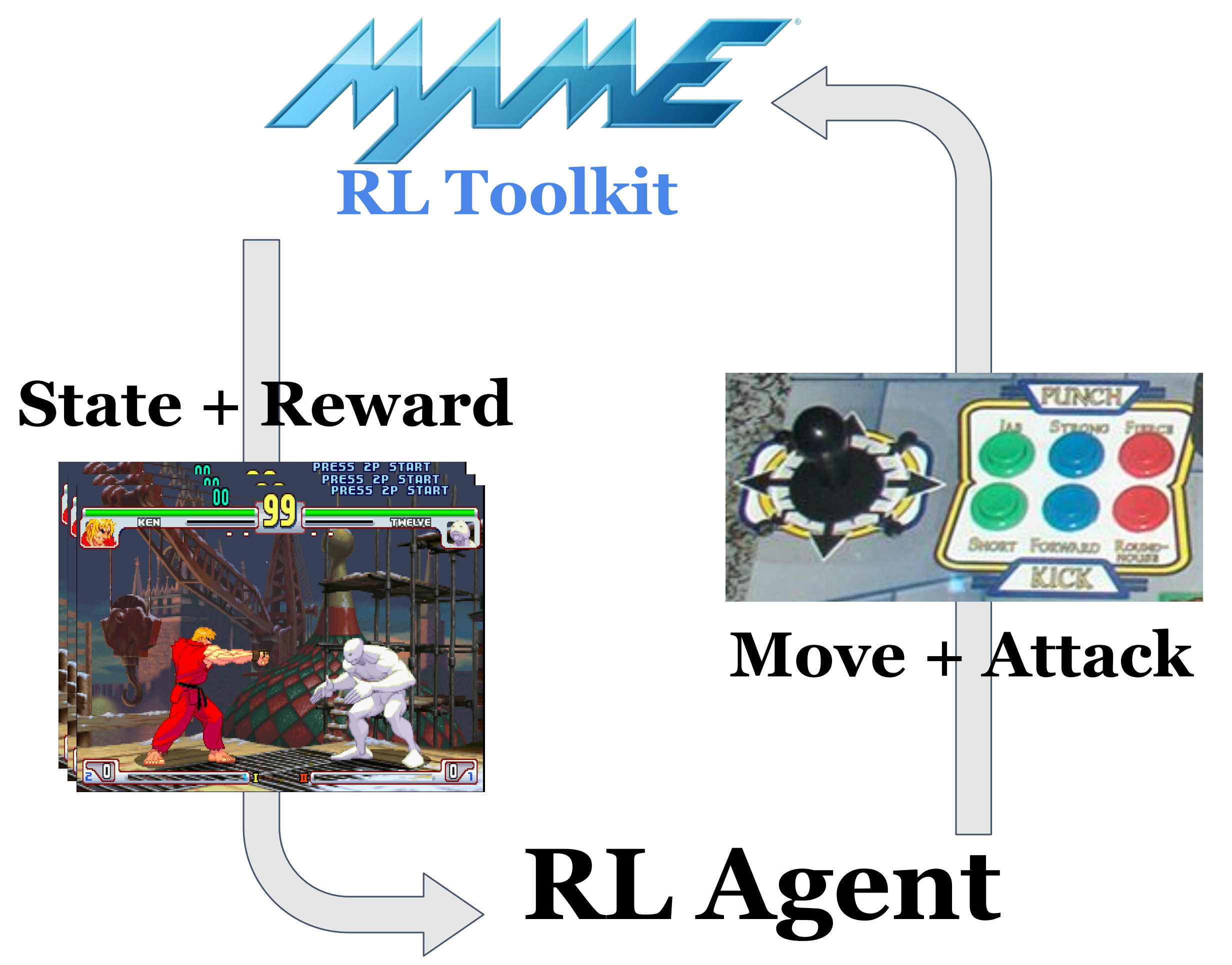

Figure 1. Agent takes frames and reward as input, update the Q-value function, and select the next move and attack actions.

CNN For the Q-Value Function

- **Model:** The model for the Q-value function is a convolutional neural network inspired by ResNet.

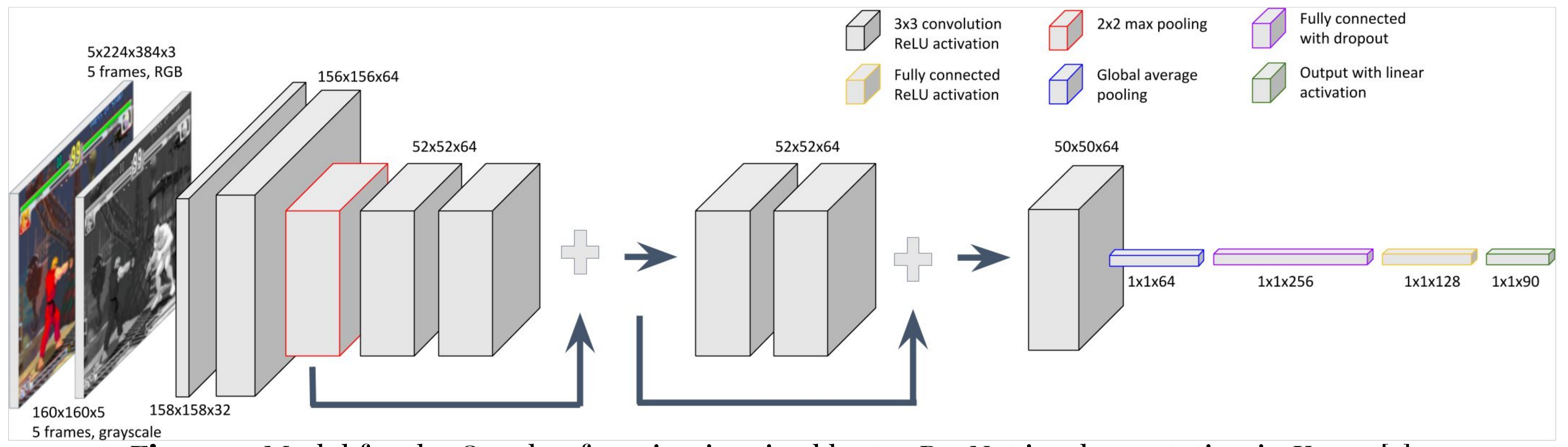

Figure 2. Model for the Q-value function inspired by toy ResNet implementation in Keras [3].

- The input to the CNN is a stack of 5 RGB frames, which are resized into 160x160 grayscale images. The output of the CNN are the predicted Q-values (or the *utilities*) for each of the 90 possible actions (9 move actions times 10 attack actions).

Learning Algorithm

- An ε-greedy Q-learning algorithm is implemented to train the agent. It consists of two phases: episode playout and experience replay.
- **Episode Playout:** We play out a game and select the agent's action using an ε-greedy strategy. We store a *memory* consisting of the current state, action chosen, the reward returned, the next state, and whether we reached an end state, namely, the end of a fight in a queue.
- **Experience Replay:** We randomly sample a subset of memories from the queue. For each piece of memory, we run the state through the CNN to get the estimated utilities of all possible actions, and update the estimated utility corresponding to the action in the memory using the reward. These utilities are then the labels used to train the CNN.

Implementation Details

- The algorithm is implemented in Python using Keras as the core modeling libraries.
- The training is performed on a Ubuntu Linux machine with 32GB of RAM and a NVIDIA GeForce 1080 Ti with 11GB of VRAM.

Results and Discussions

- We trained the agent over 160 games, which took around 4 days to train.

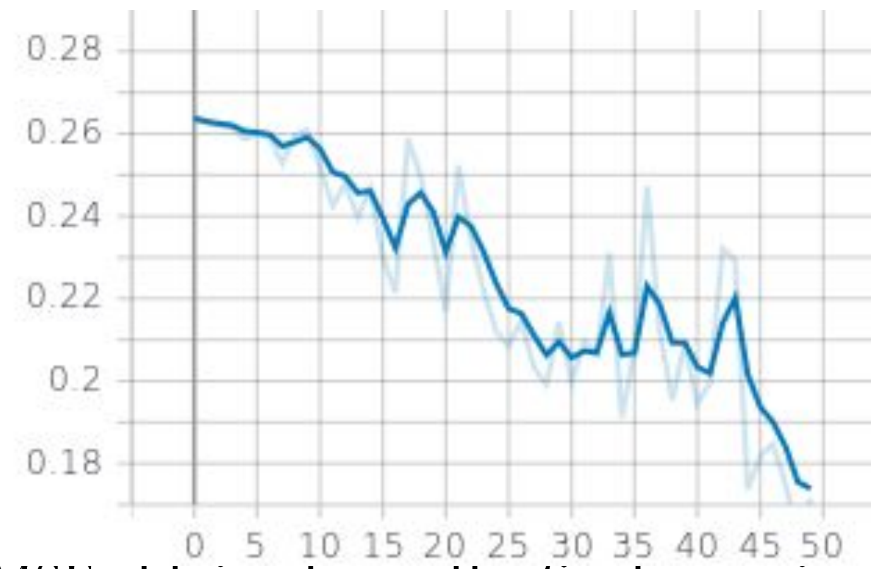

Figure 3. MSE obtained over the final experience replay.

- While we observed convergence in training the CNN, the agent trained appears to be stuck in a local minimum. It only selects a few actions repeatedly and on average is not better than an agent playing at random.
- At this point, we believe the value function may be too complex to learn in sufficient amount of time under the current setup. Also our approach of updating only one of the 90 predicted utilities at a time makes it too easy for the model to fall into a local minimum.

Future Work

- Tweak the architecture of the CNN to speed up inference (which has been a bottleneck in training).
- Use policy gradient methods (such as actor-critic) to directly train the agent to learn an optimal policy.
- Self-play: Let the agent play against itself to improve its performance.

References

[1] Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David & Graves, Alex & Antonoglou, Ioannis & Wierstra, Daan & Riedmiller, Martin. (2013). *Playing Atari with Deep Reinforcement Learning.*

[2] The MAME RL Algorithm Training Toolkit: https://github.com/M-J-Murray/MAMEToolkit

[3] The Keras functional API in TensorFlow - A Toy ResNet Model https://www.tensorflow.org/guide/keras/functional#a_toy_resnet_model