Deep Reinforcement Learning with Actor Critic Progress Report

Christopher M. Lamb
Department of Computer Science
University of California San Diego
San Diego, CA 92122
c2lamb@eng.ucsd.edu

Daniel Reznikov
Department of Computer Science
University of California San Diego
San Diego, CA 92122
drezniko@eng.ucsd.edu

Luke Liem
Department of Computer Science
University of California San Diego
San Diego, CA 92122
lliem@eng.ucsd.edu

Alexander Potapov
Department of Electrical and Computer Engineering
University of California San Diego
San Diego, CA 92122
apotapov@eng.ucsd.edu

Sean Kinzer
Department of Computer Science
University of California San Diego
San Diego, CA 92122
skinzer@eng.ucsd.edu

Christian Koguchi
Department of Electrical and Computer Engineering
University of California San Diego
San Diego, CA 92122
ckoguchi@eng.ucsd.edu

Abstract

Deep Reinforcement Learning (RL) is a powerful approach to agent learning in complex environments. Its applications range from gaming, to robotics to financial-market trading, achieving state-of-the art results across some distinctly challenging problem spaces. The aim of this work is to survey the actor-critic based techniques that have been applied to agent learning in Atari games [**?**]. We profile baseline approaches, and experiment with changes to network topologies and configurations as well as the asynchronous multi-agent learning described by Minh et al [**?**].

1 Introduction

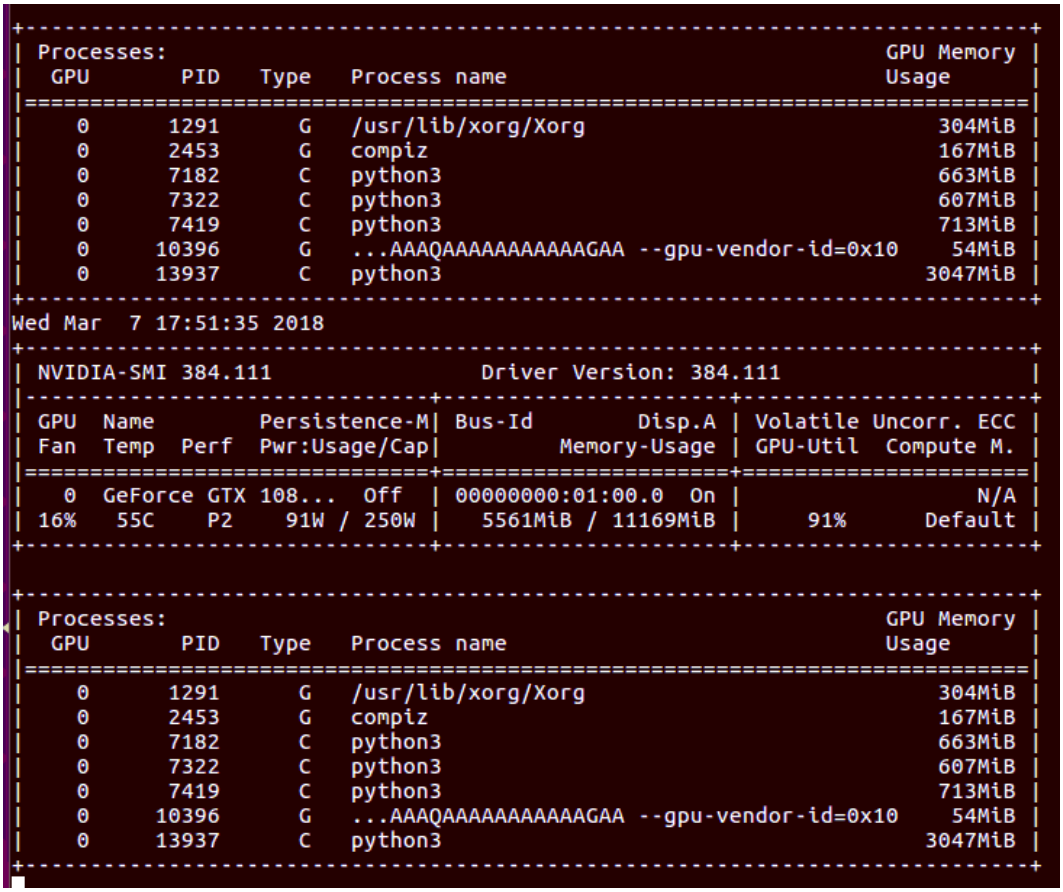
The actor-critic paradigm is simple and logically expressive. The actor's role is to engage in some (pre-determined) policy, and the critic is responsible for evaluating this policy. Through interactions with the environment (in our case game-play) and explicit rewards (points scored) both the actor and critic are able to iteratively update the representations of their tasks. Our goal is to profile a baseline approach which uses a single-agent advantage actor-critic model (where the actor is learning a policy through policy gradient [**?**] and the critic is learning a value function similar to classic Q-Learning [**?**]), and explores ways we can modify the network to achieve better performance. We also try to reimplement A3C [**?**] for asynchronous multi-agent play what has recently been shown to reduce

learning time drastically. In sum, our efforts represent a survey of the state-of-the art methods, which some experimentation do understand more deeply how network topologies (architectures and convolution vs linear approaches) affect both the ability of agents to learn and the training time.

2 Methodology

2.1 System

Our group is lucky to have access to a GPU machine with 4 core Intel i7 processors, 32GB of RAM, and a powerful Nvidia GTX-Geforce 1080-Ti GPU with 11GB of VRAM. We install OpenAiGym [**?**] to support Atari game-play and PyTorch [**?**] for neural network (NN) development.



2.2 Experiment Setup

We clear Cuda GPU cache to avoid biasing from data cache hits in profiling run-times experiments.

2.3 Metrics

To compare the performance across experiments, we measure *score_vs_episodes*. To understand who the experiment configuration affect training time, we also report on *time_per_episode*. For a comparison, we thought it would be interesting to include the performance of human players as reported by Minh et al [**?**]. The human score is given by the 75th percentile score achieved by an average of human experts.

2.4 Games

In order to profile a wide-range of difficulties 3 different games: Pong, Breakout, and Space Invaders. While the size of the action space varies from $[4, 20]$, the major differentiating factor for complexity comes from the delay in reward assignment. However, due to time constraints and frame-rate difficulties in Space Invaders, we were unable to effectively train a neural network for this particular task and wish to reduce scope to just Pong and Breakout while expanding on their depth.

Game Name	Difficulty Score	Category
Pong	2	Easy
Breakout	3	Easy
Space Invaders	5	Medium

Figure 1: Game Difficulties

OpenAIGym exposes two methods of game-play. RAM provides the memory state of the game, and RGB which only provides a RGB image of the screen, reflecting the pixel values that would be rendered for game-play. Each action provided by the agent is repeatedly performed for a duration of k frames where k is uniformly sampled from $\{2, 3, 4\}$. We exclusively use RGB mode, which provides our model with the same information that a human player would have.

3 Experiments

Model 1: 2-Layer Linear + Softmax + A2C

We first take PyTorch's Actor-Critic implementation for Cartpole as the starter code, and adapt it to play Pong. The end result is an agent built on PyTorch which barely beats the game AI (running reward of 1.0), significantly underperforming Karpathy's implementation. We also observe that the model performance has high variance across episodes as shown in Figure 2 below. We suspect this saturation in performance is a result of converging to a local minima.

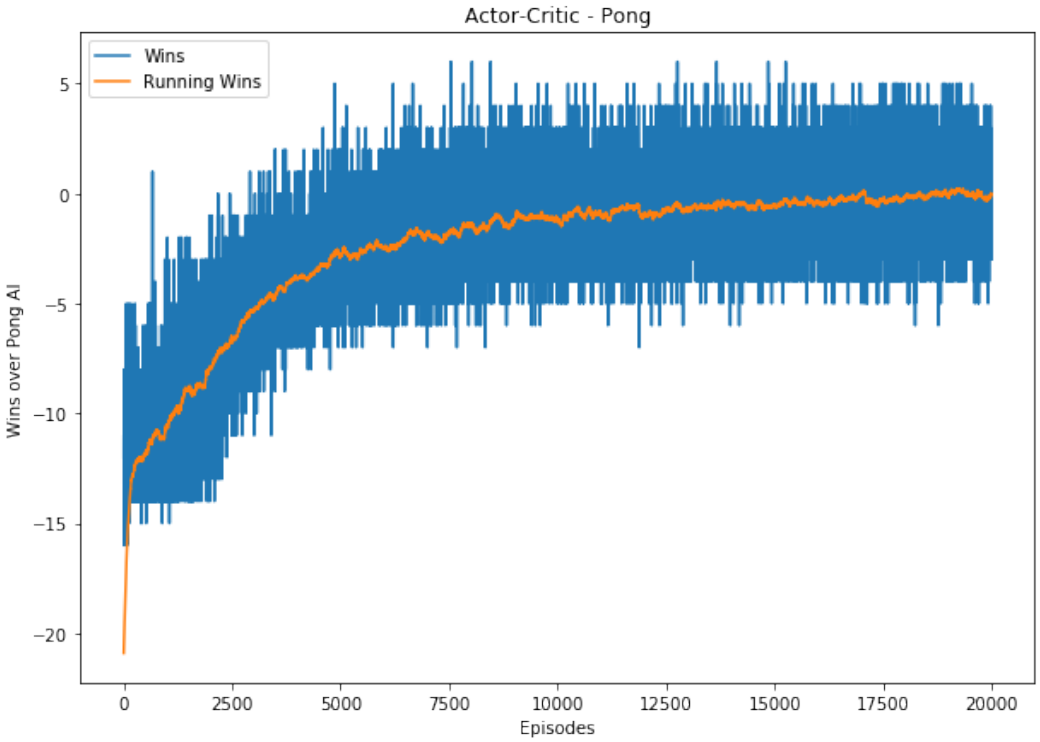


Figure 2: Model 1: 2-Layer Linear w/Softmax Activation + A2C

Model 2: CNN + A2C

Using a convolutional neural network using Actor-Critic (as shown in blue in Figure 3) and a temperature in the softmax classifier, we can solve the basic explore-exploit problem. We observe that is learns very quickly over the first thousand episodes but saturates around 12 points. This exceeds the human-expert by 2 points (shown in orange in Figure 3).

The architecture of the neural network is shown below in python code:

```
# 4x3x80x80 input # Stack of 4 Atari game frames

Policy(
  (features): Sequential(
    (0): Conv2d (4, 16, kernel_size=8, stride=4, padding=2)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True)
    (2): ReLU(inplace)
    (3): Conv2d (16, 32, kernel_size=4, stride=2, padding=1)
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
    (5): ReLU(inplace)
```

```
  )
  (action_head): Linear(in_features=256, out_features=6)
  (value_head): Linear(in_features=256, out_features=1)
)

Action = softmax(action_head(x)/temperature)
Value = value_head(x)
```

Model 3: CNN + LSTM + A2C

For the next, model we wish to introduce LSTM units into the model. For the simple CNN model alone, we can only effectively capture information that is 4 frames deep, which may be enough to recognize where the ball and paddles are moving in the case of Pong. However, using LSTM units introduces longer memory into the mix and can recognize patterns of higher complexity across time depending on the number of time steps and hidden units we use. In simple games such as Pong, the advantage of LSTM units may not be as obvious. However, in games such as Breakout, it may be able to recognize that the optimal strategy is to break out of the tunnel. Or in the case of Space Invaders, it may remember that the game speeds up over time, developing strategies for early-game and late-game.

In Figure 3, we superimpose the rewards for a CNN alone and a CNN with LSTM units. We observe that the CNN learns faster but plateaus sooner at 12 while the LSTM model learns slower but will plateau at a higher running reward of 16.

The architecture of the Neural Network with LSTMs is shown below:

```
# 4x3x80x80 input # Stack of 4 Atari game frame

LSTM Policy(
  (features): Sequential(
    (0): Conv2d (4, 16, kernel_size=8, stride=4, padding=2)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True)
    (2): ReLU(inplace)
    (3): Conv2d (16, 32, kernel_size=4, stride=2, padding=1)
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
    (5): ReLU(inplace)
    (6): Conv2d (32, 32, kernel_size=4, stride=2)
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
    (8): ReLU(inplace)
  )
  (lstm): LSTMCell(512, 256)
  (action_head): Linear(in_features=256, out_features=6)
  (value_head): Linear(in_features=256, out_features=1)
)

Action = softmax(action_head(x)/temperature)
Value = value_head(x)
```

We observe a large decrease in performance after around 10,000 episodes using LSTM units. We suspect that this is possibly due to the exploding gradient problem. We wish to verify and mitigate the effects of this problem next.

Figure 3: Model 2 and 3: Reward Results for CNN and CNN + LSTM

Model 4: CNN + LSTM + A2C

To mitigate the problems with possibly exploding gradient, we attempt to use a k-step truncated backpropagation through time to train the models. We cross-validate across different values of k and plot these results in Figure 4.

We notice that we no longer get these large drops caused by the exploding gradient as shown in blue (refer to Figure 4). We see that LSTM with 512-time steps (shown in green) trains about as quickly as simply using CNNs alone (purple) from the previous model (model 2). However, using LSTM with higher chunks results in slower training times with perceived higher performance as determined by their slopes in the graphs. More specifically, as we see in the plot in red, LSTM with 768 time steps trains more slowly but does not appear to saturate after 10,000 episodes and appears that it will outperform all the other models (Figure 4). All of our models outperform human-expert performance as shown in brown.

In the plot below, we also show the results for the game of Breakout. We observe that we can beat human-expert levels of breakout by around 100,000 episodes using the convolutional network without LSTM units in model 2 (refer to Figure 5). We have running results for the networks with LSTM units as well and suspect they will not saturate as early as the CNN model (as shown in blue and orange in Figure 5). We wish to explore this further to verify if LSTM units will outbeat human-expert performance and the pure-CNN model.

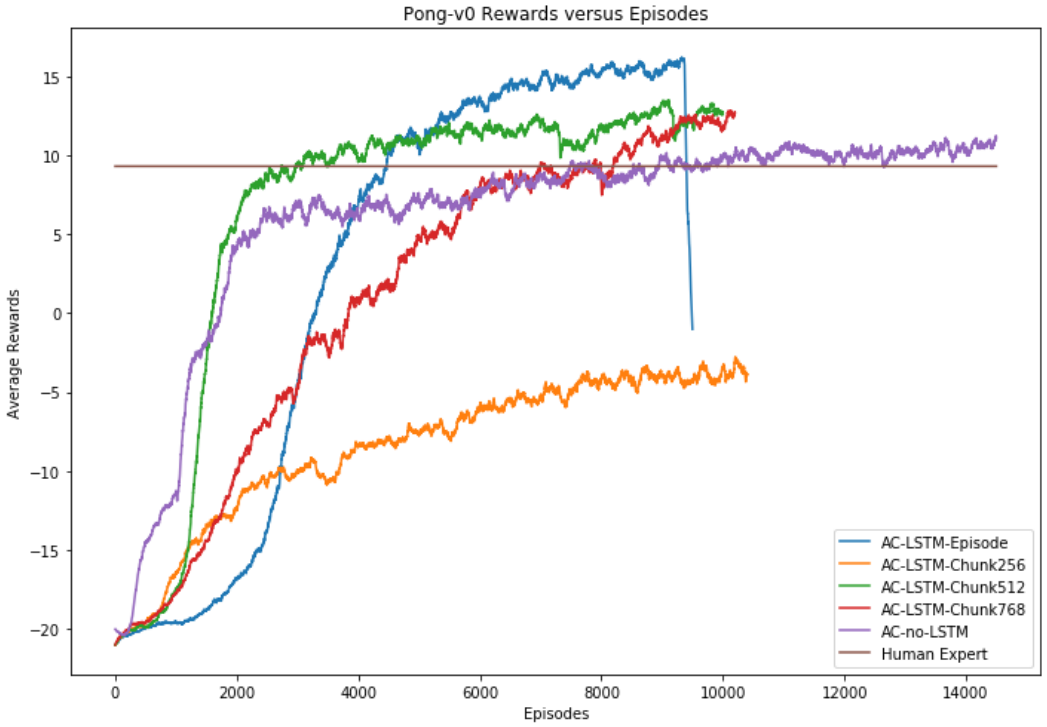


Figure 4: Model 4: Reward Results for Pong - LSTM with various Chunk sizes

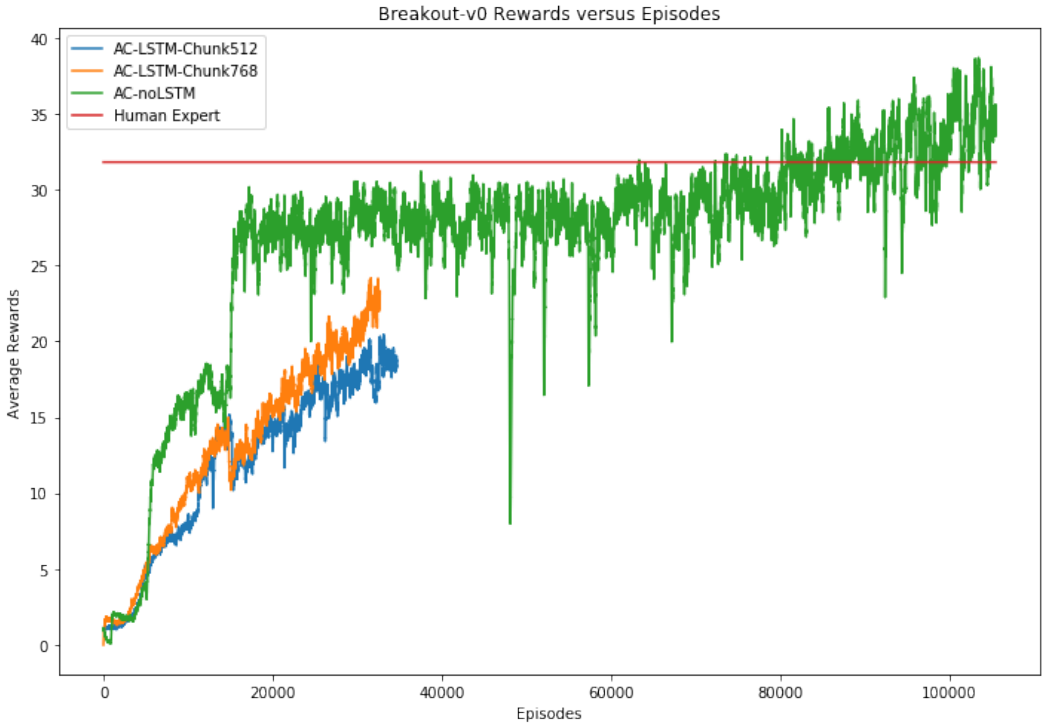


Figure 5: Model 4: Reward Results for Breakout - LSTM

Space Invaders

We discovered that due to the sampling rate of Space Invaders in AI Gym, the lasers from the enemies do not show up. This prevents our models from learning and preliminary plots are shown

below in Figure 6. Due to this issue, we wish to stop further development for this game in favor of deepening our efforts in the first two games (Pong and Breakout) due to time constraints.

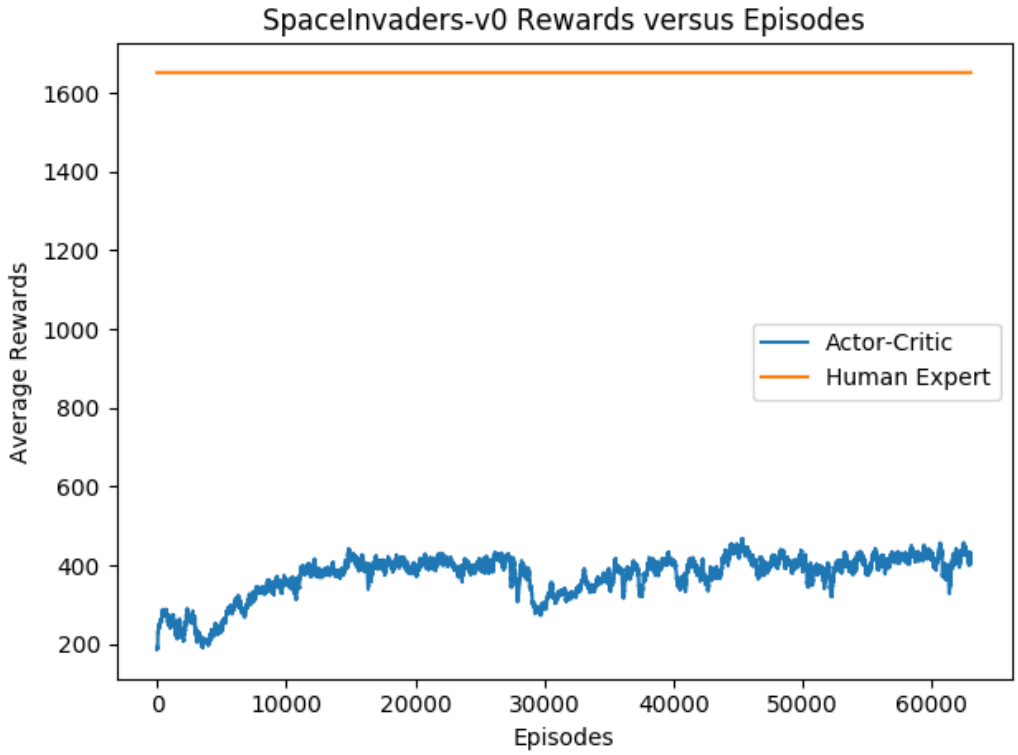


Figure 6: A3C Benchmark for Breakout

A3C Benchmark

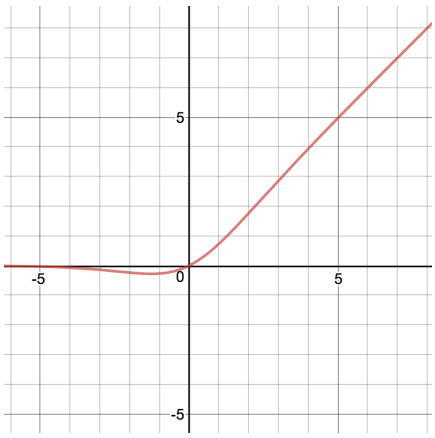
Our A3C benchmark uses existing starter code and beats human experts very easily as shown below in Figure 7 for Pong and Breakout respectively. It is very important to note that the 4-agent and 16-agent ones converge and win 21-0 after around 150 and 300 episodes respectively. For Breakout, there isn't an upper limit for how much better an AI could beat a human-actor which is illustrated in the green line

Figure 7: A3C Benchmark for Pong and Breakout

Future Works

Option 1

A new activation function, "Swish" and 2 Convolutional Layers.
The function for "Swish" is $f(x) = \frac{x}{1+e^{-x}}$ and its plot is shown below:




```
# 4x3x80x80 input # Stack of 4 Atari game frames

LSTM Swish(
  (features): Sequential(
    (0): Conv2d (4, 16, kernel_size=8, stride=4, padding=2)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True)
    (2): Swish(inplace)
    (3): Conv2d (16, 32, kernel_size=4, stride=2, padding=1)
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
    (5): Swish(inplace)
  )
  (action_head): Linear(in_features=256, out_features=6)
  (value_head): Linear(in_features=256, out_features=1)
)

Action = softmax(action_head(x)/temperature)
Value = value_head(x)
```

Option 2

Four Convolutional Layers with Swish activation function and LSTM units.

```
# 4x3x80x80 input # Stack of 4 Atari game frames

USASwish(
  (features): Sequential(
    (0): Conv2d (4, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
    (2): Swish(inplace)
    (3): Conv2d (32, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
    (5): Swish(inplace)
    (6): Conv2d (32, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
```

```
    (8): Swish(inplace)
    (9): Conv2d(32, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (10): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True)
    (11): Swish(inplace)
  )
  (action_head): Linear(in_features=256, out_features=6)
  (value_head): Linear(in_features=256, out_features=1)
  (lstm): LSTMCell(800, 256)
)
```

Option 3

Explore using entropy rather than temperature for policy exploration/exploitation.

Option 4

Truncated BPTT through a full episode to see what time step results in best performance.

Option 5

Re-run A3C with standardized metrics.