Learning Atari: An Exploration of the A3C Reinforcement Learning Method

Tyler Chesebro and Aleks Kamko

December 15, 2016

Abstract

We experiment with various Deep Learning inspired methods for playing an Atari 2600 emulator and attempt to improve on the state-of-the-art. We begin with a Deep Q-Learning Network as a baseline model but are not able to get promising results. We move on to the more recent Asynchronous Advantage Actor-Critic (A3C) model and have more success. We take A3C further by experimenting with adding an LSTM layer, with incorporating Generalized Advantage Estimation, and with adding a Spatial Soft Argmax layer. The first two additions prove to be quite beneficial in speeding training time and convergence. Our final implementation uses A3C with LSTM layer and Generalized Advantage Estimation and fully trains the model on the game Pong after 1.5 million episodes.

1 Introduction and Problem Statement

In 2013, DeepMind published the seminal paper *Playing Atari with Deep Reinforcement Learning* [2] in which they invent Deep Q-Learning, "the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning" They used this model to play seven Atari 2600 games directly from the screen pixels of an Atari emulator using the Arcade Learning Environment platform [2]. For our project, we implement DeepMind's original Deep Q-Learning model as a baseline and compare it against the state-of-the-art model: Asynchronous Advantage Actor-Critic (A3C), [3] also invented by Mnih. et. al. We iterate on top of the original A3C model by experimenting with an LSTM layer, with Generalized Advantage Estimation [4], and with a Spatial Soft Argmax layer [1].

At a very high-level, our challenge was to produce an algorithm that was able to play basic Atari video games with an input of only the screen display and an output of a single action (button) every few frames. This means that we are solving a Reinforcement Learning (RL) problem, but rather than using predefined features as in a traditional theory-driven approach, we take a data-driven approach by using a convolutional neural network (CNN) to learn and extract features directly from the game screen pixels.

Using a CNN to determine features for a reinforcement learning problem represents a significant step for RL for two reasons. First, implementing an "automatic" method for extracting features frees the programmer from nuanced manual feature design. Second, a single well-tuned CNN design can be retrained across several similar environments with different goals and output configurations, e.g. the same hyperparameters can be used for both Pong and Space Invaders.

Reinforcement Learning also presents several challenges to Deep Learning as well. Data sets used for deep neural networks (DNNs) are traditionally composed of large sets of hand labeled samples, and it is usually assumed that there is a constant underlying probability distribution within the data. These data sets also provide immediate feedback to the accuracy of a given prediction for every sample in the set. RL algorithms instead receive sparse rewards and any agent must be able to predict whether a given action will result in a positive outcome, sometimes many time steps into the future. Additionally, most Deep Learning

assumes that each point in a data set is independent, where as RL should consider each data point to be correlated with those near it. [2]

The high level metric we will use to evaluate our results is a measure of average reward per episode. As with most reinforcement problems, a good model should increase over the training time. Because we are using neural networks to model a policy rather than a classifier or generator, there is no dataset and there for test or cross validation sets involved in the problem.

Deep Reinforcement learning is a relatively new concept, yet is still of great interest to the fields of Artificial intelligence and Robotics. Neural networks have the possibility to be better models for policy algorithms than any human-designed implementations. Additionally, training the entire network together including convolutions means that there is no need to hand-designed features or that robots can learn policy just from visual input and a reward function. Greater reseach is this topic is carried out by Sergey Levine here at UC Berkeley in his paper "End-to-End training of Deep Visuomotor Policies" [1]

2 Data Source and Preprocessing

2.1 Tensorflow

We ran our models with Google's Tensorflow machine learning library. Tensorflow is an open-source library of numerical computation and machine learning. The main challenge that we have faced so far with the library is defining non-standard loss functions and label masks as Tensorflow objects. Or really getting the tensors to flow in general.

2.2 OpenAI Gym

We are using OpenAI Gym, an open-source reinforcement learning environment provided by OpenAI. OpenAI Gym provides a unified interface for many RL and video game learning challenges, including classic RL problems like CartPole, fully playable classic Atari-video games, and even an entire DOOM emulator. The API steps through the game and provides an observation of the pixels of the screen at every step. It accepts an action from an agent, executes the action, and returns the reward and next observation. For the Atari video game emulators, all observations are a 210x160x3 (RGB) frame of the pixels displayed. Gym accepts between 2 and 6 possible actions (button presses) as input, such as right, left, or "shoot". Gym will generate a positive reward when the algorithm executes an action that brings the game closer to a win and a negative reward that brings the game closer to a loss or a premature (meaning before win) termination of the game. When an action does neither, the environment returns a reward of zero. All of this is already implemented and automatically handled by the Gym API.

2.3 Amazon Web Services EC2

Were ran our DQN and initial A3C models on a g2.2xlarge EC2 instance. This instance is equipped with 1 Nvidia GRID K520 GPU, 8 vCPUs, and 15GiB of memory. We achieved reasonable performance on these machines but were still training for 24+ hours.

Our final A3C model (with Generalized Advantage Estimation) was trained on a m4.4xlarge instance. This instance is equipped with no GPUs, 16 vCPUs, and 60GiB of memory. With our updated model (taking cues from OpenAI's universe-starter-agent, elaborated below), we were able to run our models an order of magnitude faster on these machines because we were able to run more threads in parallel.

2.4 Frame Preprocessing

To reduce memory consumption and the computational cost of our convolution layers, we pre-process each Atari emulator frame by:

1. Converting to grayscale by taking the pixel average across the RGB channels
2. Cropping out a square of the center, where most of the important features are located
3. Compressing down to 42x42 pixels

In short, we compress each frame from 210x160x3 $\rightarrow$ 42x42x1. We have empirically noticed that the last step of our preprocessing in particular sped up our algorithm by nearly 3x.

Finally, for our DQN baseline model, we only input a new action on every 4th frame, repeating our previous action for the next 3 frames, as suggested in [2]. This has the effect of "speeding up" training by 4x since the agent can finish episodes 4x faster.

3 OpenAI's `universe-starter-agent`

Just last week (December 5, 2016), OpenAI open-sourced their own implementation of a Tensorflow-based A3C agent on Github at `github.com/openai/universe-starter-agent`. Coincidentally, universe-starter-agent agent explores some of the same improvements that we were exploring for this project.

universe-starter-agent performs remarkably well: on a 16 core machine, the A3C model with 16 threads can solve Pong in 30 minutes on average! Our first iteration of A3C was only able to get close to solving pong in ≈ 3 days. Naturally, we have since incorporated some of their techniques into both our baseline model and our final model. The most notable changes are as follows:

1. **Resizing down to 42x42.**

   The seminal DeepMind models only size down to 84x84 [3] [4]. We have observed empirically that further resizing does not have a significant impact on convergence time while speeding up computation considerably.

2. **Adding more convolutions with smaller filters.**

   DeepMind A3C agent: Conv(filters=[16], size=[8,8], stride=4) $\rightarrow$ Conv(filters=[32], size=[4,4], stride=2)
   universe-starter-agent: 4 x [Conv(filters=[32], size=[3,3], stride=2)]

   We hypothesize that the latter model performs better because (1) it has many more parameters, and (2) the smaller filter sizes allow for much faster convolution.

3. **Using ELU non-linearities between layers instead of ReLU**

   [5] explores the performance of ELU vs ReLU non-linearities and concludes that ELU performs better in most cases for a variety of reasons.

4. **Increasing sync-with-global steps from 5 to 20.** (A3C only)

   In the DeepMind A3C model, each agent thread applies its gradients upstream to the global model every 5 steps. Conversely, universe-starter-agent threads only apply gradients upstream every 20 steps.

   On one hand, the latter approach "slows down" learning because the global parameters are updated less often. However, longer intervals between updates have the effect of lowering the variance of updates, which empirically leads to a more stable algorithm.

4 Baseline Model: Deep Q-Learning

Our baseline model aims to reproduce the Deep Q-Learning Network (DQN) model described in [2]. The DQN combines the concept of Q-learning with a CNN.

In a standard Q-learning model, the Q value of each state, or the total amount of reward that can be achieved from a state by following an optimal policy, is represented as a Q-matrix with an entry for each

state. The Q-value of each state is learned as the algorithm explores the environment. Unfortunately, for Atari games, a Q-matrix is unfeasible: our matrix would have to be of size $2^{(210*160*3)}$ to hold all of the states – more than the number of atoms in the universe.

The DQN avoids this issue by combining the concept of Q-learning with a CNN. In particular, the convolutional network acts as a function representation of the Q-matrix, drastically reducing the required parameter space.

4.1 Replay Memory

The DQN learns "offline" by performing gradient descent on randomly sampled mini-batches of a stored replay memory. At each step, we store (s_t, r_t, a_t, s_{t+1}) in memory for training. We store up to 1 million previous states, though this limit is imposed by the memory of the machine we used, not the algorithm. The stochasticity of the random mini-batch keeps the model from over fitting to the most recently visited states while also preserving the probability of updating based off more commonly occurring states.

4.2 ε-greedy Exploration

During training, we use an ε-greedy exploration policy to encourage exploration of all states. The model chooses the action with the greatest Q-value from given from a feed-forward pass of the current state through the network with a probability ε, otherwise it chooses randomly from the set of valid actions.

4.3 Network Architecture

We diverge from the network described in [2] by taking hints from universe-starter-agent. We use 4 convolution layers with filter size [3, 3] and stride 2, followed by a fully-connected layer of 256 units, and ending in a fully-connected layer with units equal to the the number of output controls for a given Atari game (for Pong this is 3: "noop", up, and down). We place ELUs between all layers except between the last two fully connected layers.

4.4 Gradient Update and Loss

We Computed the loss as the model as the square loss of the expected reward and expected reward approximation of the Q function, Equation 1. Where the expected loss is updated from the previous iteration of the network weights in Equation 2

$$L_i(\theta_i) = \mathbf{E}_{s,a\ p(.)}[(y_i - Q(s,a;\theta_i))^2] \tag{1}$$

$$y_j = \begin{cases} r_j & \text{if terminal state } s_j \\ r_j + \gamma Q(s_{j=1}, a; \theta) & \text{for non-terminal } s_j \end{cases} \tag{2}$$

The difference from y_i and $Q(s,a;\theta_i)$ comes from the use of a separate target and training network. The target network is the set of weights from before the most recent gradient update. This in addition to the reward received is used to approximate the Q-function and give the objective result y_j from which we compute the loss of the newly update weights and compute the next gradient. The resulting update gradient for a given timestep at s_t is given in equation 3

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbf{E}_{s,a\ p(.)}[(r + \gamma \max_{a'} Q(s_{t+1}, a_{t+1}; \theta_{i-1}) - Q(s_t, a_t; \theta_i)) \nabla_{\theta_i} Q(s_t, a_t; \theta_i) \tag{3}$$

The full pseudo-code for the algorithm is given in Figure 2

DQN Data Flow

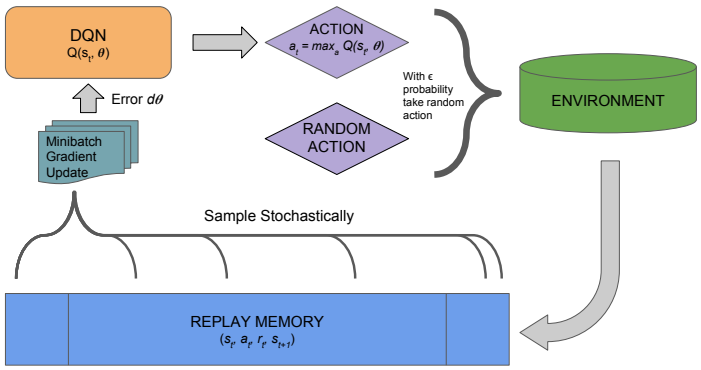

Figure 1: Graphical flow of a single N-step DQN.

DQN Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2: Pseudo Code for the training a Single DQN worker. Source [2]

A3C Algorithm

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t+1$
    $T \leftarrow T+1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t,\theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1,\ldots,t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial\,(R - V(s_i;\theta'_v))^2/\partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 3: The pseudo-code for a single actor-critic thread in A3C [3]

5 Asynchronous Advantage Actor-Critic (A3C)

For our final model, we implement and improve upon the state-of-the-art model from DeepMind's "Asynchronous Methods for Deep Reinforcement Learning" [3]. This seminal paper introduces the Asynchronous Advantage Actor-Critic (A3C) model which has several distinct differences from our baseline DQN model. A3C uses a deep neural network to model both a policy network $\pi(a_t|s_t;\theta)$ and a value network $V(s_t;\theta)$. For a given state s_t, the policy network (the "actor") predicts the optimal action to take at s_t while value network (the "critic") approximates the future reward from taking the optimal action at s_t. In theory, these two networks are separate, but in practice, we use the same convolutional layers for both the policy and value networks with separate output layers at the end.

The "Asynchronous" part of A3C means that multiple actor-critic threads are running at the same time, each with their own environment. Each thread steps through its environment with its own local CNN, periodically updating a globally shared CNN (all networks have an identical architecture.) For each thread, at every t_{max} local steps or when a terminal state is reached, that thread syncs its local parameters with the global parameters, computes gradients, and applies them upstream to the global network.

In contrast to the baseline DQN model, A3C learns "online" using a policy gradient method, directly from the states as they are processed by each thread. develops naturally as each thread runs its stochastic environment, so updates to the global parameters are implicitly uncorrelated. This suggests that A3C does not overfit to any particular state trajectory, and can therefore forego using a replay memory.

Our initial implementation of A3C roughly followed the pseudo-code in Figure 3.

We calculate the Advantage function $A(s_t,a_t;\theta,\theta_V)$ to be the discounted future expected rewards accumulated to t_{max} or a terminal state:

$$A(s_t,a_t;\theta,\theta_V) = \sum_{i=0}^{k-1} \gamma^i r_{t+1} + \gamma^k V(s_{t+k};\theta_V) - V(s_t;\theta_V) \tag{4}$$

Actor-Critic Work Flow

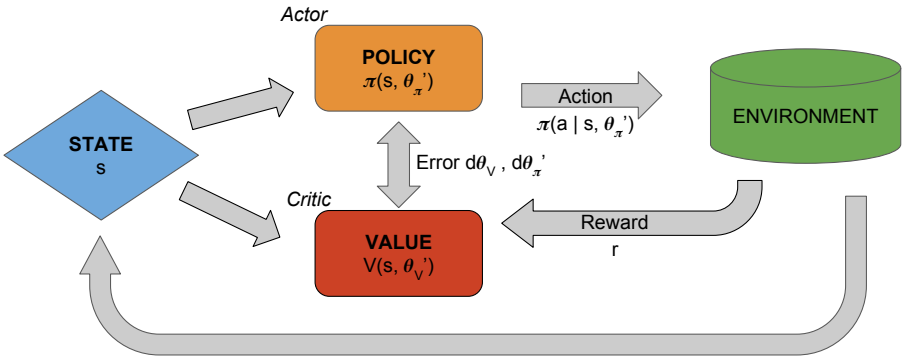

Figure 4: Graphical flow chart of a single Actor Critic worker

Gradients to the policy and value network are given by 5 and 6, respectively, summing gradients over all of the states in the past t_{max} local iterations.

$$\nabla_{\theta'} log\pi(a_t|s_t;\theta)A(s_t,a_t;\theta,\theta_V) \tag{5}$$

$$d\theta := d\theta + \partial(R - V(s_t;\theta'_V))^2/\partial\theta' \tag{6}$$

6 Extensions to A3C

For both our versions of the A3C model, we added an LSTM layer after the last fully connected layer. This gave us *considerable* improvement, shown below in the Results section.

For our best performing model, we implemented Generalized Advantage Estimation (GAE) from the paper "High-Dimensional Continuous Control Using Generalized Advantage Estimation" [4]. This algorithm combines with A3C well as it tries to improve the training time for policy gradient based RL problems. The algorithm is a modification of the reward policy using equations 7 8

$$\delta_t^V = r_t + \gamma V(s_{t+1}) = V(s_t) \tag{7}$$

$$A_t^{GAE} := \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+1}^V \tag{8}$$

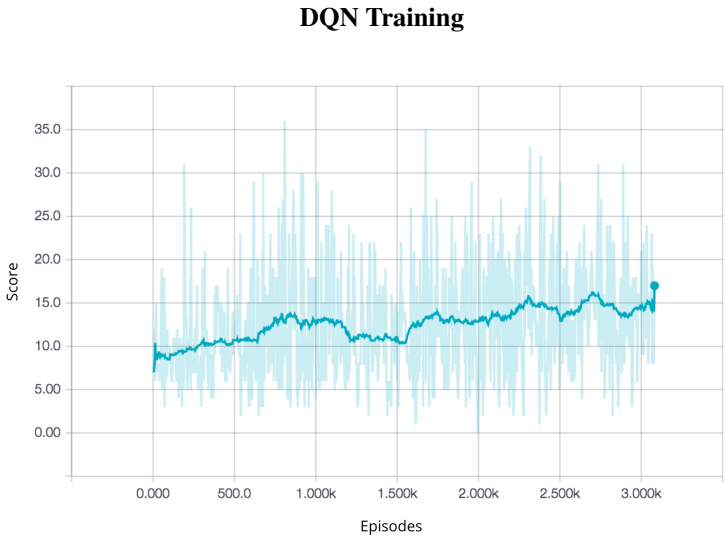

Figure 5: The training results of our initial DQN implementation. The algorithm trained on six thousand steps over the time period of 24 hours, resulting in a 5 point increase in the average reward. The algorithm trained in the Space-Invaders Environment

Where λ is an additional discount factor applied after reward shaping from γ. Taking inspirations from the paper [4], we choose values for λ in [0.96, 0.98, 0.99, 1.00].

Finally, we briefly experimented with adding a Spatial Soft Argmax layer [1] at the end of our convolutional layers instead of using an ELU non-linearity. Unfortunately, this didn't pan out as well as we'd hoped, which we document in the Results section below.

7 Results

All of our models were trained on Pong except for the DQN, which was trained using Space Invaders.

7.1 DQN Results

Our initial DQN was very slow and did not fully train as we ran out of time and resources. A plot of our training score on Space Invaders is presented in Figure 5. We hypothesize that our slow training may be due to a bug in our implementation, or maybe we simply needed to run it for much longer to see results comparable to [2].

7.2 Initial A3C Results

Our initial results for A3C are presented in Figure 6. These are the results we presented at our last presentation in the class. This version of our model did not incorporate the new changes from the universe-starter-agent. In particular, this model used only 2 convolutional layers instead of 4, used ReLU instead of ELU non-linearities, only downsampled images to 84x84, and updated the global parameters every 5 iterations instead of every 20. These hyperparameters follow those laid out by DeepMind in [3]. (However, we still incorporated an LSTM layer in this model.)

The results were promising, showing much improvement over our DQN implementation, but it took **4 days** to train vs. the $\approx$ 1 hour training times we see with our latest implementation. Additionally, we noticed

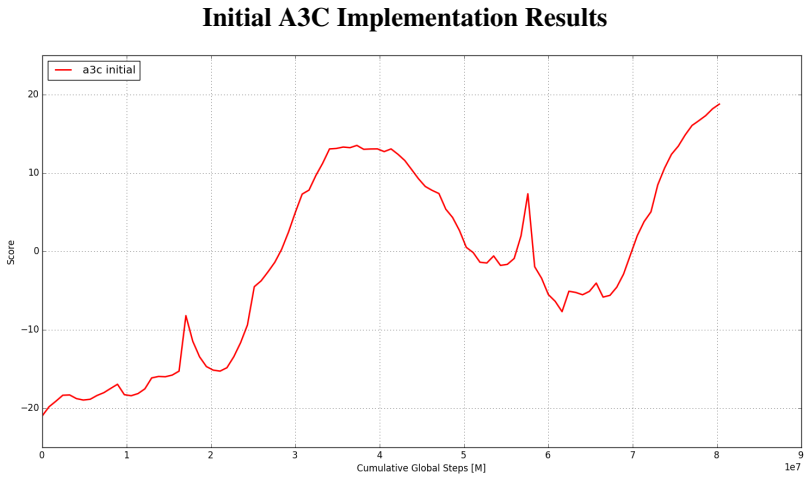

Figure 6: Rewards per episode for training with our initial implementation of A3C. This model trained at a rate of 80 Million episodes over 4 days

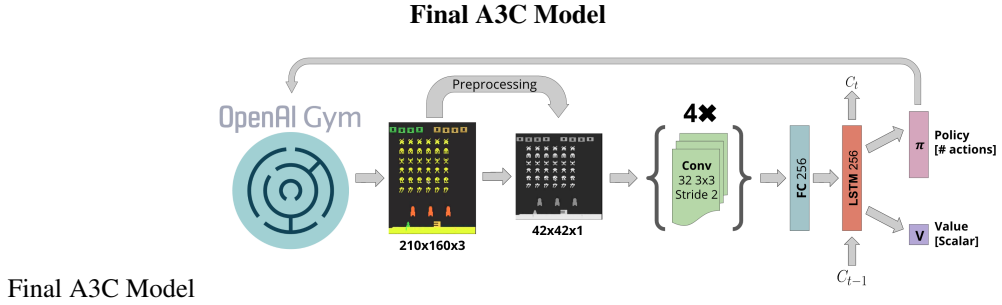

Figure 7: A summary of our final A3C architecture.

a small bug in our implementation after the fact: we weren't resetting the cell state and hidden state to our LSTM layer between episodes, causing strange correlation issues between episodes.

8 Final A3C Results

The results from here forward were obtained by running and tweaking our final version of A3C, summarized in Figure 7, in which we took cues from the universe-starter-agent model described in Section 3. This model trained much faster than our initial LSTM, down from 24 hours to $\approx$ 1 hour. Consequently, we were able to pump out results much faster; however, we lack a bit of data comparing our new A3C model vs our old A3C model (since our old model takes a long time to run).

8.1 GAE Results

To perform analysis on what values for GAE performed best we tested the algorithm using λ values [0.96, 0.98, 0.99, 1.00], Figure 8. The results were collected from training the model with 16 parallel workers for two distinct runs, per value of λ, and averaging results. We also compared our best value using GAE against a model without GAE using the simple advantage estimate described in Section 5, shown in Figure 9. Our experiments suggest that GAE is definitely an improvement over the simpler advantage estimator given a good choice of λ, and also that the optimal value for λ was roughly 0.99.

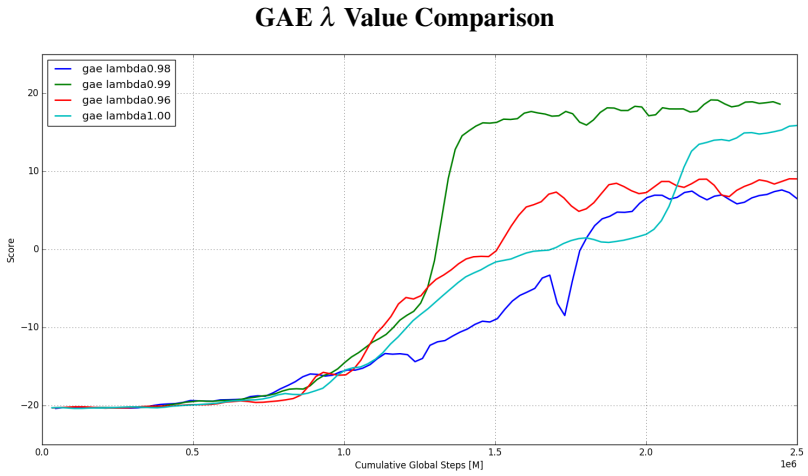

Figure 8: The training speeds of the giving λ values for our GAE implementation of A3C. Trained on Pong.

We also noticed (not shown) that a few of the the first and second runs for a given λ performed very differently. We think this is due to the highly stochastic starting environments of each training session.

8.2 LSTM v.s. Simple Feed Forward

We choose to test how important the final LSTM layer was to the performance of the A3C algorithm as a whole. We normally place an LSTM layer after the final fully-connected layer and before the output layers. For this benchmark we experimented with and without that last LSTM layer, using our best A3C with GAE model from Section 8.1. Our results are shown in Figure 10. With the LSTM layer, our model converges after $\approx$ 1.5 million episodes; the simple feed-forward network doesn't come close. Based off these results and our earlier error in the code, the LSTM layer is essential to decreasing training time for the model.

8.3 Spatial Soft Argmax

Finally, we also experimented with adding a Spatial Soft Argmax (SSA) non-linearity in front of our convolutional layers, described in [1].

Briefly, SSA takes an output from a convolutional layer, and then for each channel: computes a softmax across all pixels, multiplies each pixel by its relative 2D position (x, y) $\in$ ([-1, 1], [-1, 1]), and sums across all pixels. The result is a pair of numbers (f_{cx}, f_{cy}) for each channel c, denoting the expected location of features in each channel.

Intuitively, SSA should work well for reducing the dimensionality of the parameters within each channel of a convolutional output because it takes any channel of dimension HxW and reduces it to dimension 2. However, with the 4 convolutional layers in our final A3C model, the dimension of the output channel from our last conv layer is only 3x3!

Consequently, since it does not make much sense to simply add an SSA non-linearity in front of our final A3C model's convolution layer, our setup for this experiment was a bit different from the ones above. In particular, we went back to a model that used 84x84 input images and only 2 convolution layers as described in [3], with the final conv output dimension being 11x11. We added an SSA nonlinearity in front of the last layer of this modified model, and benchmarked this modified architecture against our best GAE-based model from Section 8.1. Our results are shown in 11.

Unfortunately, the results are not very promising, Figure 11. The GAE model performs much better. We hypothesize that is is likely due to the SSA-based model having far fewer parameters than the GAE model,

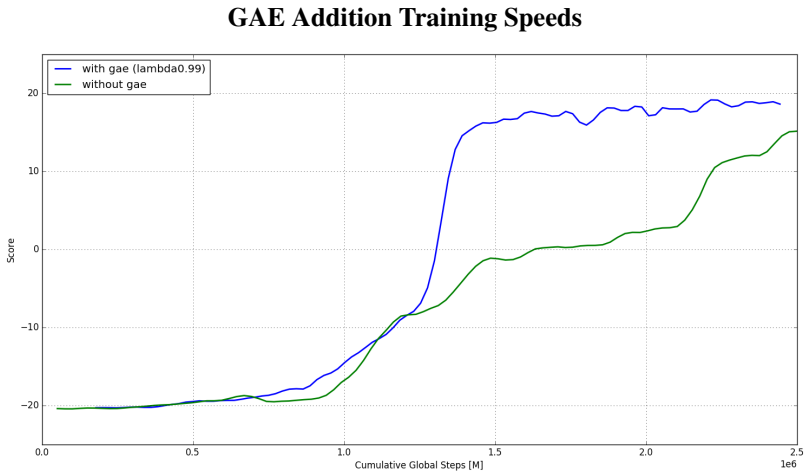

Figure 9: Comparison of training in with GAE implemented and without GAE implemented. Trained on Pong.

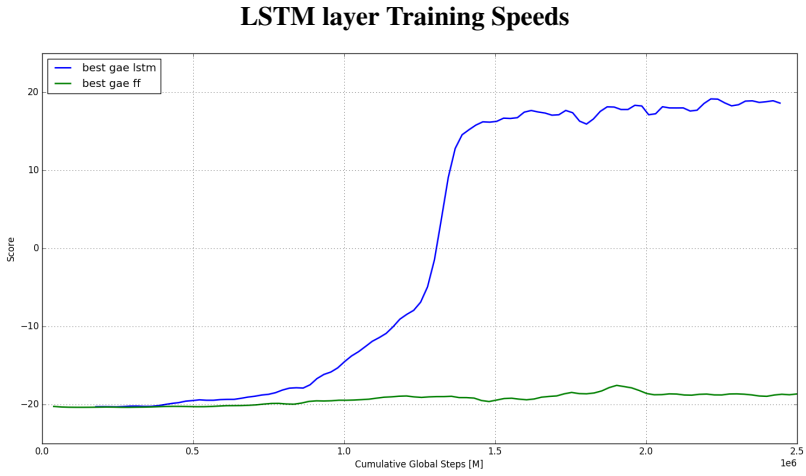

Figure 10: Training speeds of the A3C algorithm with and without a final LSTM layer. Trained on Pong.

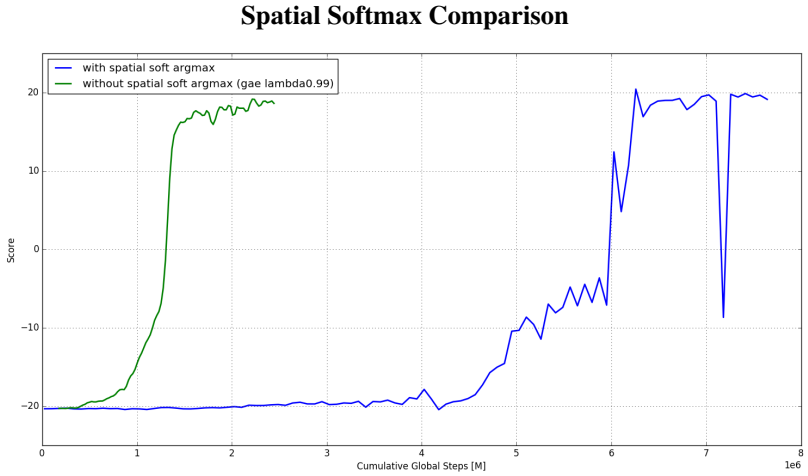

Figure 11: Training speeds of the A3C algorithm with and without Spacial Softmax implementation. Trained on Pong.

due to a reduced number of convolutional layers. Furthermore, (not shown) the SSA model trained about 2x slower, likely due to the increased image input size.

9 Lessons Learned

We found that combining several state-of-the-art models for deep reinforcement learning led to positive results in improving the training time for Atari games emulator. Given that we trained mostly on pong, the goal was that improve the speed at which the algorithm converged to the maximum score. We found that the A3C policy gradient method far outperformed a deep Q-learning implementation. Furthermore, the addition of a LSTM layer to the neural network was essential to rapid convergence of the algorithm. Finally, the advantages of using GAE were obvious when the optimal hyperparameter was found, but non-optimal values would lead to results that varied highly depending on the starting states of the algorithm, which were stochastic in nature.

10 Team Contributions

We both worked on programming with Tyler contributing 50% and Aleks contributing 50%. Alek's main individual role was supervising the server run-time of the training and Tyler's main individual role was researching and reading more papers to implement. We both contributed equally to the writing of report and presentations. The overall effort was split Tyler - 50%, Aleks - 50%.

References

[1] Levine, et. al. End-to-End Training of Deep Visuomotor Policies. Website Archive: https://arxiv.org/abs/1504.00702

[2] Mnih, et. al. Playing Atari with Deep Reinforcement Learning. Website Archive: https://arxiv.org/abs/1312.5602

[3] Mnih, et. al. Asynchronous Methods for Deep Reinforcement Learning. Website Archive: https://arxiv.org/abs/1602.01783

[4] Shulman, et. al. High-Dimensional Continuous Control Using Generalized Advantage Estimation. Website Archive: https://arxiv.org/abs/1506.02438

[5] Djork-Arn Clevert, Thomas Unterthiner, Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). Website Archive; https://arxiv.org/abs/1511.07289