A3C implementation for solving Atari game environments

Erik Kristian Janezic

Abstract—The area of reinforcement learning has been an interesting topic of machine learning for a long time. It mainly concerns the topics of training ”intelligent” actors which have to navigate an unknown environment while maximizing a certain predefined score. In this project we explore and implement an recently proposed approach of reinforcement learning known as asynchronous advantage actor critic or A3C for short. The model was proposed by a Google Ai team in 2016, and it is best known for its success in solving Atari 2600 based environments, and that it has no expensive hardware demands (such as GPUs) to successfully train the model.

Index Terms—reinforcement learning, asynchronous advantage actor critic, convolutional neural network, long short term memory, OpenAi gym

1 Introduction

Deep convolutional reinforcement learning has been the main ”go-to” approach for solving virtual game environments and motor control problems in the recent past. In 2016 a novel asynchronous approach was introduced by *Volodymyr Mnih et.al.*[1]. In this project we introduce our own implementation of Asynchronous Advantage Actor Critic (A3C) with convolution and LSTM layers. The main goal of the project is to train a model for solving Atari 2600 game environment Breakout provided by Open Ai gym.

2 Background

The modern approaches and ideas in reinforcement learning (RL) originate from the merging of three areas of research in the late 1980’s, first of them being *trial and error learning* inspired by research in animal psychology. Insights obtained by observing animal learning process when subjected to negative and positive stimuli form the rudiments of RL paradigm, namely positive-negative reward system dictated by the actors interaction with the environment.

The second important area were the studies of *optimal control* on Markov decision processes (MDP). Fundamental work in this area was done by Bellman in 1957 by introducing a family of function value optimization methods, known as dynamic programing, based on a dynamic state variable. This work represent the core concept of RL, namely finding the optimal actor action in each encountered environment state, to maximize the reward.

Third important thread of research contributing greatly to modern day RL was *temporal difference learning*. Also originating from animal psychology principles, it focuses on time sensitive estimation of some quantity. It was Marvin Minsky who first proposed that temporal nearness of learning reinforces might have important implications for artificial learning systems.

The three areas were connected together by Chris Watkins in 1987 with his development of Q-learning, which is the ground zero for understanding modern day RL methods and all its branches.

The base idea of RL approach today is that we represent our problem with actors and an environment on which actors can preform some actions. The environment is composed of many different states and have a reward associated with them. The goal of the actor is to take actions to transit between these states and maximize the reward return while it does so.

2.1 Asynchronous advantage actor critic

In this section we will describe the principles of A3C. To put it short we will have multiple actors executing actions on multiple instances of the environment and then the combined gradients of the policy loss will be used to update the shared model.

2.1.1 Actor critic

Usually we can classify RL algorithms as one of the following categories:

- Actor only methods search for the optimal policy directly which is guided by gradient estimations. The policy is directly evaluated and its parameters changed in the direction of improvement. An example of actor only method is William's REINFORCE [4]. These methods usually suffer from high variance in gradient estimations and the lack of learning when the policy changes since the new gradient is estimated independently[3].
- Critic only methods attempt to approximate exclusively the value function which can then be used to derive a near optimal policy. Examples of critic only methods are Q-learning, dynamic programing and temporal difference learning. The problem with these approaches is that they lack reliability in terms of finding a near optimal policy.[3]

Actor critic approach attempts to combine the best features of both approaches. The search for an optimal policy is guided by an incrementally optimized critic (which is a state value function). This approach greatly reduces the variance and improves the convergence of finding the near optimal policy if the search is gradient based.

2.1.2 Asynchronous

A major problem in online RL is that the agent observes non stationary data while the online updates of the model are strongly correlated. One way to address this problem is by introducing the experience replay, where agents store the experience which can be batched or randomly sampled to preform updates and this way reduce non-stationarity and decorrelate updates. The downside of this approach is that it limits us on the use of off-policy learning methods such as Q-learning.

Asynchronous execution of agents on multiple environments has been shown to resolve this issue [1]. The stabilizing effect comes from different actors exploring different environment states and contributing to a common state value function which is then used to evaluate the actors policy. Because of this there is no more need for experience replay, which reduces the memory complexity of the algorithm and enables us to use on-policy methods as well as off-policy.

2.1.3 Advantage

The advantage function is defined as:

$$A = Q(s,a) - V(s)$$

Where the $V(s)$ is a state value function and has a role of a critic and $Q(s,a)$ is the Q value for the action the agent chose to play. All the actors contribute to and share $V(s)$ which is essentially the approximation of the state s. By calculating the difference of $Q(s,a)$ and $V(s)$ the actor gets a feed back about how much better or worse its action was compared to the shared idea of the environment state.

3 Environment

Game environments for this project were obtained from OpenAi Gym [5]. Initially we decided to train the model on Atari 2600 games, more specifically on the Breakout-v0 environment. The environment object packages information about the state of the game and an interface through witch an actor can interact with the environment. The observation space of the environment is initially $210*160$ 2D array of 3 channel pixel values. We will later describe the preprocessing of these default observation values to reduce the computation complexity, but essentially our actor will learn to navigate the environment based on raw pixel value inputs. The Breakout environment also provides us with a set of 4 legal actions, them being: none,fire, move left and move right. Each execution of an action returns the new state of the environment and the reward for taking that action. In breakout rewards are gained for destroying a block and its value is dependent of block's tier.

4 Setup

The model was implemented with PyTorch[7] using Python 3.6. The training was done on a 8 core Intel(R) i7-3770 3.50GHz CPU on Ubuntu based desktop PC.

5 Model

5.1 Preprocessing

Our model will take a sequence of raw pixel values as inputs which will be the data to predict future actions. Before training the model we want to reduce the computational complexity of the problem.

In the picture below we can see a raw environment observation returned by the environment at each step.

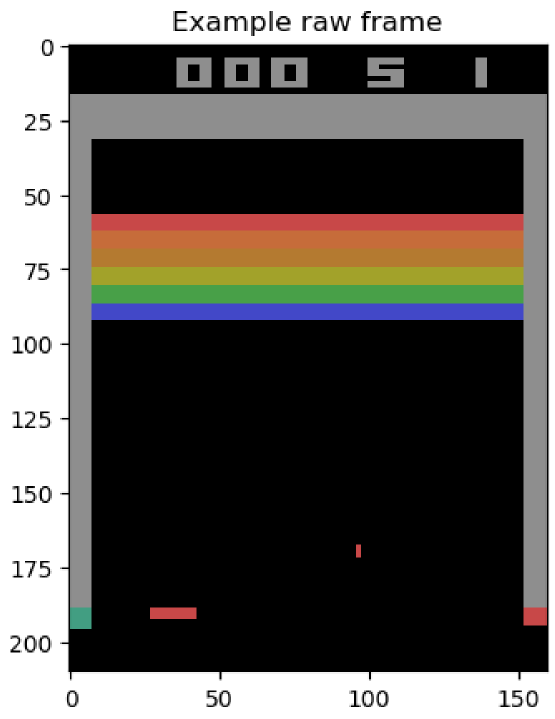

Fig. 1. 210 by 160 dimensional 3 channel raw output of the environment

We see that there is some irrelevant pixels at the top of the image, containing game score for the player to see. We decided to remove 20 topmost pixels and then make a two step transform of the image, first to $100x100$ pixels and then to $42x42$ pixels. Next we calculate the mean of the RGB value for each pixel and multiply it by $(1/255)$ to produce a black and white representation of the image, which is then normalized. The final output image of the normalized environment is seen on figure 2 below.

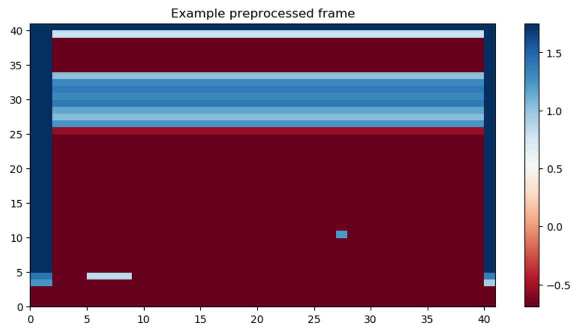

Fig. 2. 42 by 42 one channel preprocessed image

5.2 Convolution layers

Since our model will learn to navigate the environment from raw pixel values we will use a 4 layered convolutional neural network to learn features from environment images. The first convolution layer accepts the single channel 42 by 42 dimensional tensor of pixel values and detects 32 features on that image. Each of the three subsequent convolution layers accept a 32 dimensional volume of the previous layer as the input and outputs a new 32 dimensional volume tensor. In all layers we use a kernel of size 3 which we move over the input image with a stride of 2. The padding in all layers is set to 1. After the last convolution we flatten its output tensor to one dimensional tensor with 288 values which is then fed in to the long short term memory network.

5.3 Long Short Term Memory (LSTM)

LSTM network enables the model to learn sequence sensitive predictions from a sequence preceding environment states. For example in our environment we are trying to bounce the ball of a controllable platform which demands from the model to predict in which direction is the ball moving just from the environment state image. This would be nearly impossible without LSTM layer since it is impossible to determine the movement of a ball from a single stationary image. Thanks to the propagation of hidden and cell states among different time steps of the running environment in the LSTM we translate information of previous states to the currently observable state. Over time the network learns which features are important to predict the ball movement and position the controllable platform.

The input for our LSTM is a tensor of dimensions $1x288$ obtained from flattening the output of the last convolution layer, and a tuple (hx, cx) representing hidden states and cell states from the previous time step (hx and cx are initialized to 0 at each environment restart). The output of the LSTM is a $1x256$ dimensional encoded tensor which is then used as a state value approximator and agent actor selector.

5.4 Actor and critic

In our model actor and critic share all network layers up until now. The $1x256$ dimensional output of the LSTM is used to encode the critic and the actor separately. For the actor we use a fully connected linear layer which maps $1x256$ vector to a $1x5$ dimensional tensor representing actions available to play. The critic is also produced with a fully connected linear layer only this time we map all of 256 entries of LSTM output to one single value which represent the estimation of the state value.

5.5 Training

Before the training we initialize one shared model which will be updated by multiple actors and multiple actor models which will be used to explore the environment (in our case we used 8 exploratory models since our CPU has 8 cores, each actor is executing in its own CPU core).

At the start or restart of the environment we ensure that our actors share the learnable parameters with the shared model. The model will learn by accumulating a reward over 20 steps after which it is updated. At each step the actors neural network is given a environment state and outputs a state value tensor, Q values for the actions and hidden and cell states to be used in the LSTM in next iteration. The action Q values are transformed with a soft-max function so probability of choosing each action is evaluated on the interval $0 < p_a < 1$. The action space is then randomly sampled from the multinomial probability distribution and a single action to be played is selected. Multinomial sampling introduces the exploratory aspect to the model.

At this point it is important to note that we will take policy entropy into account when computing the policy loss, since it has been shown that it decreases the probability for sub-optimal premature convergence of the model [1]. Entropy for the policy is calculated by the following formula:

$$H = -\sum_{n=1}^{numActions} p_n * log_2 p_n$$

where p_n is the probability of taking each action and $log_2 p_n$ is the logarithm of that probability.

The computed entropy of the policy is appended to a list for later use as is the $log_2 p_a$ of the action model decided to play and the current state value returned by the critic. The selected action is then executed in the environment which returns the immediate reward which is also appended to a list of rewards.

When the algorithm does 20 exploratory steps and accumulates immediate rewards, state values, logarithms of probabilities of played actions and policy entropies we start the procedure of computing the value and policy loss.

We use advantage function and cumulative discounted reward to compute the value loss. For computing the policy loss we use generalized advantage estimations, logarithms of probabilities of taken actions at each step and entropy at each time step

5.5.1 Computing the value loss

We define the computation as an iterative procedure over over the 20 exploratory steps taken. First we initialize the cumulative reward (R) and value loss (VL) to 0. Then we iterate over the list of rewards in the reverse order and compute the R_t, advantage (A_t) and VL_t for each time step:

1) $R_t = \gamma * R_{t-1} + r_t$ Where r_t is a reward at index t in the list and γ is the discount factor which gives more importance to recent rewards and has the value 0.99 in our example
2) $A_t = R_t - V_t$ Where V_t is the state value at index t in the list of accumulated state values
3) $VL_t = VL_{t-1} + 0.5 * A_t^2$

5.5.2 *Computing the policy loss*

We define the computation as an iterative procedure over over the 20 exploratory steps taken. Since we are going to use generalized advantage estimation (GAE)[6] we also need to compute the temporal difference (TD) of rewards in each step.

First we initialize the policy loss (PL) and GAE to 0. Then we iterate over the list of rewards in the reverse order and compute the PL_t, advantage (GAE_t) and TD_t for each time step:

1) $TD_t = r_t + \gamma * V_{t+1} - V_t$ where r_t is the reward at index t in the accumulated list of rewards, V_{t+1} is the state value of the previous step ($t+1$ because we are iterating over the list in reveres order) and V_i is the state value of the current state.
2) $GAE_t = GAE_{t-1} * \gamma * \tau + TD_t$
3) $PL_t = PL_{t-1} - log_2 p_t * GAE_t - 0.01 * H_t$

The computed VL and PL is then combined as follows:

$$TotalLoss = PL + 0.5 * VL$$

and backpropagated through the neural network to update the weights based on the gradient.

5.6 Optimizer

Evaluating the gradients of an objective function (loss) with respect to all of its parameters is computationally expensive and thus infeasible. The basic go-to approach to address this problem is using the stochastic gradient descent to evaluate gradients. However this method hasn't proven to be efficient in on-line learning and when the gradients are sparse. Different gradient descent optimization algorithms have been developed to resolve this issue. In this project we use an optimization algorithm known as Adam (adaptive moment estimation) presented in [2]. Adam has been derived from two previous optimization algorithms AdaGrad, which works well with sparse gradients and RMSprop which works well in on-line and non non-stationary settings.

5.7 Model Schema

This section illustrates the general schema of the A3C model. In the picture below 3 agents are represented although in our case we used 8 exploration agents.

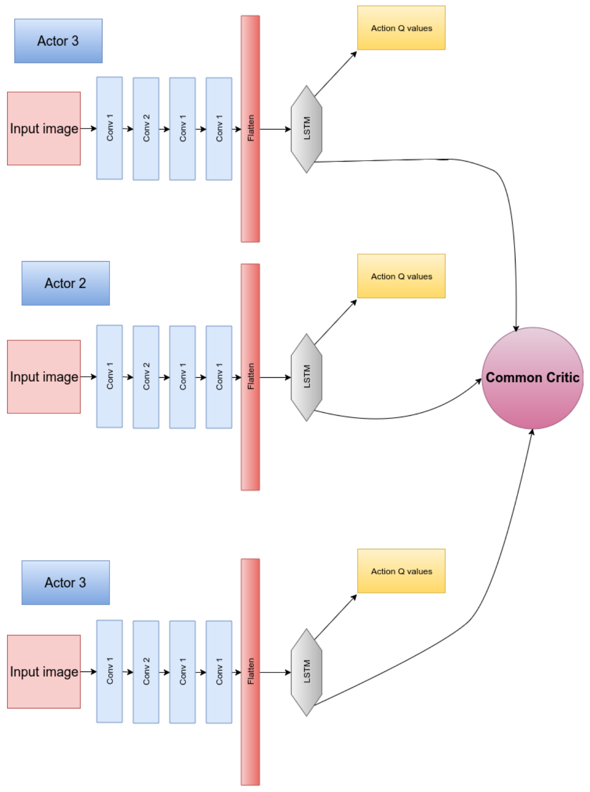

Fig. 3. Schematic representation of our A3C model

6 Results

The first run of the model lasted 48 hours and the training was stopped when we started observing limiting behavior of the moving average reward seen in figure 4 as blue trend. We were interested if adding an extra LSTM layer would improve temporal generalizations. We let the model with 2 LSTMs run for 69 hours since the trend was more consistent at 48 hours where we stopped the first model. The trends of the second model are colored orange.

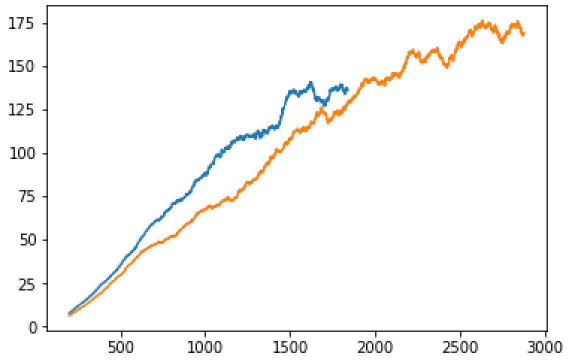

Fig. 4. Breakout 200 data point moving average of rewards for 48 hours (blue, one layer LSTM), and for 69 hours (orange, two layer LSTM)

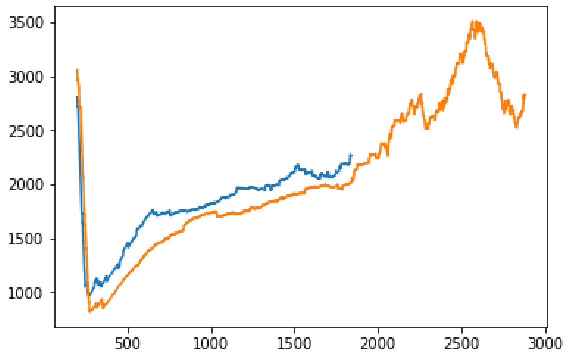

Fig. 5. Breakout 200 data point moving average of episode lengths for 48 hours (blue, one layer LSTM), and for 69 hours (orange, two layer LSTM)

It seems from figure 5 as if our models episode length increased dramatically in the interval between 2300th and 2700th testing epoch. But after inspecting the videos and a graph with no averaging (figure 6) we saw that the algorithm decided to stop playing if it achieved a good enough score thus increasing episode length to its maximum (defined by us to 10000)

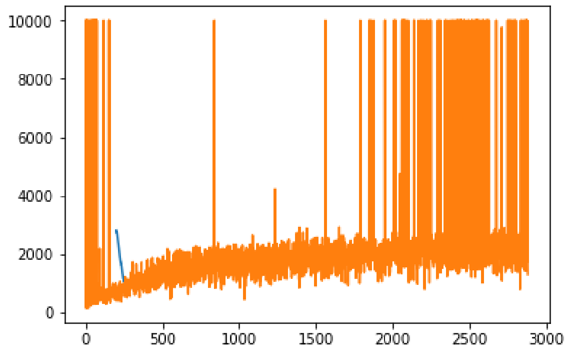

Fig. 6. Breakout episode lengths for 48 hours (blue, one layer LSTM), and for 69 hours (orange, two layer LSTM)

It is also interesting to note that the average reward didn't start plateauing until around 1600th testing episode while the average episode length started plateauing at around 700th testing episode already. After examining the video recordings we speculated that the agent started to exploit the game mechanics in such a way that it created a small hole in all levels of bricks which enabled the ball to bounce between the remaining top level bricks and the edge of the environment, thus producing a large increase in reward value and incrementing episode lengths only slightly.

7 Conclusions

We successfully implemented an A3C algorithm for solving Breakout. We believe that the performance could be improved even further with fine tuning the meta parameters and playing around with different objective functions. Due to the lack of time this isn't possible so only base experiments were performed. In the videos of the agent playing we can see that the model successfully predicts where the ball will fall and moves the platform to that position, however the movement of the platform is a bit jerky which is most often the reason for loosing a life. The other reason is that the model sometimes fails to compensate for an accelerated ball (the ball starts to move faster if it destroys a brick of a higher tier), and thus arrives to the balls position to late. We suspect that the first problem might be resolved if we reduce the number of steps taken by the model before updating the weights. If there will be time we will do some extra tests before the presentation day.

REFERENCES

[1] Mnih Volodymyr et.al., Asynchronous Methods for Deep Reinforcement Learning, 2016.
[2] Diederik P. Kingma, Jimmy Ba, Adam: A Method for Stochastic Optimization, 2014.
[3] Vijay R. Konda,John N. Tsitsiklis,Actor-Critic Algorithms, Nips, 2000, vol.42.
[4] Ronald J. Williams, Simple Statistical Gradient-Following Algorithms for Connections Reinforcement Learning, Machine Learning, 1992, vol.8.
[5] OpenAi Gym, *https://gym.openai.com/docs/*
[6] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, Pieter Abbeel, High-Dimensional Continuous Control Using Generalized Advantage Estimation, 2015
[7] PyTorch, *http://pytorch.org/*