Atari: Deep Reinforcement Learning with Parallel advantage actor-critic

Introduction

One of the goals of the deep learning community is to create AI that can make optimal decisions based on raw visual and audio input. In 2014, Google DeepMind successfully used convolutional neural networks (CNNs) with deep reinforcement learning to teach a computer to play classic score-based Atari games. For this project, I use OpenAI Gym, an actively developed interface for gaming environments. Simply, it is an interface which provides various environments which simulate reinforcement learning problems. One of the core challenges with computer vision is obtaining enough data to properly train a neural network, and OpenAI Gym provides a clean interface with dozens of different environments.

Data

The input to the model is a sequence of pixel images as arrays (Width x Height x 3) generated by a particular OpenAI Gym environment. I then use a PAAC to output an action from the action space of the game. The output that the model will learn is an action from the environments action space in order to maximize future reward from a given state. Specifically, each environment has an observation state space, an action space to interact with the environment to transition between states, and a reward associated with performing a particular action in a given state. This information is fundamental to any reinforcement learning problem.

All of the data used for training the model comes from interacting with the OpenAI Gym Interface. Interacting with the Gym interface has three main steps: registering the desired game with Gym, resetting the environment to get the initial state, then applying a step on the environment to generate a successor state. The input which is required to step in the

environment is an action value. More specifically, it is an integer ranging from [0, numactions). [3]

Preprocessing

Once it receives the image from the Gym environment, it applies a couple steps to format the image to finally use an input to the model. First, it converts the image to gray scale, which should reduce discrepancies between episodes with different background settings. It also reduces the size of the third dimension from 3 for RGB to 1. Afterwards, it normalizes values to be between 0 and 1 and reduce the image to 80x80 pixels. This allows for consistent input sizes across games as well as decreasing dimensionality of each layer of the network, allowing for faster passes and fewer parameters per layer. Finally, it stacks last 4 frames as input to network to form the 80x80x4 input to the network. For the reward values, it clips all incoming rewards such that positive rewards are all 1, negative rewards are -1, and 0 rewards remain 0.

Method

The implementation of PAAC is using the TensorFlow library, and the model are running on Google Cloud Compute Engine with 8 cores and an Nvidia Tesla K80 GPU. The Network architecture is outlined in figure [1]. It are using 2 convolution layers with ReLU non-linearity activations, followed by 2 fully connected layers. Fully connected layers are separated by ReLU function similar to convolution layers. The final output layer dimension is equal to the number of valid actions allowed in the game. The values at this output layer represent the given the input state for each valid action. At startup, it initialize all the weight matrices using normal distribution with a standard deviation of .01. At beginning of training, it first populate the replay memory by choosing random actions for 10,000 steps and it are not updating network weights during this

preliminary training step. Once replay memory buffer is partially filled, it start training. It use TensorFlow's Adam optimization algorithm with an initial learning rate of 0.001.

For this project, it will explore the effectiveness of image processing techniques when applied to Parallel Advantage Actor-Critic Methods for Deep Reinforcement Learning. Specifically, it will look at using various convolutional neural network models and techniques to improve the PAAC method for the purpose of generalizing game play across multiple game instances. The PAAC algorithm is effective because it employs parallel actors which each follow a different exploration policy. It essentially aggregates and summarizes the experiences of each actor, which effectively stabilizes training by removing correlations in the data. This differs from a synchronous approach such as experience replay used in Deep Q-Learning which also removes correlations, but at the cost of more memory and computation [1]. Experience replay requires maintaining a history of state, action, reward, and next state pairs. A batch is then randomly sampled from the history to update the policy gradients[1].

The asynchronous aspect of PAAC simply means that the algorithm utilizes worker threads to collect independent experience rollouts while having a global network to aggregate all of the information into its network parameters [1]. At each step, the worker threads get a copy of the global network's parameters and use those to select its agent's next action.[1] Each worker computes its corresponding loss and gradients using the experiences and then the global network's parameters are updated.[1] The benefits of using an asynchronous algorithm are two-fold: there is no explicit need for experience replay as used in the PAAC algorithm [3] since rollouts are independent of one another (although it could still be used for data efficiency [3]) and the other is faster data collection.

This means that updates and action selection must be done individually on a per-actor basis, something not well suited for GPUs. PAAC performs action selection for all actors at once, and a

single update with data from all actors, synchronously. This leads to a computationally efficient algorithm that performs true on-policy.

Evaluation

We now present the results from the three inference models we used to provide intermediate reward signals to the PAAC agent. I evaluate the results using metrics, specifically the reward per episode of the model on the training period. The final evaluation consists of observing game-play, as the goal is to ensure that the agent makes turns. The episode-reward graph shows how the agent learns over the training period. Clearly, the agent improves quite dramatically.

Conclusion

My first exploration into teaching a reinforcement learning agent to learn how to play Atari games was quite difficult because of the introduction of new concepts and experience. Setting up the OpenAI environment on remote server is troublesome because it requires installing the correct drivers with no OpenGL files. The main reason is that there is no GUI while running on cloud computing, and thus, recording video files is a cumbersome task. In addition, Keras and TF.learn had some initial issues when I first started to write the code, and eventually switched to TensorFlow.

Sources

[1] Clemente, Alfredo. Efficient Parallel Methods for Deep Reinforcement Learning. 2017.

[2] Mnih, Volodymyr. Asynchronous Methods for Deep Reinforcement Learning. 2016.

[3] Mnih, Volodymyr. Playing Atari with Deep Reinforcement Learning. 2013.