Project Baseline

CS 8395-03 Visual Analytics & Machine Learning

Saroj Kumar Sahoo
Vanderbilt University
saroj.k.sahoo@vanderbilt.edu

1 Introduction

With the ever-increasing amount of data being generated the number of problems that can be solved using Artificial Intelligence nowadays is enormous. Deep Learning received a lot of attention from researchers and can be seen in use in our day to day lives. Whereas, Deep reinforcement learning is still in its research phase and is believed to have an equal impact as Deep Learning. Due to the black box nature of Deep Reinforcement Learning visualizing and understanding the decision-making process can help improve the behavior of the models. However, this topic is still to be explored. In this project, I explored an already existing work [1], where saliency maps were used to show the attention of an expert agent while choosing an action given a state.

2 Detailed Description

2.1 PPO

In the original implementation the author used A3C algorithm to train the agent. For this project, a different algorithm was used which is the current state of the art technique in Deep RL and is known to perform well namely Proximal Policy Optimization (PPO)[2].

Before we can discuss the specifics about the PPO algorithm, we must first address an issue which arises from a fundamental difference in how RL agents and ANNs learn. A neural network requires immediate feedback after every state to know if it made the right choice. However, in an RL problem, it is usually not possible to immediately decide if an action was good or not. The agent must wait and see for a few states, perhaps until the end of the episode, before it can figure out whether the sequence of actions were really beneficial or not. The way current Deep RL algorithms get around this issue is through a method called **Action-Replay**. In Action-Replay, the ANN is run in evaluation mode, where no training occurs. The first state is sent to the network, and it returns a policy. Then an action is chosen based on this policy, and the environment returns a new state and a reward for the action. This new state is then fed to the network, and the process repeats a fixed number of times (a parameter to be varied in the course of experimentation). This allows the program to build a sequence of (state-action-reward) triplets. This sequence is known as a trajectory. Then the return can be calculated for each triplet based on all the following triplets. Finally, the network is trained on all the triplets in this trajectory (usually multiple times), using the calculated returns to generate a seemingly immediate feedback to the network. A great advantage of this method is that multiple environments can be run in parallel (perhaps on different processes) while a single network (sitting on a parent process) can be used to determine the action for each of these environments. As long as the returns are calculated separately for each environment's trajectory, there is no issue. This can greatly reduce training time. Another trick that is commonly used is to shuffle the triplets in the trajectories during training (after calculating the returns). This is done to de-correlate the states the network sees as consecutive inputs, which greatly increases the network's ability to learn the important features in the state.

Now that the issue of how to generate immediate feedback has been solved, the next question is: what form should this feedback take? In other words, how exactly should the return of a state be defined, or calculated? The answer to this lies in how the A2C model works. The Actor in the A2C is in charge of predicting the policy. This means, it must learn which actions performed better than expected, and which actions performed worse than expected. With this information, the Actor can then try to decrease the probability of poorly performing actions, and increase the probability of well performing actions. However, the keyword in "better than expected" is "expected". The network must have some idea of how well it should perform from a specific state. In other words, it must have some idea of the value function of the state. This is where the Critic comes in. The Critic predicts an expected value, and then the Actor compares this with the actual return, and decides whether the action was good, or bad. This comparison is done with what is called the Advantage function. This function is calculated by a method called the General Advantage Estimation (GAE), which can be described by a recurrent equation:

$$A_{t=T-1} = R_t - V_t$$

$$A_t = R_t + \gamma V_{t+1} - V_t + \gamma\lambda A_{t+1}$$

Where A_t, R_t, V_t are the Advantage, Reward, and Critic predicted Value of state t respectively. T refers to the terminal state, or the end of the episode. Since it doesn't make sense to talk about the action chosen at the terminal state, the Advantage is not calculated for it. Instead, it is calculated from the state just before the terminal state. γ and λ are parameters that need to be found through exploration.
A little exploration of the equation shows that the Advantage function defined as above can be thought of almost as $return - value$, which is what we wanted to use to determine whether a chosen action was good or not.
From the Advantage function, it becomes clear that $J_t = A_t + V_t$ is the return (a discounted sum of future rewards) for state t. Thus, some kind of L^p norm such as mean-square error between J_t and V_t can serve as the loss for the Critic part of the network.

$$L^{Critic} = L^p(V_t, J_t)$$

For the Actor part, the loss is composed as:

$$L^{Actor} = -min(\frac{p_t}{p_t^{old}}A_t, clamp(\frac{p_t}{p_t^{old}}, 1-\epsilon, 1+\epsilon)A_t)$$

Where p_t p_t^{old} are the probabilities of the chosen action in the current and previous networks respectively (before that round of training started). The $clamp()$ function forces the ratio of probabilities to be between $1-\epsilon$ and $1-\epsilon$ where ϵ is another parameter that must be learned from experimentation. Finally, the minimum of the product of the clamped ratio of probabilities with the Advantage, and the original ratio of probabilities with the Advantage, is used as the loss function for the Actor. The reason the clamping is done is so that the updates to the Actor network don't cause large changes in the policy and lead to instability in learning. Finally, an entropy bonus may also be added to the loss to help further stabilize training, and also help slightly with exploration. The entropy is a measure of the uncertainty of a probability distribution. For example, a policy with a high probability for one action and low probabilities for the other actions would have low uncertainty, and thus low entropy. Forcing the network to increase the entropy would help decrease sudden increases in the probability of a certain action. Although counting on just the entropy bonus for exploration will not yield positive results.

For our project, we used a single network for both the actor and the critic, and so we combined all the above losses to form the overall loss function:

$$L = L^{Actor} + Critcoef * L^{Critic} - Entrcoef * Entropy(\pi_t)$$

Where π_t is the policy of state t and both $Critcoef$ and $Entrcoef$ are parameters that need to be found through experimentation. After putting everything we have seen above together, we get the PPO Algorithm. Since nowhere in the algorithm the agent tries to understand which actions in a state correspond to which of the possible next states (the state-action transition function), the algorithm is considered model-free.
The final piece of any RL algorithm is in how the exploration vs exploitation trade-off is formulated. We want the agent to explore new options, to see if it can find a better trajectory to beat the task, but since there are countless possible trajectories, it is not possible to try them all, and so we also want the agent to follow those trajectories (at least partially) that it knows will yield good results. The way we have done this in our project is to slightly change how our network chooses an action. After the Actor has returned a policy, the probabilities of the actions are sequentially added to

each other. That means, the probability of the first action is added to the probability of the second action, and is considered the summed probability of the second action (just for this action selection part, during training, the original probabilities are used). The original probabilities of the first and second actions are added to the third action, and this is considered the summed probability of the third action. This process is repeated for all the actions. Note that although we are calling these summed values probabilities, they do not sum to 1. At this point, we pick a random number between 0 and 1, and the first action with a summed probability larger than this random number is chosen as the action of the agent. This method allows the agent to explore, but also makes sure that a lot of the time, the action with highest probability is chosen.

2.2 Saliency Maps

To understand how the agent learned and what actions an expert agent takes we will use Saliency Maps. Saliency Maps may give us some important information that the agent uses to make decisions. To do this saliency maps is constructed for both Actor and Critic. These are not normal jacobian based saliency maps rather these are Perturbation based Saliency. In Perturbation based saliency, a perturbation is used to produce rich and insightful saliency maps. Given an Image I_t at time t, we let $\phi(I_t, i, j)$ denotes perturbation I_t centered at pixel coordinates (i,j). The following formula is used to create perturbations:

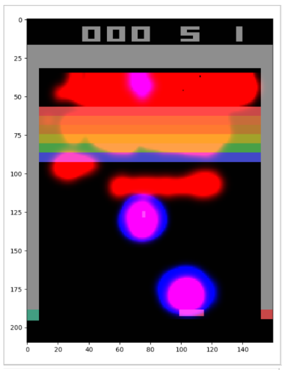
$$\phi(I_t, i, j) = I_t \odot (1 = M(i,j)) + A(I_t, \sigma_a) \odot M(i,j)$$

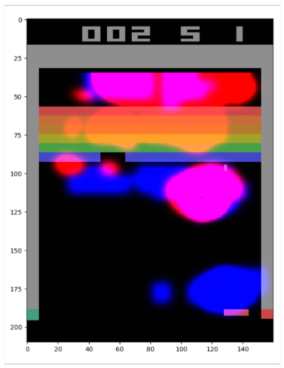
The above equation can be interpreted as spatial uncertainty added to region around (i,j).

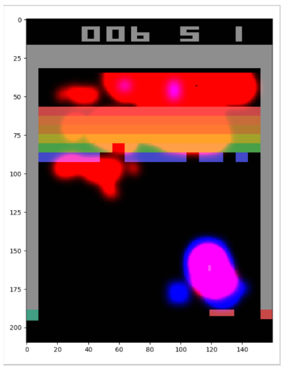
Next, both the unaltered image and perturbed image are passed through the actor, critic network to predict the policy and value. Saliency metric is given by:

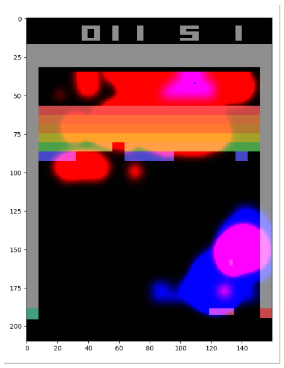
$$S_\pi(t, i, j) = 0.5 * (\pi_u(I_t) - \pi_u(I'_t))$$

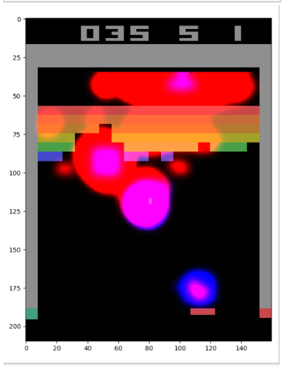
3 Results











4 Discussion

The training of RL agent takes much longer than we expected. Even after 24+ hours of training, an overfit model could not be produced. The model had not converged so we went ahead and created the saliency maps. As it can be seen from the images that these are not very informative as of now. Maybe changing and parameters might help? Training more might help as well. We failed to reproduce the tunneling effect as mentioned in [1]. I would guess the agent need to overfit for the effect to be seen.

Code from these github profiles were used in someway [3] and [4]

References

[1] Sam Greydanus, Anurag Koul, Jonathan Dodge, and Alan Fern. Visualizing and understanding atari agents. *arXiv preprint arXiv:1711.00138*, 2017.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[3] Wizdom13. Rnd. `https://github.com/wizdom13/RND-Pytorch`.

[4] Sam. `https://github.com/greydanus/visualize_atari`.