RL Project

David S. Hippocampus
Department of Computer Science
University of Bath
Bath, BA2 7AY
hippo@bath.ac.uk

1 Problem Definition

2 Background

2.1 DQN (Mirco)

2.2 A2C (Chris)

Intro to Policy Gradient Methods

A set of alternative methods to value-based reinforcement methods that can be applied to this problem is policy gradient methods. Whilst value-based methods such as DQN learn values of actions and use these estimates to select actions, policy gradient methods do not need to consult a value function and, instead, learn a parameterised policy to select actions (Sutton and Barto, 2018).

One commonly used policy parameterisation for discrete action spaces, *soft-max in action preferences*. In selecting actions using a parameterised policy, policy gradient methods use *action preferences* rather than action values. This distinction is crucial as it confers several benefits to policy gradient methods such as the ability to learn a stochastic optimal policy (Sutton and Barto, 2018). One benefit particularly relevant to Breakout is that the policy can approach a deterministic policy which would not be possible if action-values were used (Sutton and Barto, 2018). This is relevant to the problem of breakout as in some situations such as where the ball is close to the paddle, the desired behaviour is for the agent to deterministically move toward the ball to return it and have no chance of moving away from it.

Policy Gradient techniques have seen success in high-profile reinforcement learning breakthroughs in game-playing. Most notably, AlphaGo (Silver et al. 2016) used the REINFORCE algorithm to train the policy network used in the agent.

Intro to Actor Critic

One Policy-Gradient method that has been shown to be effective on similar tasks to Breakout is the Actor Critic method.Actor Critic chosen as a promising method for this problem over other methods as it offers significant benefits over other policy-gradient methods such as REINFORCE. REINFORCE is a Monte Carlo algorithm and as such, suffer from problems such as slow learning and issues with online implementation. In contrast, Actor-Critic methods are analogous to TD methods and so avoid these issues (Sutton and Barto, 2018). Avoiding the issue of slow learning is particularly crucial for a complex environment such as Breakout. Training a DQN agent capable of performing above average performance on Atari games required a training time of 8 days on a GPU (Mnih et al. 2016). For a small team and the time window on this assignment, an agent that learned slower than this would not be feasible. Variants of Actor-Critic including Asynchronous Advantage Actor-Critic (A3C) and Advantage Actor-Critic (A2C) have shown strong performance on similar tasks to the problem considered in this paper such as in Minh et al. (2016).

2.3 Async Q-Learning (Peter)

Mnih et al (2016) present asynchronous variants of reinforcement learning algorithms including both Q-Learning and Actor-Critic (A3C). The paper claims the asynchronous variants can produce excellent results on single multi-core CPUs with training times of just a few hours.

Unlike DQN, the asynchronous methods do not require an experience replay memory because multiple agents working in parallel means that the data is fragmented and is sufficiently decorellated.

The results published by Mnih et al (2016) are impressive and show that after 10 hours training with breakout, Async Q-Learning scored 240 and A3C scored 460, whilst DQN only managed 25.

TODO : Weaknesses? Check publications for A3C vs A2C, I've seen comments that say A3C does not improve on A2C, but can't find anything definitive. ??

3 Method

3.1 DQN (Mirco)

3.2 A2C (Chris)

Basic Actor Critic explanation

The second method chosen to solve our specific problem of Atari Breakout is Advantage Actor-Critic (A2C). As discussed in section 2, A2C is a variation of the Actor-Critic method which implements an advantage function. Whilst several variations of Actor-Critic methods are used in practice, all Actor-Critic methods are comprised of two key components at their core: an Actor and a Critic. The role of the actor is to learn a parameterised policy which is used to select the actions. The role of the critic is to provide a reinforcing signal to the actor based on actor performance which is used to update the parameters of the policy. This reinforcement signal is based upon the state-values learned by the critic.

Implementation details for Actor Critic networks including architecture and pre-processing

In this implementation, the actor policy is parameterised by a convolutional neural network (CNN) where the parameters are the weights of the neural network. The critic is also implemented as a convolutional neural network. The neural network architecture for both the actor and the critic is drawn from the architecture used in Silver et al. (2015) with some notable changes. For the Actor, the output layer uses a softmax activation function for the output layer with a single neuron for each possible action that the actor can take. This is to achieve the softmax in action preferences policy parameterisation described in section 2. This softmax output is sampled in order for the agent to take an action in a given state. The critic network uses the same architecture but uses a single output neuron with linear activation. This output value represents the action-value the critic prescribes to a given state. Both networks also use a RandomUniform initialisation to the output layers with a relatively small range between min and max values (0 - 0.02). This was included as initial runs of the algorithm showed high sensitivity to initial weights on the output layers, particularly on the actor.

The input to both networks is a pre-processed 84x84x4 image that represents a state the Breakout environment. The preprocessing applied is the same as that applied in Silver et al. (2015), including greyscale, image cropping and frameskip. The 4 channels of the image are implemented using the StateHistory class. The instance of StateHistory passed to the agent as input includes the current frame and the 3 previous frames. This gives the agent context to the current frame such as the direction the ball is travelling in.

Advantage-Actor-Critic explanation

The issue with vanilla Actor-Critic as described above is that it is generally implemented as an on-policy learning algorithm and on-policy learning algorithms are unstable due to temporally correlated data across timesteps (Li, Bing and Yang, 2018). This was addressed by Minh et al. (2016) through the introduction of Asynchronous Advantage Actor-Critic (A3C). In A3C, the asynchronous component is that multiple agents are executed in parallel on multiple instances of an environment with the aim of removing the non-stationarity since agents will be experiencing a variety of different states at any one time step (Minh et al. 2016). The Advantage component of A3C refers to the use of an

advantage function in the reinforcing signals sent to the actor. This advantage value quantifies how much better or worse an action is than the average available action by using the critic's valuation of states (Graesser and Loon, 2020). An algorithm that implements the advantage function but does not execute multiple agents in parallel is known as Advantage Actor Critic (A2C).

Results from OpenAI (2017) showed that A2C may be a more suitable candidate for the problem of Breakout than A3C. Indeed on a set of Atari games, including Breakout, their results showed that A2C outperformed A3C and the noise introduced by the asynchronous implementation failed to deliver any performance benefit (OpenAI, 2017). As such, A2C is chosen as the Actor-Critic implementation.

In A2C, the inclusion of the Advantage function is motivated in a similar way to inclusion of baselines in other policy gradient methods which is to decrease variance and thus improve the stability of training (Sutton and Barto, 2018). In the case of A2C, the state-value produced by the critic is used as the baseline. The update rule for A2C is given in Yoon (2019) and shown below:

Advantage Actor Critic

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t) \tag{1}$$

Where:

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t) \tag{2}$$

The critic uses a TD update to adjust it's estimate of the value of a state. The intuition behind which is as follows: Each step of the environment, the critic network calculates a value estimate for both the previous state and the next-state that our agent is in. In a similar fashion to TD(0), the critic uses the bootstrapped estimate of the next state, the reward at timestep t+1 and the discount factor to calculate a target value for the state. The mean squared difference between the critic's initial value of the state and the target value is used to update the critic and force it to adjust it's estimate of the state closer to the target value.

However, the target TD value is also used in the calculation of the advantage function and, hence, the actor loss. As seen in the above equation, The advantage function is calculated by subtracting the critic's valuation of the previous state from the target TD update. The intuition behind this is that the difference between these two values represents the value of taking the action that the actor took in that state against the average value of that state (The average value of that state) (Yoon, 2019). This is used in the actor update to provide a measure of how good that action was against the baseline for that state.

The update equation above is calculated for the actor. The first component ∇_θ represents gradient (delta), i.e which action the agent took whether the agent went with or against the current policy which sets the direction of the update for each action. For example, if the agent is selecting from two actions and selects action one, delta will be positive for action one and the size of delta will be larger if the action went against the current policy. This value is multiplied by the log probabilities of the action taken in the given state. The negative log probabilities are used rather than the raw probabilities in order to provide numerical stability as multiplying together probabilities approaches 0 which can yield issues in computing these numbers quickly.

The advantage, which represents the reinforcement signal from the critic, tells the actor whether the action taken was better than the average reward for that state. Thus, combining this with the above variables, if the advantage is positive, the agent will be told to take more of the action it did take and less of the ones it did not take. Conversely, a negative advantage value tells the agent to take less of the action it did take and more of the actions it did not take.

A2C implementation including inclusion of entropy bias and details of custom loss function

In this implementation, the critic loss is easily implemented using mean squared loss of the TD update. However, the loss for the actor is calculated in a custom loss function called *actorcustomloss*. This function implements the update described above aswell as implementing techniques useful for ensuring stability in training such as clipping values. However, the loss function implemented here also includes an entropy regularisation term. This yields a new update equation for the actor

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} -\nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t) + \beta H_t \tag{3}$$

Here, β is the entropy weight which is a hyperparameter that controls how much the actor loss comprises of the entropy value. The entropy term was included as initial runs showed that even with very small learning rates (1e-5 and below), the agent would very quickly select one action every time, making it incapable of sampling other actions. Thus the agent would always get stuck in a local minima, unable to improve. The entropy term, originally proposed by William and Peng (1991) for use with the REINFORCE algorithm, encourages the actor to output a distribution with higher entropy by penalising low entropy outputs through the loss function. A distribution of actions with higher entropy is a distribution with a high chance of sampling each of the different values. For example, in an environment with four actions such as Breakout, a maximum entropy output distirbution would have a 0.25 probability of selecting each action.

3.3 Async Q-Learning (Peter)

Async Q-Learning was chosen as it was similar to DQN, but promised improved learning in a shorter time. Pseudo code is shown in Appendix C: Async Q-Learning pseudo-code

Like DQN, Async Q-Learning uses an e-greedy policy based on a value network to select actions to take and then uses a target network to get an action value used to calculate the loss. The weights in the target network are occasionally reset to match the value network.

Where Async Q-Learning differs from DQN is that it runs multiple worker agents in separate processes which share the value and target networks. Each worker agent keeps a local copy of the value and target networks to calculate the losses and gradients which it then applies to the shared networks every few steps. Once the worker agent has updated the shared network, it resets the weights in its own copies to match shared ones. As the worker agent takes a number of steps before updating the shared network, the updates are done like mini-batches and the chances of one worker agent overwritting changes being made by another are reduced.

Having multiple worker agents running at the same time, and the non-deterministic nature of breakout means that different areas of the game will be explored at the same time. The different worker agents were also given different epsilon values for their e-greedy policies to further increase the variety of exploration.

The e-greedy policies for the worker agents all started with an epsilon of 1.0, which decayed down to 0.05, 0.1 or 0.1 after 1,000 episodes.

A controller process handles the creation of multiple workers as different processes. The controller gives each worker a queue which allows communication between controller and workers. Each worker starts the atari environment and plays games of breakout. An episode is a complete game, and at the end of each episode the worker sends some details to the controller.

The controller keeps track of the total number of episodes completed by all the workers, and uses another process to play games with a policy based on the shared value network with an epsilon of 0.01 to provide a measure of how well the policy performs. The graphs in the results section use

TODO : Maybe a diagram of the different processes would help here?

4 Results

4.1 DQN (Mirco)

4.2 A2C (Chris)

(Need to finish Final Run before writing this one) Fig 4.1 below shows the performance of the agent averaged peer 10 episodes over T number of episodes. (Insert performance graph here)

As can be seen in Fig 4.1, the

4.3 Async Q-Learning (Peter)

TODO : Add images for the training using asynch Q-Learning. Show score against number of frames processed.

TODO : add a graph showing 10 hours training of asynch Q-Learning ?

5 Discussion

5.1 DQN (Mirco)

5.2 A2C (Chris)

5.3 Async Q-Learning (Peter)

Initial results were encouraging, but the performance tended to level off quite quickly and often started to decrease.

Asynch Q-Learning did not produce results in the time scales mentioned by Mnih et al (2016). However, matching the scores on the data efficiency and training time graphs from Mnih et al (2016) show that they completed approx 25 epochs in 10 hours. A single epoch corresponds to 4 million frames giving 100 million frames in 10 hours. In contrast our implementation for asynch Q-Learning only managed to process 5 million frames in 10 hours, so approximately 20 times slower.

TODO : put these numbers into a suitable table ??

6 Future Work

6.1 A2C (Chris)

In the A2C implementation used, the advantage function is calculated using n-step (Where n=1) returns. This only includes one step of actual rewards. Since rewards are generated from a single trajectory they tend to have high variance but are unbiased. However, the state valuation reflects all trajectories seen so far and thus has lower variance but introduces bias. As a result, the value of n controls the bias-variance tradeoff (Graesser and Loon, 2020).Thus, using 1 step returns provides us with a tradeoff of low variance but introduces significant bias through the reliance on state value.

In contrast, Generalised Advantage Estimate (GAE) overcomes the need to set n as a hyperparameter by using an exponentially-weighted average of advantages calculated with different values of n (Schulman et al., 2015). GAE provides a method by which the low-variance, high-bias one step advantage is most heavily weighted but the lower-weighted contributions from higher values of n still contribute. By tweaking the decay rate of contribution from these higher-variance estimators with higher values of n, GAE allows a more finely controlled tradeoff between bias and variance than the hard cut-off tradeoff in n-step methods (Graesser and Loon, 2020). In comparison to the 1-step method implemented, this is likely to allow a significant reduction in bias whilst controlling the increase in variance. By testing several values for the GAE decay rate, it is possible that a lower-bias, higher-variance solution may exist that performs better than the n-step advantage implementation.

7 Personal Experience

In the training of the A2C agent, one factor that was both a difficulty and a surprise was the sensitivity of the agent to changes in the algorithm hyperparameters. For example, the entropy weight was found to be a very important and sensitive hyperparameter. If set slightly too low the agent would revert to selecting a single action with certainty every time. If set too low the agent would not move significantly from a random sample across the four action. However, the difference between these outcomes was often a very small change. The value used in the implementation was 0.06. A value of 0.075 was found to get stuck very close to an even weighting of actions for all states (0.25 uniformly) and a value of 0.045 would get stuck in a local minima of selecting only one action. Thus, with only a difference of 0.015 either side of the value used, the performance did not just degrade slightly but became practically untrainable.

References

file:///Users/mogul/Downloads/Mastering_the_game_of_Go_with_deep_neural_networks.pdf - Silver et al. Alpha Go Sutton and Barto (2018)

https://bath-ac-primo.hosted.exlibrisgroup.com/primo-explore/fulldisplay?docid=44BAT_ALMA_DS5184633880002761&context=L&vid=44BAT_VU1&lang=en_US&search_scope=CSCOP_44BAT_DEEP&adaptor=Local%20Search%20Engine&isFrbr=true&tab=local&query=any,contains,actor%20critic&offset=0&pcAvailability=false (Hands on RL Python)

https://arxiv.org/pdf/1602.01783.pdf - Mnih et al.

https://arxiv.org/pdf/1806.06914.pdf - Li, Bing and Yang, 2018

https://openai.com/blog/baselines-acktr-a2c/ - A2C vs A3C

https://www.nature.com/articles/nature14236?wm=book_wap_0005 - Silver et al 2015, human level control

https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f Yoon 2019

Schulman et al. - generalised Advantage Estimate William and Peng, 1991 - Entropy maximisation term for regularisation on REINFORCE

Mnih, V., et al., 2016. Asynchronous Methods for Deep Reinforcement Learning. ArXiv, [Online] 1602.01783. Available from: https://arxiv.org/pdf/1602.01783.pdf [Accessed 28 Dec 2022]

Appendices

Appendix A: DQN pseudo-code

Taken from the course notes.

Initialise replay memory D to capacity N
Initialise action-value network $\hat{q}_1$ with parameters $\boldsymbol{\theta}_1 \in \mathbb{R}^d$ arbitrarily
Initialise target action-value network $\hat{q}_2$ with parameters $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$
for each episode **do**
    Initialise S
    for for each step of episode **do**
        Choose action A in state S using policy derived from $\hat{q}_1\left(S, \cdot, \theta_1\right)$
        Take action A observe reward R and next-state S'
        Store transition (S, A, R, S') in D
        for each transition $\left(S_j, A_j, R_j, S'_j\right)$ in minibatch sampled from D **do**
            $y = \begin{cases} R_j & \text{if } S'_j \text{ is terminal} \\ R_j + \gamma \max_{a'} \hat{q}_2\left(S'_j, a', \boldsymbol{\theta}_2\right) & \text{otherwise} \end{cases}$
            $\hat{y} = \hat{q}_1\left(S_j, A_j, \boldsymbol{\theta}_1\right)$
            Perform gradient descent step $\nabla_{\theta_1} L_\delta(y, \hat{y})$
        Every C time-steps, update $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$

Appendix B: A2C pseudo-code

TODO

Appendix C: Async Q-Learning pseudo-code

From Mnih et al (2016)

// Assume global shared θ, θ^-, and counter $T = 0$
Initialise thread step counter $t \leftarrow 0$
Initialise target network weights $\theta^- \leftarrow \theta$
Initialise network gradients $d\theta \leftarrow 0$
Get initial stats s
while $T <= T_{\max}$ **do**
    Take action a with ϵ-greedy policy based on $Q\left(s, a; \theta\right)$
    Receive new state s' and reward r
    $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q\left(s', a', \theta^-\right) & \text{for non terminal } s' \end{cases}$
    Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \frac{\delta(y - Q(s,a,\theta))^2}{\delta\theta}$
    $s = s'$
    $T \leftarrow T + 1$ and $t \leftarrow t + 1$
    if T mod $I_{target} == 0$ **then**
        Update the target network $\theta^- \leftarrow \theta$
    if t mod $I_{AsyncUpdate} == 0$ or s is terminal **then**
        Perform asynchronous update ot θ using $d\theta$
        Clear gradients $d\theta \leftarrow 0$

Appendix D: