CSCI-4968 Proximal Policy Optimization

Christopher Metcalfe, Avery Iorio

April 2023

1 Introduction to Reinforcement Learning

Definition and representation. [6] Reinforcement learning is a type of machine learning that focuses on training agents to make decisions in a dynamic environment in order to maximize a long-term reward signal. Mathematically, reinforcement learning can be modeled as a Markov Decision Process (MDP) represented by a tuple (S, A, P, R, η), where:

- S is the set of states of the environment
- A is the set of actions the agent can take
- P is the transition function that defines the probability of transitioning from state s to state s' after taking action a:

$$\mathbb{P}[S_t|S_{t-1}, A_{t-1}] = P(S_{t-1}, A_{t-1}, S_t)$$

- R is the reward function that defines the immediate reward the agent receives after taking action a in state s:

$$R(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$$

  By convention, the reward associated with some transition is actually received on the next step. The reward is typically determined by which state you land in.

- η is the initial state distribution

Episode or Trace. We can then define an episode or trace as a sequence of states, actions, and observed rewards:

$$S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, ...$$

Return. Define the return G_t of a trace as the total cumulative discounted reward from your current point in time.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$$

$$\gamma \in (0, 1)$$

Intuitively this is motivated by the fact that current rewards are more important than later ones, it also allows us to bound our total return to a finite value given that all R_t are bounded by some M for all t:

$$G_t \leq M \sum_{k=0}^{\infty} \gamma^k = \frac{M}{1-\gamma}$$

The policy. The policy π is a function that maps observations/states to actions. It can either be deterministic:

$$\pi(s) = a$$

or stochastic:

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$$

State and Action value functions.[6] We are now ready to define the State-Value function $v_\pi^t(s)$. The value function returns the expected return from being in state s given you are operating on your policy π, it is a measure of how 'good' the current state is. We will be considering the infinite horizon case only.

$$v_\pi^t(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s] = \mathbb{E}_\pi[G_t|S_t = s]$$

The action value function is similar to state value, but instead considers a specific action:

$$q_\pi^t(s, a) = \mathbb{E}_\pi[R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+...|S_t = s, A_t = a] = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

Intuitively it measures how good a specific state-action pair is. It should be noted that one can express the state value function in terms of the action value function as such:

$$v_\pi^t(s) = \sum_a \pi(a|s) q_\pi^t(s, a)$$

Bellman equations. [6] There exists a recursive definition for both functions known as the Bellman equation, which define the state and action value in terms of the value of the next state visited and the next action taken. For the infinite horizon case they are defined as follows:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$

Bellman Optimality Equations. A policy π is better than another policy π' if the following holds:

$$v_\pi(s) \geq v_{\pi'}(s), \forall s \in S$$

Policy from value function. Given a value function therefore the associated greedily optimal corresponding deterministic policy can be extracted as such:

$$\pi^{'}(s) = arg\max_a q_\pi(s,a) = arg\max_a [R(s,a) + \sum_{s^{'}} \gamma v_\pi(s^{'}) P(s,a,s^{'})]$$

Policy Improvement Theorem. [6] The previous definitions build together into the policy improvement theorem, which states a policy π' is as good, or better than, another policy π if the following holds:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

Optimal Policy A policy π^* is optimal then if there exists no policy better than π^*. This leads to the bellman optimality equation which defines an optimal policy as any policy that satisfies the following:

$$\pi_*(s) = arg\max_a q_{\pi_*}(s,a)$$

Core differences with supervised learning. The core difference between reinforcement learning and supervised learning is that reinforcement Learning involves learning from feedback obtained by interacting with a dynamic environment, attempting to learn a policy that will maximize cumulative expected return while in supervised learning we are learning a function by optimizing to minimize a loss function that measures how well the learned function is predicting outputs given the ground truth values (labels).

Overall optimization objective. Putting all the subsequent points together the general optimization objective in Reinforcement Learning can be framed as maximizing the expected return with respect to some policy π.

$$\max_\pi \mathbb{E}_\pi[G]$$

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

2 Q-Learning

Temporal Difference Learning. Temporal Difference learning is a learning technique that utilizes the bellman equation along with the current existing estimate of the value function to make a 'bootstrapped' estimate of the return at a state. The update rule for TD learning is as follows:

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Its titled temporal difference as it's measuring the difference between our current estimate $V(S_t)$ and the value of $V(S_t)$ as defined by the bellman equation given the next state in time S_{t+1} and the next reward R_{t+1}.

Q-Learning. Q-Learning is a type of off-policy temporal difference reinforcement learning algorithm. The algorithm schema is pictured above. Q-Learning works very well in systems with easily discretizable action and state-spaces however, it does not function well in environments with infinitely many actions/states. Note, S^+ is the set of all states (State-Space) while $A(s)$ is the set of all actions possible at a state s. Finally a terminal or sink state is a state with transition probability of 1 and a reward of 0 (Think the finish line of a race). This is an important concept in RL as an MDP with terminal states allows us to frame the situation in the infinite horizon setting, where the value functions become time-independent. Examining the update rule of Q-learning we have:

$$Q(S, A) = Q(S, A) + \alpha[R + \gamma \max_a Q(S^{'}, a) - Q(S, A)]$$

The current Q-Value is updated according to a TD estimate of the return, however the next action for purposes of the UPDATE is not chosen based off the current policy, but rather based upon whatever action will maximize the subsequent Q-value function in the next state $S^{'}$.

ϵ-Greedy Policies. An epsilon greedy policy is some deterministic policy π that is modified such that the policy takes the action $\pi(s)$ with probability $1 - \epsilon$ and any other possible action with probability ϵ. It exists to encourage exploration, specifically the visitation of new state-action pairs as is necessary for Q-learning to converge to an optimal policy.

On-Policy vs. Off-Policy. An On-Policy Reinforcement Learning algorithm gets training data from the policy it is currently optimizing, whereas an Off-Policy algorithm has a behaviour policy to collect training data and a separate target-policy that it optimizes based off the experiences collected by the behavior policy.

Deep Q-Learning. [3] Classic Q-Learning stores all state-action pair values in a table however this is intractable for problems with a large or infinite state-space. Instead the Q-Value function can be approximated by another function such as a Neural Network as is the case in Deep Q-Learning. To learn a good Q-function we can map the problem into the supervised learning setting, with the labels being generated by another target network (which in practice is just a snapshot of our current network at some point in the past, whose parameters are updated to our current networks weights after every N steps).

$$y = R_{t+1} + \gamma \max_a Q_{\theta_{i-1}}(S_{t+1}, a)$$

The loss can then be calculated as the distance from the current Q(S,A) estimate value and this bootstrapped value, constructed with MSE (other losses are valid) as:

$$L(\theta_i, S_t, A_t, y) = (Q_{\theta_i}(S_t, A_t) - y)^2$$

And can be optimized by gradient descent for a single step. In practice we usually apply experience replay to train the Q-Network, in which we generate and save experiences letting our agent act based on a ϵ greedy policy derived from our current Q Network. Once a full batch of experiences has been generated, a random experience is sampled from the batch for training. Randomly sampling from the experiences breaks the bias that may have come from the sequential nature of a particular environment.

DQN Architecture. [3] The architecture for the Q-Network is generally some number of linear layers with an input dimension equivalent to the dimension of the state S_{t+1} and an output dimension the number of possible actions a. Such that the output neuron activation's are equivalent to the Q-Value for that state-action pair. An action a can then be selected by an ϵ-greedy approach.

Limitations of DQN. While DQN is an exceptionally powerful and popular method capable of handling continuous-state-space problems it is still restricted to finite action-space MDP's, this motivates the next topic, policy gradient methods.

3 Policy Gradient Methods

Optimization Objective. As stated before the overall optimization objective of reinforcement learning can be framed as maximizing the total expected reward with respect to some parameterized policy. Formulated formally as:

$$\max_{\theta} J(\theta) = \max_{\theta} \mathbb{E}_{\pi}[r(\tau)]$$

Where $r(\tau)$ represents the total reward from some trajectory τ generated by a policy π_θ. It can be seen then we may improve our policy parameters by gradient ascent as follows:

$$\theta^{k+1} = \theta^k + \alpha \nabla J(\theta)$$

Policy Gradient Theorem. [6] The policy gradient theorem states that the derivative of the expected total reward is the expectation of the product of the total reward and the gradient of the log of the policy π_θ:

$$\nabla J(\theta) = \mathbb{E}_{\pi}[r(\tau) \nabla \log \pi(\tau)]$$

Given:

$$\pi_\theta(\tau) = \mathcal{P}(s_0) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}, r_{t+1}|s_t, a_t)$$

Where $P(s_0)$ represents the ergodic distribution of starting in some state s_0 and $p(s_{t+1}, r_{t+1}|s_t, a_t)$ captures the state transition probabilities. These are all multiplied together as the transitions are independent of each other as they are part of a Markov chain. Taking the log yields:

$$\log \pi_\theta(\tau) = \log \mathcal{P}(s_0) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \sum_{t=1}^{T} \log p(s_{t+1}, r_{t+1}|s_t, a_t)$$

And then taking the gradient w.r.t. θ:

$$\nabla \log \pi_\theta(\tau) = \sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)$$

This is a key point of the derivation, both the initial state distribution and the transition dynamics are removed, overall resulting in the policy gradient:

$$\nabla J(\theta) = \mathbb{E}_{\pi} \left[r(\tau) \left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) \right) \right]$$

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

In practice we can replace $r(\tau)$ with the discounted return G_t as in the context of our objective rewards from the past contribute nothing. Arriving at the policy gradient utilized in REINFORCE (Vanilla Policy Gradient):

$$\nabla J(\theta) = \mathbb{E}_\pi \left[\left(\sum_{t=1}^{T} G_t \nabla \log \pi_\theta(a_t|s_t) \right) \right]$$

REINFORCE (Vanilla Policy Gradient). REINFORCE uses the gradient estimate derived above to improve it's existing policy by gradient ascent with some hyperparameter α. It takes a gradient ascent step for each state transition, scaled by the discounted return G_t.

REINFORCE with Baseline: In practice the return G_t introduces a lot of variance into the gradient estimate, to mitigate this variance a baseline can be introduced to normalize the returns, such as the state-value. The gradient estimate then becomes:

$$\nabla J(\theta) = \mathbb{E}_\pi \left[\left(\sum_{t=1}^{T} (G_t - V(S_t)) \nabla \log \pi_\theta(a_t|s_t) \right) \right]$$

Where another network would be trained to approximate the Value function. While this method has reduced variance and improved performance it still is extremely sensitive to variations in the learning rate parameter α and can be difficult to tune between different problems. The core issue is that too large a policy change can easily cause divergence, as future training data is based upon the updated policy. Furthermore, REINFORCE has poor data efficiency, as new trajectories need to be generated following every single policy update to estimate a new gradient. To address these concerns the design of a more robust algorithm, that's data efficient and scalable is motivated.

4 Proximal Policy Optimization

Overview. [4] Proximal Policy Optimization aims to replicate the data efficiency and reliability of TRPO, while utilizing only first-order optimization. ’

Advantage Function. Both TRPO and PPO utilize an advantage estimate, which is essentially a reshaped reward signal. The advantage function measures the difference between the action value and state value at a state s under a policy π:

$$A^{\pi_\theta}(s,a) = \mathbb{E}(Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s))$$

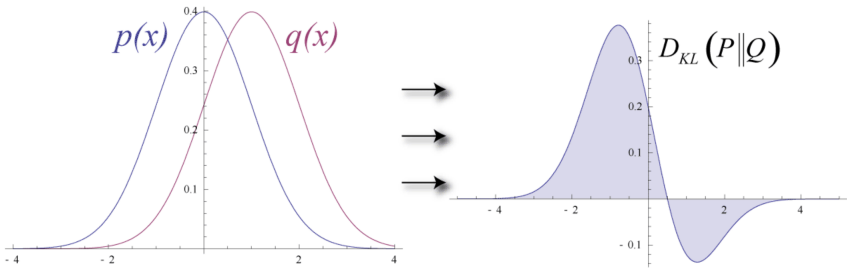
Intuitively measuring how advantageous an action a is.

Kullback–Leibler divergence (KL). The KL divergence is a type of statistical distance that measures how different one probability distribution is from another. It can be calculated as follows:

$$KL(P||Q) = \sum_x P(x) \log(\frac{P(x)}{Q(x)})$$

Or for the continuous case:

$$KL(P||Q) = \int_{-\infty}^{+\infty} P(x) \log(\frac{P(x)}{Q(x)}) dx$$



A visualization of the KL divergence of two probability distributions, the overall value of $KL(P||Q)$, is the integral of the function $P(x) \log(\frac{P(x)}{Q(x)})$.

Trust Region Policy Optimization (TRPO).[5] In TRPO a surrogate objective function is maximized subject to a constraint on the size of the policy update:

$$\max_{\theta} \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}(a_t|s_t)}} A_t \right]$$

$$\text{subject to } \mathbb{E}_t[\text{KL}[\pi_{\theta_{old}(.|s_t)}, \pi_\theta(.|s_t)]] \leq \delta$$

Where θ_{old} is the vector of policy parameters before the update. This problem can be relatively efficiently solved utilizing the conjugate gradient method, following a linear approximation of the objective and a quadratic approximation of the constraint. Furthermore the underlying theory justifying TRPO actually suggests to utilize a penalty constraint:

$$\max_{\theta} \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}(a_t|s_t)}} A_t - \beta \text{KL}[\pi_{\theta_{old}(.|s_t)}, \pi_\theta(.|s_t)] \right]$$

This objective is not utilized however as finding a β that performs well between different problems or even through a single problem is difficult. PPO's goal is to further modify the objective function of TRPO such that the monotonic improvement of TRPO can be emulated while utilizing only first-order methods.

Probability ratio. Let us define the probability ratio $r_t(\theta)$ as follows:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}(a_t|s_t)}}$$

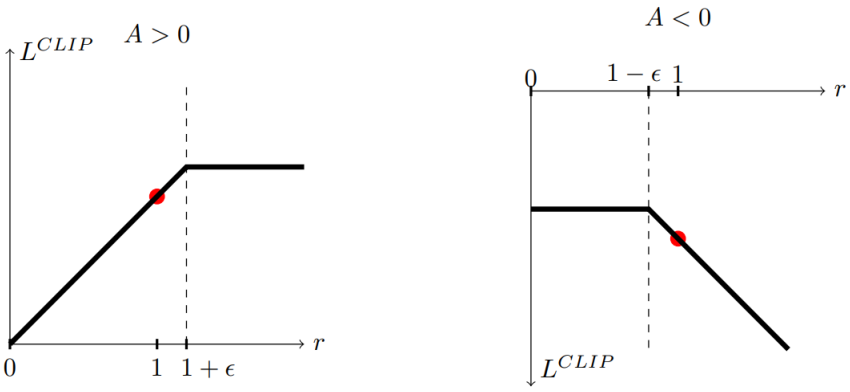
Such that:

$$r_t(\theta_{old}) = 1$$

PPO clipped objective function. The main proposed surrogate objective function for PPO can then be formulated as follows:

$$L_t^{CLIP}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t \right]$$

Where epsilon is a hyperparameter that roughly governs how far from the original policy the update is allowed to go. The motivations for this objective function is to form a lower (pessimistic) bound on the unclipped objective function $r_t(\theta)A_t$ from TRPO. By clipping the probability ratio between $[1-\epsilon, 1+\epsilon]$ the incentive for the probability ratio r_t to change beyond that interval is removed. By taking the minimum of the unclipped and clipped objective we ensure the resulting objective function is a strict lower bound in which the probability ratio is ignored only when it would improve the objective, and included when it would make it worse.



The plots above depict a single time-step of the surrogate function L^{CLIP} as a function of the probability ratio r_t, note that a positive or negative advantage determines whether r_t is clipped at $(1+\epsilon)$ or $(1-\epsilon)$. The red dot indicates the initial starting point of the optimization. Many of these terms our summed together to form L^{CLIP}.

Practical Considerations. In situations in which a neural network architecture that shares parameters between the policy and value functions is used, the loss function must be modified to combine the policy surrogate and value function error term. Combining these terms yields the resulting objective:

$$L_t^{CLIP+VF}(\theta) = \mathbb{E}_t[L_t^{CLIP}(\theta) - cL_t^{VF}(\theta)]$$

Where c is some constant and $L^{VF} = (V_\theta(s_t) - V_t^{targ})^2$, where the target network is some snapshot of $V_\theta(s_t)$ that is updated every n iterations, where n is a hyperparameter. A simple implementation of PPO is pictured below.

Algorithm 5 PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for $k = 0, 1, 2, \ldots$ **do**
Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
Compute policy update

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k})\right]\right]$$

end for

Appendix A

Problem Set

1. The function, $clip(v, a, b)$ is expressed as:

$$clip(v, a, b) = \begin{cases} b & v \geq b \\ v & a < v < b \\ a & v \leq a \end{cases}$$

The original form of PPO with clip is,

$$L_{\theta_k}^{CLIP}(\theta) = E_{s,a,\theta_k}\left[min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), \quad clip\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right)A^{\pi_{\theta_k}}(s,a)\right)\right]$$

Using the definition of Clip and the equation above, derive the simplified PPO with clip that will be used in question 2. (The final equation should not contain the clip function)

2. The simplified PPO with clip update is given by the following equation:

$$\theta_{k+1} = argmax_\theta E_{s,a,\pi_{\theta_k}}[L(s, a, \theta_k, \theta]$$

where,

$$L(s, a, \theta_k, \theta) = min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), \quad g(\epsilon, \ A^{\pi_{\theta_k}}(s,a))\right)$$

where,

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

One of the benefits of PPO is that policy updates are done judiciously without simply shrinking the step size. By manipulating L, show why $g(\epsilon, A)$ must handle two different cases of A and explain what these cases intuitively mean when it comes to updating the policy.

3. Plot the function L for when the policy has a more likely negative reward at the given state and when the policy has a more likely positive reward at the give state (i.e. when the advantage A is both positive and negative).

4. Why is KL divergence computationally expensive during gradient descent as compared to the clip function. Why might KL divergence fail to protect the optimization from making a very poor policy update (hint: think of where it gets its probability distributions from).

5. Looking back to the function $L^{CLIP}_{\theta_k}(\theta)$ from problem 1 what is the benefit of taking the minimum in the function

$$min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), \quad clip\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right)\right)$$

In other words, if we are already constraining the policy update region using the clip function, why do we still need to further restrict policy updates?

Answers

1. To derive the simplified PPO with clip update.
   Take,

$$L(r, A, \epsilon) = min(rA, \quad clip(r,\ 1-\epsilon, 1+\epsilon)\ A)$$

   If $A \geq 0$:

$$L = A\ min(r, clip(r,\ 1-\epsilon, 1+\epsilon))$$

$$L = A\ min\left(r, \begin{cases} 1+\epsilon & r \geq 1+\epsilon \\ r & (1-\epsilon) < r < (1+\epsilon)) \\ 1-\epsilon & r \leq 1-\epsilon \end{cases}\right)$$

$$L = A \begin{cases} min(r, 1+\epsilon) & r \geq 1+\epsilon \\ min(r, r) & (1-\epsilon) < r < (1+\epsilon)) \\ min(r, 1-\epsilon) & r \leq 1-\epsilon \end{cases}$$

$$L = A \begin{cases} 1+\epsilon & r \geq 1+\epsilon \\ r & (1-\epsilon) < r < (1+\epsilon)) \\ r & r \leq 1-\epsilon \end{cases}$$

$$L = A\ min(r, 1+\epsilon)$$

$$L = min(rA, (1+\epsilon)A)$$

   If $A < 0$:

$$L = A\ max(r, clip(r,\ 1-\epsilon, 1+\epsilon))$$

   Since A is negative, moving it to the outside of min changes the min to a max.

$$L = A\ max\left(r, \begin{cases} 1+\epsilon & r \geq 1+\epsilon \\ r & (1-\epsilon) < r < (1+\epsilon)) \\ 1-\epsilon & r \leq 1-\epsilon \end{cases}\right)$$

$$L = A \begin{cases} max(r, 1+\epsilon) & r \geq 1+\epsilon \\ max(r, r) & (1-\epsilon) < r < (1+\epsilon)) \\ max(r, 1-\epsilon) & r \leq 1-\epsilon \end{cases}$$

$$L = A \begin{cases} r & r \geq 1+\epsilon \\ r & (1-\epsilon) < r < (1+\epsilon)) \\ 1-\epsilon & r \leq 1-\epsilon \end{cases}$$

$$L = A\ min(r, 1-\epsilon)$$

$$L = min(rA, (1-\epsilon)A)$$

2. At a given update step there are two alternatives.
   The first is that the state action pair is has a positive reward meaning the objective surrogate reduces to,

$$L(s, a, \theta_k, \theta) = min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)},\ (1+\epsilon)\right) A^{\pi_{\theta_k}}(s, a)$$

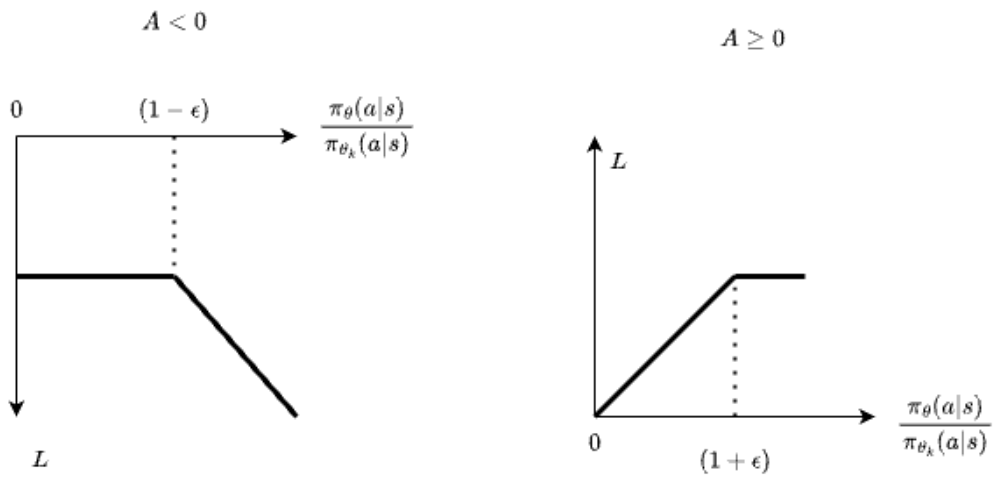
   From this we can see that the policy update will change the policy to increase $\pi_\theta(a|s)$ so that the term $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ grows. Since the policy is trying to increase the actions that led to the reward. However, once $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ becomes too large (i.e. larger than $1 + \epsilon$) the policy update does not go any further in changing the policy.

   The other alternative is that the state action pair has a negative reward meaning the objective surrogate reduces to,

$$L(s, a, \theta_k, \theta) = max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)},\ (1-\epsilon)\right) A^{\pi_{\theta_k}}(s, a)$$

   Similarly to the first case, the policy update wants to decrease $\pi_\theta(a|s)$ which can be thought of as the likelihood of that state policy pair. However, the update will not be able to change the policy too much since the most it can decrease $\pi_\theta(a|s)$ is $(1 - \epsilon)$.

3. There are two possible cases for the function L



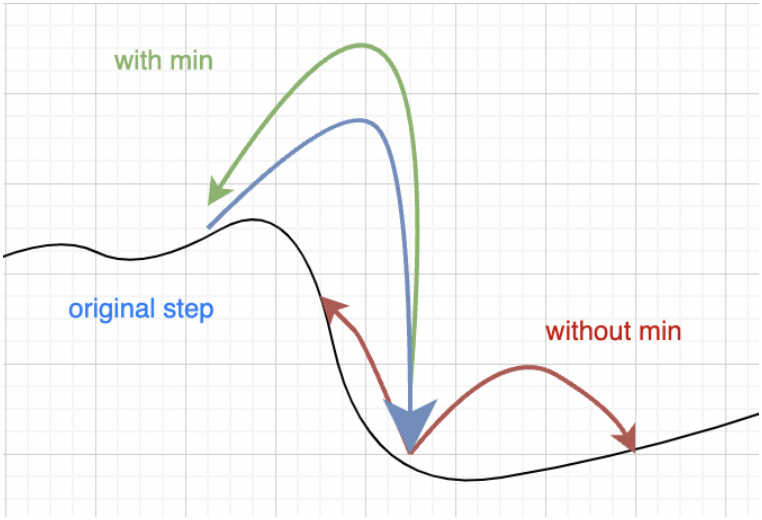
   These graphs visualize how the PPO is able to have its own version of a "trust region" which is a portion of the local policy space where it feels

comfortable updating its policy to. This helps mitigate the risk of the policy going off of a "cliff" and getting stuck with little to no hope of recovering.

4. KL divergence can be difficult to calculate because it requires approximating an indefinite integral or infinite sum. Finding the gradient of these functions is difficult and computationally expensive making TRPO complicated and difficult to implement. Also, KL divergence can fail to prevent a poor policy update if the policy update significantly changes the probability distribution but also happens to result in a similar KL number. Another reason could be that the KL divergence might be poorly approximated due to the probability distributions being hard to integrate numerically.

5. The reason for taking the minimum is to handle the case where we make a large policy update that gives us a higher probability of a negative reward. While we could just allow $CLIP$ by itself to try to find its way back to the previously better policy, there is a chance that the $CLIP$ function for the following step won't allow an update large enough to get back to where the policy originally was. This would then cause $CLIP$ to prune those possible changes and then make the best policy update in the new "best" direction.

   However, by taking the min we allow an update with a greater probability of a negative award to simply undo the change and revert back to the previous policy. This allows PPO to proverbially go to the precipice of a policy cliff to explore potential new paths but then come back to safety instead of falling off and having to find a new way back up.



Appendix B: Implementing PPO

Introduction

To be clear, the primary motivation behind PPO, TRPO, and similar reinforcement learning algorithms is to quantify and subsequently constrain the change in the policy without simply shrinking the learning rate. Although these techniques attempt to converge to an optimal solution more stably than Vanilla Policy Gradients, the primary benefit is that they can move much more freely in the policy space as long as the Kullback-Leibler divergence or clip constraints are met.
However, this brings up the question of whether for simple reinforcement learning problems it is more practical to simply increase the step size to get the same rapid exploration of the policy space. To answer this question, as well as show the benefits of PPO, we will look at two very simple tasks from the OpenAI gym: CartPole-v1 and Acrobot-v1:

Figure 1: CartPole-v1 vs Acrobot-v1

CartPole is a benchmark problem in RL, where the agent receives information about the pole's angle and velocity, and uses this information to decide whether to move the cart left or right in order to balance the pole attached by an unactuated joint, with the goal of keeping the pole upright for as long as possible.

Acrobot is a similar problem that consists of two links connected linearly to form a chain, with one end of the chain fixed. The joint between the two links is actuated. The goal is to apply torques on the actuated joint to swing the free end of the linear chain above a given height while starting from the initial state of hanging downwards.

Comparing Reinforce and PPO

We chose to examine these two problems in particular to demonstrate how the dominance of PPO over Vanilla Policy gradient methods varies based on the application and how likely an overly-greedy policy update will result in the algorithm failing.

CartPole

REINFORCE Due to the low-dimensionality of its action space the optimal policy for the CartPole benchmark can be easily found with REINFORCE with an aggressive learning rate.

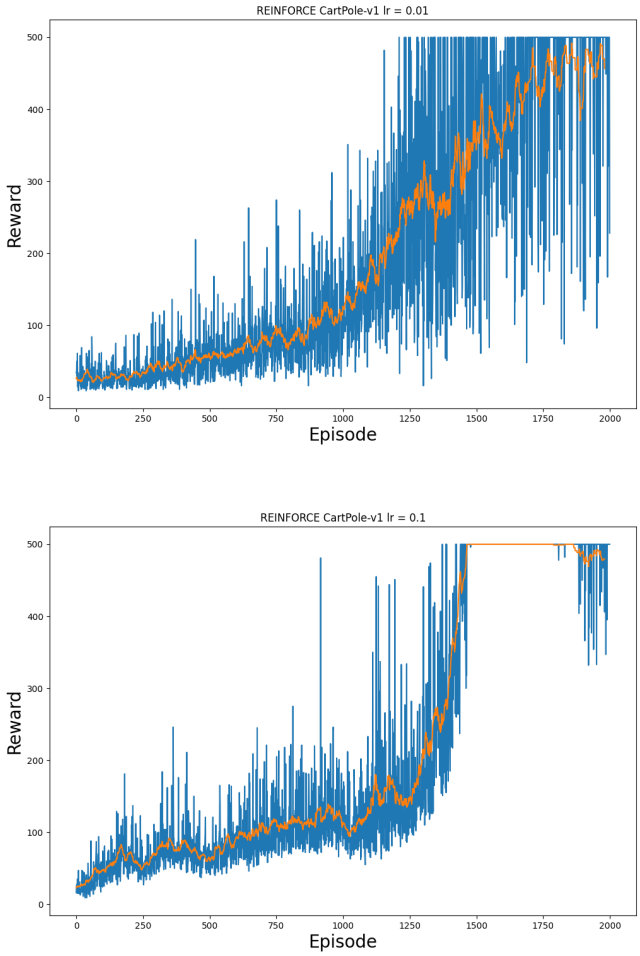
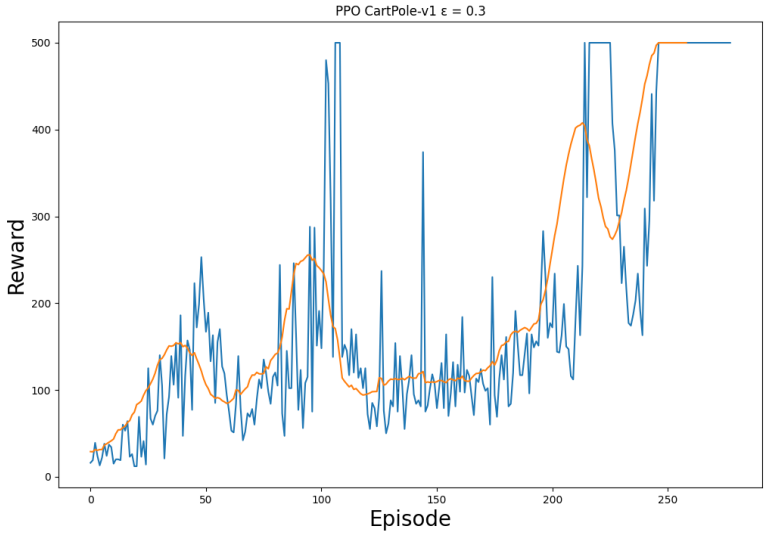


Figure 2: REINFORCE CartPole-v1 with a learning rate of .01 and 1

Despite the second test having a learning rate $\lambda = .1$ that would fail in most applications, the expected reward is increasing almost monotonically. This can be explained by the simplicity of the optimal policy for CartPole and it likely indicates that the reward function space is relatively convex.

PPO PPO is also able to work for CartPole and is even able to converge to an optimal policy in about one-sixth as many episodes.



While PPO's reward function is non-monotonically increasing, PPO's ability to quickly explore the policy space allows it to increase it's reward by 100's of points in very few episodes. Once again, this ability is due to PPO stopping itself from changing its policy too much even though its reward can change drastically.

Acrobot

Acrobot presents more of a challenge since it has an action space with a high-dimensionality relative to CartPole. Also, Acrobot can only attain its maximum reward if it uses the momentum of the links to swing upward (since the actuated joints are not powerful enough on their own to lift the bottom link to the required height). This causes poorly trained Acrobot models to simply flail the Acrobot to maximize the height of the bottom link. However, this sub-optimal policy is a policy "cliff" that traditional Vanilla Policy Gradient algorithms (e.g. REINFORCE) are often unable to recover from.

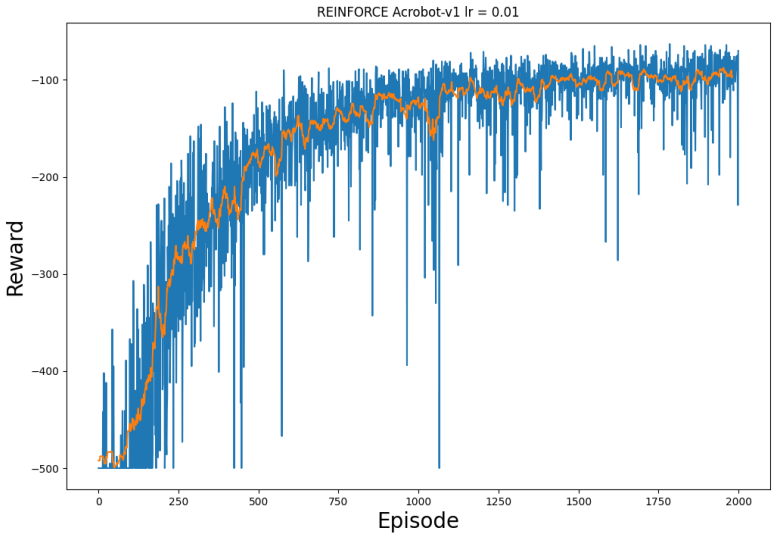


Figure 3: REINFORCE Acrobot-v1 with a learning rate of .01 and 1

Despite REINFORCE being able to converge relatively smoothly on Acrobot for $\lambda = .1$, the REINFORCE algorithm completely failed for higher learning rates (overflow errors during policy updates caused TensorFlow to throw an error).

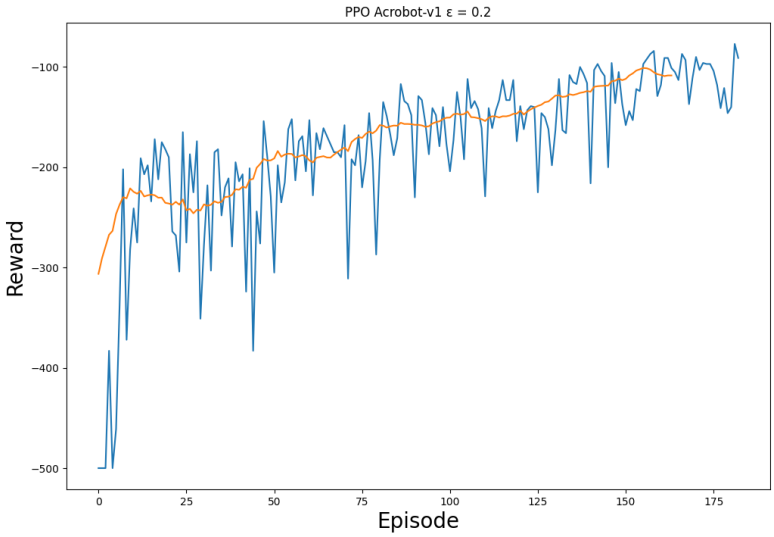


Figure 4: PPO Acrobot-v1 with clip param $\epsilon = 0.03$

PPO Like the CartPole test, PPO was able to converge much faster than REINFORCE for the Acrobot benchmark (about one-twelfth as many steps). Also, PPO from looking at the mean reward calculated from the previous 20 episodes, we can see that PPO was able to converge linearly as opposed to the logarithmic convergence of REINFORCE. Finally, we can observe that PPO had no major decreases to the corresponding reward for its policy whereas REINFORCE had a few episodes where it reward plummeted and if it wasn't for REINFORCE averaging policy changes across multiple episodes, it would have restarted learning from scratch.

Final Observations

Ultimately, while CartPole and Acrobot both have action spaces and reward landscapes that are respectively too constrained and simplistic to show PPO converging where REINFORCE cannot, we are still able to see the benefits of PPO from the reward plots. The important insight is that PPO allows *more* flexibility in policy updates than REINFORCE because it uses the clip function rather than a low of a learning rate to prevent the policy from taking too large a misstep.

References

[1] Prafulla Dhariwal et al. *OpenAI Baselines*. `https://github.com/openai/baselines`. 2017.

[2] Shengyi Huang et al. "CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms". In: *Journal of Machine Learning Research* 23.274 (2022), pp. 1–18. URL: `http://jmlr.org/papers/v23/21-1342.html`.

[3] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: `1312.5602 [cs.LG]`.

[4] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: `1707.06347 [cs.LG]`.

[5] John Schulman et al. *Trust Region Policy Optimization*. 2017. arXiv: `1502.05477 [cs.LG]`.

[6] Richard S. Sutton and Andrew G Barto. *einforcement Learning: An introduction*. 2018.

[7] Eric Yang. Yu. *"Coding PPO from Scratch with Pytorch*. 2020. URL: `https://medium.com/analytics-vidhya/coding-ppo-from-scratch-with-pytorch-part-1-4-613dfc1b14c8`.