I. Introduction

We consider the standard reinforcement learning setting in which an agent interacts with an environment over a discrete number of timesteps. At each timestep t, the agent receives an observation of its current state s_t and selects an action a_t from some set of possible actions A according to its policy π, where π is a mapping from states to actions. The agent moves to the next state s_{t+1} and receives reward r_t. The central problem across many kinds of reinforcement learning (RL) problems is how to take a set of observations and use them to choose the optimal action. At first glance, this may seem identical to the challenge of traditional machine learning, with the goal of fitting a function to the input data in order to minimize error. There are indeed many similarities to traditional "curve fitting", but there is one key feature which sets the RL task apart - time. When an RL agent takes an action, the action's value may not be immediately apparent; instead, the agent's success depends on the many future actions it will take until the agent reaches a terminal state, or the process restarts. A simple regression technique may choose a greedy strategy which maximizes reward one step at a time with no thought for the future, or even fail to converge if the reward is not obtained until the end of the task, leaving the agent with no guidance on the value of its individual actions. For some problems, this obstacle can be overcome by simply modifying the perceived reward. Consider a soccer robot, which instead of just tracking goals scored, might be programmed to understand the laws of physics and simple rules such as "kicking the ball straight to an opponent is bad" or "controlling the ball near the goal is good." However, this approach requires sophisticated domain knowledge and must be redesigned for every new problem. A general learning agent must teach itself to recognize the value of states, in terms of expected total future reward, and this requires increased complexity.

II. Background

A. Q-learning

Q-learning was first introduced in 1989 by Watkins, and is designed to learn the best policy, π^*, which maps every possible state to the best action. This mapping assumes that the environment is Markovian, meaning that the outcome of an action depends only on the current state and the action itself, with perhaps some stochastic noise. For example, it is not sufficient to define the state of a Q-learning self-driving car as its current position plus the position of the surrounding objects; the outcome is strongly dependent on velocity and acceleration also, and these must be included in the state explicitly to eliminate the history-dependence and approximate a Markovian process. The policy is used to choose the action a to move from a state x to a future state y. $a \in A$ is chosen to have the maximum action-value Q under a policy π, given as follows (Watkins and Dayan 1992):

$$Q^{\pi}(x,a) = R_x(a) + \gamma \sum_y P_{xy}[\pi(x)]V^{\pi}(y)$$

$R_x(a)$ is the immediate reward for action a, while $\gamma <= 1$ is a discounting factor for future rewards, useful because a future reward is less certain to occur than an immediate reward. P_{xy} and the summation over possible outcomes y are only necessary if the system is probabilistic instead of deterministic. Finally, $V^{\pi}(y)$ is the state-value discussed in the introduction, and is the key to the system of learning. V is defined recursively, with the value of a state equal to the discounted value of the next state plus the reward gained by changing states. This may seem pointless; how can we determine the state value if doing so requires calculating another state value, repeated infinitely into the future? This is why the agent must be trained. The agent starts with an initial (very inaccurate) value of V for every possible state, and acts to maximize Q, and every time it gains a reward it updates the value of the previous states to reflect their outcome. Knowing the value for each individual state reduces the problem to a traditional machine learning problem, where the actual controller no longer needs to consider further than one timestep into the future, and it is fit to locally optimize Q one step at a time.

Although the previous algorithm works if the agent has explored every possible state, this leads to very slow training and we would like to estimate V for states not yet visited. One way to estimate V is a neural network with weights θ, and a loss function which is the squared error between the predicted and actual value of a state. The Q-network can be updated by backpropagation with stochastic gradient descent. One recent modification of the Q-value neural network is the Deep Q-Network (DQN)(Mnih et al 2013), which stores a long sequence of past experience in memory and trains the Q-network on randomly selected passages from that memory, re-using each state many times to increase efficiency. The DQN takes the state as an input, and has an output node for every possible action giving the value Q for that action, which may require breaking a continuous action space into discrete intervals. Mnih et al. successfully applied a DQN to a variety of different Atari arcade games, and the number of output nodes was the only change needed to the architecture when transferring from one game to the next. After training for 10 million frames for each game (approximately 46 hours at regular game speed) the model was able to outperform human players in simple games such as Pong, and the DQN performed better than other models for complex games such as Space Invaders even though it could not match human performance.

B. Policy Gradient

Recall that the primary goal in reinforcement learning is maximize the sum of expected rewards by learning an optimal policy, π^*, which an agent employs to determine the next action based on the current state of its world. In other words, this policy maps states to actions. Because the state space of a given problem may be huge, we are primarily interested in parametrized representations of policies. While there are many algorithms to learn such a policy, one popular method is the policy gradient algorithm.

Consider a deterministic policy, π_θ, in a deterministic environment. If we let $J(\theta)$ be a closed form expression representing the reward when π_θ is executed, then we can use standard optimization procedures to optimize our reward function. We then define the policy gradient vector as $\nabla_\theta J(\theta)$. If $J(\theta)$ is differentiable, we can express the policy gradient in closed form. Otherwise, we can follow the empirical gradient by hill climbing to achieve an approximation for the policy gradient. In the case of a stochastic policy, which maps states to a probability distribution, and stochastic environment, we can represent the policy gradient as

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})$$

In which for trajectory or rollout t, we can consider the state $s_{i,t}$ and action $a_{i,t}$ at the ith step of the trajectory according to π_θ and receive reward $r(s_{i,t}, a_{i,t})$. We can use the policy gradient to tell us how we should improve our policy such that we might converge to a local minima. Observe how this algorithm is very similar to gradient descent in structure. We define the policy gradient algorithm (Williams 1992) as the following

procedure REINFORCE
Sample from π_θ
$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})$
$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

C. Actor-Critic

The two primary reinforcement learning methods are value-based methods, such as Q-Learning or Deep Q-Learning, and policy-based methods, such as policy gradients. Recall that value-based methods learn a value function that map each state action pair to a value, and we calculate our policy by taking the action that with maximum value. Policy-based methods directly learn such a policy without a value function. The actor-critic algorithm is a hybrid method that uses both value-based and policy-based methods. More specifically, the actor-critic algorithm uses two controllers: a policy-based actor that controls how an agent behaves in its environment, and a value-based critic that measures how good the action taken is and how the actor should adjust its policy. We can define an advantage function that defines how much better an action was than expected

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

The actor-critic algorithm trains the actor and critic in parallel. The actors parameters are updated in a very similar way to the vanilla policy gradient algorithm, but the update step uses the advantage function as a scalar multiplier of the policy gradient. The critic is trained to directly minimize the advantage function. The actor-critic paradigm helps reduce the variance of policy gradient.

III. Dueling Deep Q-Network

Deep Q-Learning, the new version of Q-Learning with neural network fitting in was first introduced in 2014 and since then, a lot of improvements have been made. Especially, one version of the alternatives, Dueling DQN, leads to better policy evaluation if we are given many similar-valued actions and outperforms the state of the art on the Atari game with 2600 domain[**?**]. Q-values correspond to how good the agent is to be at the state and taking an action at the state and the Q-value at the given fixed state is denoted by $Q(s,a)$. The main idea of Dueling DQN is that it decomposes $Q(s,a)$ as $V(s)$, the summation of the value of staying at that state and $A(s,a)$, the advantage of taking action a at the state. Intuitively, the advantage function subtracts the value of the state from the Q function to obtain a relative measure of the importance of each single action given that the agent stay at that state. Suppose we are given a fixed policy π, then

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

The key insight behind the new architecture is that for some states, it is unnecessary to estimate the value of each action choice; However for other states, it is of paramount importance to know which action to take[**?**]. In the prosthetic project, each action vector with dimension of 19 is less important to think about if the values of the different actions are very similar. On the dueling architecture, it can learn which states are valuable without having to the learn the effect of any action. The reason Dueling Deep Q-networking is useful in our project is that the agent might not need to take any additional action in the state when the reward is high. No action needs to be taken at the moment when the skeletal muscles are supposed to

be relaxed.Therefore, when we design our convolutional neural networks for the prosthetic project, instead of following the convolutional layers with a single sequence of fully connected layers, we use two sequences of fully connected layers. Then the two sequences are aggregated together and make the output Q. If we aggregate the stream using

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

we will fall into the issue of identifiability[**?**] that we cannot find $A(s,a)$ and $V(s)$ and it is troublesome for gradient descent. To avoid that, Dueling DQN will use

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{A}\sum_{a'} A(s,a';\theta,\alpha))$$

which is subtracting the average advantage of all actions in that state so that the Duel DQN will accelerate the training and help use find more reliable Q values for each action. In our implementation, keeping the same convolutional neural networks, we will modify the DQN architecture by only adding new sequences (one is for value and one is for advantage) and the aggregating layer.

IV. Proximal Policy Optimization Algorithms

The main purpose for applying Proximal Policy Optimization (PPO) is that it strikes a balance among easy implementation, sample efficiency and easy to tune. Unlike DQN which only learns form stored offline data, proximal policy optimization learns online. For off-policy algorithms like DQN, the agent cannot pick up actions and it will learn via exploration but play without exploration. For proximal policy optimization, it will follow its own policy and pick up its own actions so that it will directly learn from whatever if it encounters. Since policy gradient methods will only use the collect experience once for updating, policy gradient methods are less sample efficient than DQN. The most commonly used general policy optimization methods usually start by defining the policy gradient laws as

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t]$$
$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t]$$

π_θ is the policy that it takes it takes the observed states from the environment and makes the action to take. $\hat{A}_t$ is basically trying to estimate what the relative value of the selected action. We call it advantage estimate which tells us how much better was the action that we take based on the expectation of what will normally happen in the state. In short, $\hat{A}_t$ tells us whether the action is better than expected or not. $\hat{\mathbb{E}}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t]$ will be the final optimization object that is used in policy gradient.Therefore, intuitively if the $\hat{A}_t$ is positive, gradient will be positive and it will increase the likelihood to choose the action. But if we keep running gradient descent, the parameter in the neural network will be updated outside of the range. So we have to make sure that the policy will never be updated too far away from the old policy. So within introducing Trust Region Methods (TRPO)[**?**], we modify the objective function to

$$\max \hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{A}_t]$$

Moreover TRPO adding a KL constraint which effectively guarantees that the new updated policy is close to the old one. However, the KL constraint just adds overhead to the optimization and will lead to some bad training behaviors. Therefore PPO will overcome the problem above. Let $r_t\theta = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$, TRPO will be

$$\max \hat{\mathbb{E}}_t[r_t\theta\hat{A}_t]$$

so the TRPO is more readable. Then we penalize changes to the polcy that move $r_t\theta$ away from 1 then main object function will be

$$\max \hat{\mathbb{E}}_t[\min(r_t\theta\hat{A}_t), clip(r_t\theta, 1-\epsilon, 1+\epsilon)\hat{A}_t]$$

and the epsilon is a hyper-parameter. $r_t\theta\hat{A}_t$ is the default objective for normal TRPO but $clip(r_t\theta, 1-\epsilon, 1+\epsilon)\hat{A}_t$ is a truncated version of the first term so that the clipped term will limit the effect of the gradient update for advantage larger than 0[**?**]. Basically, PPO does the same as TRPO but it forces the policy updates to be conservative if it moves so far away from the current policy and moreover, PPO always outperforms. In general, PPO method have the stability and reliability of trust-region methods but are much simpler to implement[**?**]. Based on the continuous experiment given by this paper,PPO outperforms the previous methods on almost all the continuous control environments. Due to the fact that our training agent is take action continuously, we will apply this method in our implementation for training.

V. Deep Deterministic Policy Gradient

The traditional policy gradient algorithm suffers from high variance and slow convergence. Furthermore, it may be difficult to choose an appropriate learning rate for the given problem. Additionally, while algorithms such as DQN can solve problems with high-dimensional observation spaces, they cannot handle high-dimensional action spaces. Lilicrap et. all introduce a model-free off-policy actor-critic algorithm called Deep Deterministic Policy Gradient (DDPG) that addresses such challenges.

DDPG is a policy gradient algorithm that uses a stochastic policy for exploring the state space to estimate a deterministic target policy. DDPG adds a noise parameter to the actor policy to enhance agent exploration: $\mu^{'}(s_t) = \mu(s_t|\theta_t^{\mu}) + \mathcal{N}$. One common choice for this noise parameter is the Ornstein-Uhlenbeck Random Process. The actor network, $\mu(s|\theta)$, receives the current state space as the input and outputs a single real value from a continuous action space. The actor policy is updated using the sampled policy gradient

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N} \sum_{i} \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s_i}$$

The critic network, $Q(s, a|\theta^Q)$, receives the current state space and the actors choice of action as input and outputs the estimated Q-value of the provided action. The critic is updated by minimizing the loss

$$L = \frac{1}{N} \sum_{i} (y_i - Q(s_i, a_i|\theta^Q))^2$$

Observe that DDPG randomly samples from the replay buffer to calculate approximations of the gradients in an attempt to decorrelate training samples. Furthermore, DDPG uses target networks to create more stable training rollouts. The target networks are updated accordingly

$$\theta^{Q^{'}} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q^{'}}$$

$$\theta^{\mu^{'}} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu^{'}}$$

VI. Asynchronous Advantage Actor-Critic

Although utilizing a replay buffer decorrelates updates, it limits methods to off-policy reinforcement learning algorithms because the data must be batched or randomly sampled. Furthermore, the replay buffer uses significant memory and computation. Mnih et all. introduce an asynchronous advantage actor critic algorithm (A3C) to help overcome these problems.

Recall that in DQN, a single agent represented by a single neural network controller interacts with a single environment. A3C is asynchronous: it utilizes multiple agents via a single global neural network controller and each agent has its own set of network parameters. Each agent is independent of one another, which decorrelates the actors and increases the diversity of training the networks because each agent is likely to be exploring a different part of the environment. Asynchronous agents eliminate the need for a replay buffer, which allows models to pursue on-policy learning algorithms to train neural networks, such as the actor-critic algorithm. A3C employs an actor-critic paradigm to train two neural networks: the actor that estimates policy and the critic that estimates a value function. The update performed by the algorithm can be seen as

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta^{'}) A(s_t, a_t; \theta; \theta_v)$$

where $A(s_t, a_t; \theta; \theta_v)$ is an estimate of the advantage function given by $\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$ where θ and θ_v are the parameters of the policy and value functions, respectively. The resulting architecture and training of A3C agents yields much more stable neural networks with both value-based and policy-based methods, on-policy and off-policy methods, as well as discrete and continuous domains. Because of its extreme flexibility, fast training time, and model stability, A3C has become the start of the art, go-to starting point for many deep reinforcement learning problems.

References

[1] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[1] [**?**] [**?**]