Policy Gradients

Joseph Engler, PhD

Up to this point we have considered reinforcement learning algorithms that were calculating a Value Function and worked in an On-line format. Recall that reinforcement learning happens in one of two standard states, On-line learning in which the value function and hence the policy are updated with each step of the run, and Off-line in which learning occurs after termination of the run. This chapter focuses on an Off-line learning technique that seeks to optimize the agent's policy rather than a specific value such as the Q-value. This forms the first foray into the second of two main approaches for Model-Free reinforcement learning. The two main approaches for Model-Free RL are Q-learning, covered previously, and Policy Gradient methods, the topic of this, and subsequent chapters.

There exist three broad categories of policy optimization algorithms (Schulman, Levine, Moritz, Jordan, & Abbeel, 2015). The first is policy iteration methods such as Q-learning which alternate between estimating a value function under the current policy and improving the policy. The second category consists of, and are the focus of this chapter, policy gradient algorithms which use sample trajectories to estimate the gradient of the expected cost. The final category consists of derivative-free optimization methods that will not be covered in this text (Schulman et al., 2015). To further illustrate the various forms of RL algorithms, Figure 1 gives a visual breakdown of the major algorithms currently in use.

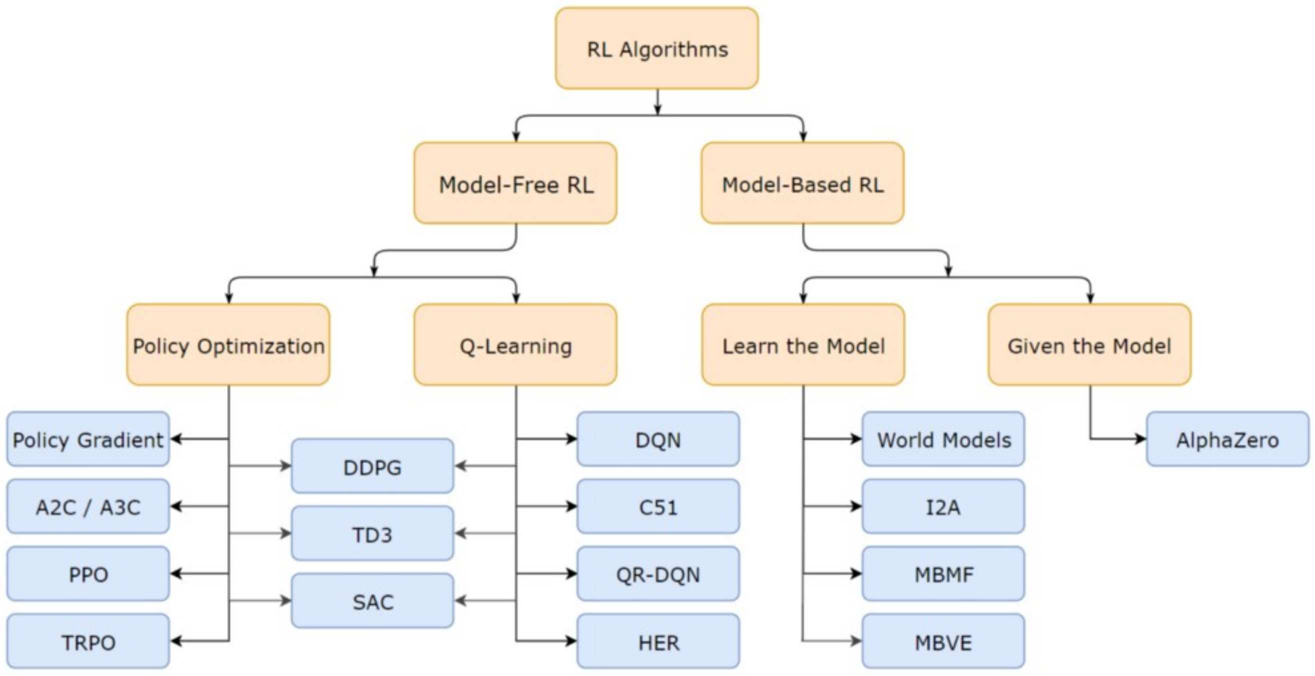

Figure 1: Breakdown of the major RL algorithms currently in use.

Policy Gradient methods are not an entirely new subject in RL. Policy gradients were discussed in detail by (Sutton, McAllester, Singh, & Mansour, 2000) and in 2004 they were being explored for controlling various robotic motion (Kohl & Stone, 2004). This is not to indicate, however, that Policy Gradients are an old or out of date technique. Quite the opposite is true as Policy Gradient research continues and may be found discussed even in recent journal articles (Wu & Li, 2020) and conferences (Li et al., 2019).

As we have previously discussed the goal of reinforcement learning is to learn a behavior policy which is optimality expressed via maximal reward. In Deep Q-learning, this optimization was achieved via value function approximation which was expressed as a parameterized learning problem (e.g. parameterizing the value function via the weights of a neural network) Policy Gradients also seek to achieve an optimal agent behavior. However, the method by which this is achieved is to optimize the policy directly rather than optimizing the value function and then inferring the policy.

As with our previous work, let $\pi(a|s)$ represent the policy which assigns the probabilities of taking action a while in the state s. Then the goal of the Policy Gradient algorithm is to find a policy, π, parameterized by a the weights, θ, of a deep learning network that chooses the actions to take in a given trajectory, τ, that maximizes the expected rewards, r, at each step in the trajectory. That is a wordy and technical way of saying that the algorithm seeks to optimize an action policy so that when the policy is applied to a complete run in the environment the maximum expected rewards are obtained.

The most basic form of Policy Gradient RL is given in the REINFORCE algorithm (Williams, 1992) which is also termed “Monte Carlo” gradient estimation (Silver & Tesauro, 2009). In this variant, the algorithm initializes a policy at random, runs sample trajectories of the environment, and updates the policy based upon the rewards obtained by each trajectory. Figure 2 illustrates this methodology.

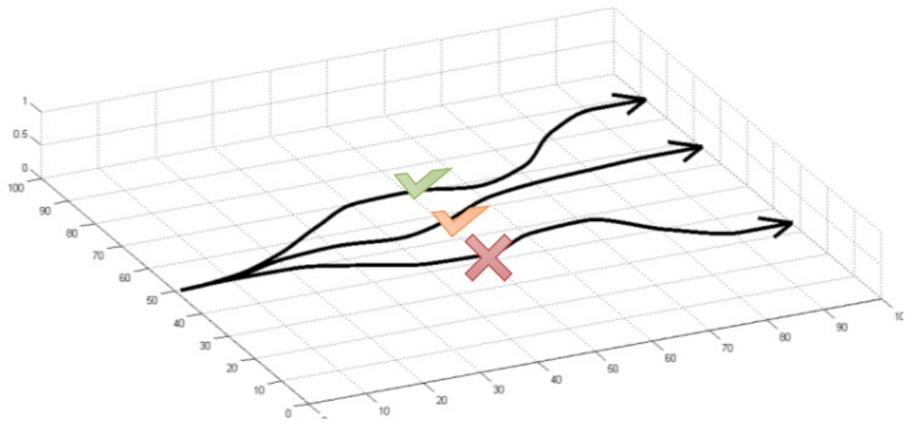

Figure 2: Monte Carlo gradient estimation example. Taken from Levine (http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-5.pdf)

Note, that the policy is not updated after N steps as with Q-learning where N=1, rather the policy is updated upon the termination of the trajectory (run) in the environment. Thus, we can define a trajectory as a series of state-action pairs as in Eq. 1 below, where *H* is the time horizon of, or the maximum number of steps taken in, the trajectory at the run's end.

$$\tau=(s_{0,}a_{0,}s_{1,}a_{1,}s_{2,}a_{2,}\ldots,s_H,a_H) \qquad \text{Eq. 1}$$

Inherent in the statement above that the policy is updated based on rewards obtained by the trajectory, τ, is the idea of optimality. To achieve the optimality of expected rewards, the goal of RL as stated above, an objective function is utilized. The objective function used in Policy Gradients is based on the sum of rewards obtained by the trajectory, τ, given by the policy, π, parameterized via the network weights, θ, as given in Eq. 2.

$$J(\theta)=\mathrm{E}[\sum_{t=0}^{H}R(s_t,a_t);\pi_\theta] \qquad \text{Eq. 2}$$

It is important to note a few equalities found in Eq. 2. First, the policy, π, gives the probabilities of actions, *a*, to take in a given state, *s* when π is parameterized by θ. Therefore, for a given trajectory, τ, following π, Eq. 3. holds true. Additionally, we can consider the sum in Eq. 2 as representing the entire trajectory, τ, and therefore can rewrite Eq. 2 as given in Eq. 4 below.

$$\pi_\theta=p(\tau;\theta) \qquad \text{Eq. 3}$$

$$J(\theta)=\sum_{\tau}p(\tau;\theta)R(\tau) \qquad \text{Eq. 4}$$

With the objective function defined above, we can now turn to the process of optimization. Just as with Q-Learning, optimization is simply a maximization of the objective function. Thus, optimization on Policy Gradients seeks to find the maximum of the objective function as given in Eq. 5.

$$\max_{\theta}J(\theta)=\max_{\theta}\sum_{\tau}p(\tau;\theta)R(\tau) \qquad \text{Eq. 5}$$

This optimization is achieved via taking the gradient of Eq. 4 and adding a small portion of that gradient to the existing parameterization, θ, in a manner explicitly similar to back propagation in deep learning and given by Eq. 6.

$$\theta=\theta+\alpha\nabla J(\theta) \qquad \text{Eq. 6}$$

To effectively achieve the optimization of the policy as shown in the above equations we will utilize sampling. In the case of the algorithm under consideration herein, we can combine a number of the equations above to rewrite the objective function under sampling as given in Eq. 7 where *i* is an arbitrary starting point in a trajectory and $p(s_t, a_t|\tau)$ is the probability of the occurrence of s_t, a_t given the trajectory τ.

$$J(\theta)=\mathrm{E}[\sum_{t=0}^{H} r_{t+1}|\pi_\theta]=\sum_{t=i}^{H} p(s_t, a_s|\tau) r_{t+1} \qquad \text{Eq. 7.}$$

If we differentiate both sides of Eq. 7 with respect to the policy parameter, θ, using Eq. 8 we obtain the derivation given by Eq. 9 -12.

$$\frac{d}{dx}\log f(x)=\frac{f'(x)}{f(x)} \qquad \text{Eq. 8}$$

$$\nabla_\theta J(\theta)=\sum_{t=i}^{H-1} \nabla_\theta p(s_t, a_t|\tau) r_{t+1} \qquad \text{Eq. 9}$$

$$\nabla_\theta J(\theta)=\sum_{t=i}^{H-1} p(s_t, a_t|\tau)\frac{\nabla_\theta p(s_t, a_t|\tau)}{p(s_t, a_t|\tau)} r_{t+1} \qquad \text{Eq. 10}$$

$$\nabla_\theta J(\theta)=\sum_{t=i}^{H-1} p(s_t, a_t|\tau)\nabla_\theta \log p(s_t, a_t|\tau) r_{t+1} \qquad \text{Eq. 11}$$

$$\nabla_\theta J(\theta)=\mathrm{E}[\sum_{t=i}^{H-1} \nabla_\theta \log p(s_t, a_t|\tau) r_{t+1}] \qquad \text{Eq. 12}$$

Since Eq. 12 is an expectation, we can utilize sampling during learning, thus we can remove the expectation in Eq. 12. as given in Eq. 13.

$$\nabla_\theta J(\theta)\sim\sum_{t=i}^{H-1} \nabla_\theta \log p(s_t, a_t|\tau) r_{t+1} \qquad \text{Eq. 13}$$

Finally, we can separate the right hand side of Eq. 13 to give Eq. 14.

$$\nabla_\theta J(\theta)\sim\sum_{t=i}^{H-1} \nabla_\theta \log p(s_t, a_t|\tau)\sum_{t=i}^{H-1} r_{t+1} \qquad \text{Eq. 14}$$

The reason for the separation given in Eq. 14 is that it allows for an intuitive understanding of what is taking place in the policy gradient. The first sum in Eq. 14 actually represents the log likelihood, or in general the likelihood, of the observed trajectory. In other words, how likely is the given trajectory under the policy parameterized by θ. Multiplying the log likelihood of the trajectory by the reward obtained for that trajectory strengthens good trajectories in the learning. Finally, recalling that $p(s_t, a_t|\tau)$ is simply the probabilities given by the policy π_θ for τ, we can consider the gradient of the objective function to be estimated by the average of a number of trajectories as given in Eq. 15.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{H-1} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \sum_{t=1}^{H-1} r(s_{i,t}, a_{i,t}) \qquad \text{Eq. 15}$$

Using Eq. 15 above, we can now fully describe the most basic variant of Policy Gradient algorithms, the REINFORCE algorithm, As stated previously, this algorithm utilizes Monte Carlo rollouts to compute the total rewards and then optimizes the policy using Eq. 6. The pseudo-code for the REINFORCE algorithm is given below.

```
REINFORCE:
while True:
        Sample {τ^i} from $\pi_\theta(a_t|s_t)$   (run the current policy)
        $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} ( \sum_{t=1}^{H-1} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) ) \sum_{t=1}^{H-1} r(s_{i,t}, a_{i,t})$
        $\theta = \theta + \alpha \nabla J(\theta)$
```

As can be seen from the pseudo-code above, the REINFORCE algorithm is quite simplistic. It has also shown to be very effective at learning new environments and is often capable of doing so at a faster rate than value iteration type algorithms such as Q-Learning. Furthermore, using the stochastic Monte Carlo style rollouts means that Policy Gradients do not express bias. However, Policy Gradients do have a drawback; they suffer from high variance. It can be easily seen that as the algorithm runs it will select various types of trajectories, some valuable and some not so valuable. These divergent samples can cause conflicts in descent direction and have been shown to hurt convergence rates.

A number of remedies exist for assisting with this variance. One means by which reducing the variance, incurred by the actions taken, is to increase the batch size when training. However, this will only reduce the variance slightly. The best method shown to date for reducing the variance in Policy Gradient algorithms is to utilize a baseline. In the equations above, we can, without penalty, subtract a term as long as that term is not related to the parameters of optimization, θ. Thus, let us define a term *b* as given in Eq. 16.

$$b = \frac{1}{N} \sum_{i=1}^{N} r(\tau) \qquad \text{Eq. 16}$$

Then we can define a term $A_t = r(s_t, a_t) - b$ and substitute it into Eq. 15 to arrive at Eq. 17. We call this term the advantage term as it will clearly indicate whether or not the trajectory taken resulted in a learning advantage over the baseline.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{H-1} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \sum_{t=1}^{H-1} A_t \qquad \text{Eq. 17}$$

To ensure that the baseline is serving to reduce variance as much as possible, the baseline is updated after each training step as given in Eq. 18.

$$b = \|b(s_t) - R_t\|^2 \qquad \text{Eq. 18}$$

To conclude this chapter, an implementation of the REINFORCE algorithm is given below for the Cart-Pole environment that was used in previous chapters. The implementation is again given in python and is given in a single file format.

```
import sys
import gym
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
import tensorflow as tf
import random
EPISODES = 1000


# This is Policy Gradient agent for the Cartpole
# In this example, we use REINFORCE algorithm which uses monte-carlo update rule
class REINFORCEAgent:
    def __init__(self, state_size, action_size):
        # if you want to see Cartpole learning, then change to True
        self.render = True
        self.load_model = False
        # get size of state and action
        self.state_size = state_size
        self.action_size = action_size

        # These are hyper parameters for the Policy Gradient
        self.discount_factor = 0.99
        self.learning_rate = 0.01
        self.hidden1, self.hidden2 = 50, 24

        # create model for policy network
        self.model = self.build_model()

        # lists for the states, actions and rewards
        self.states, self.actions, self.rewards = [], [], []
        self.min_eps = 0.01
        self.eps = 0.99

        if self.load_model:
            self.model.load_weights("./save_model/cartpole_reinforce.h5")

    # approximate policy using Neural Network
    # state is input and probability of each action is output of network
    def build_model(self):
        model = Sequential()
        model.add(Dense(self.hidden1, input_dim=self.state_size, activation='relu', kernel_initializer='glorot_uniform'))
        model.add(Dense(self.hidden2, activation='relu', kernel_initializer='glorot_uniform'))
        model.add(Dense(self.action_size, activation='softmax', kernel_initializer='glorot_uniform'))
        model.summary()
       model.compile(loss=self.categorical_crossentropy, optimizer=Adam(lr=self.learning_rate))
        return model

    def categorical_crossentropy(self, target, output):
        _epsilon =  tf.convert_to_tensor(10e-8, dtype=output.dtype.base_dtype)
        output = tf.clip_by_value(output, _epsilon, 1. - _epsilon)
        return (- target * tf.math.log(output))

    # using the output of policy network, pick action stochastically
    def get_action(self, state):
        policy = self.model.predict(state, batch_size=1).flatten()
        return np.random.choice(self.action_size, 1, p=policy)[0]


   def discount_rewards(self, rewards):
        discounted_rewards = np.zeros_like(rewards)
        running_add = 0
        for t in reversed(range(0, len(rewards))):
            running_add = running_add * self.discount_factor + rewards[t]
            discounted_rewards[t] = running_add
        return discounted_rewards

    # save <s, a ,r> of each step
    def append_sample(self, state, action, reward):
        self.states.append(state)
        self.rewards.append(reward)
        self.actions.append(action)

    # update policy network every episode
    def train_model(self):
        episode_length = len(self.states)

        discounted_rewards = self.discount_rewards(self.rewards)
        discounted_rewards -= np.mean(discounted_rewards)
        discounted_rewards /= np.std(discounted_rewards)

        update_inputs = np.zeros((episode_length, self.state_size))
        advantages = np.zeros((episode_length, self.action_size))

        for i in range(episode_length):
            update_inputs[i] = self.states[i]
            advantages[i][self.actions[i]] = discounted_rewards[i]

        self.model.fit(update_inputs, advantages, epochs=1, verbose=0)
        self.states, self.actions, self.rewards = [], [], []

if __name__ == "__main__":
    # In case of CartPole-v1, you can play until 500 time step
    env = gym.make('CartPole-v1')
    # get size of state and action from environment
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    # make REINFORCE agent
    agent = REINFORCEAgent(state_size, action_size)
```

```
scores, episodes = [], []

for e in range(EPISODES):
    done = False
    score = 0
    state = env.reset()
    state = np.reshape(state, [1, state_size])

    while not done:
        if agent.render:
            env.render()

        # get action for the current state and go one step in environment
        action = agent.get_action(state)
        next_state, reward, done, info = env.step(action)
        next_state = np.reshape(next_state, [1, state_size])
        reward = reward if not done or score == 499 else -100

        # save the sample <s, a, r> to the memory
        agent.append_sample(state, action, reward)

        score += reward
        state = next_state

        if done:
            # every episode, agent learns from sample returns
            agent.train_model()

            # every episode, plot the play time
            score = score if score == 500 else score + 100
            scores.append(score)
            episodes.append(e)
            plt.plot(episodes, scores, 'b')
            plt.show()
            #pylab.savefig("./save_graph/cartpole_reinforce.png")
            print("episode:", e, "  score:", score, " eps:", agent.eps)

            # if the mean of scores of last 10 episode is bigger than 490
            # stop training
            if np.mean(scores[-min(10, len(scores)):]) > 490:
                sys.exit()
```

Works Cited

Kohl, N., & Stone, P. (2004). Policy gradient reinforcement learning for fast quadrupedal locomotion. *Proceedings - IEEE International Conference on Robotics and Automation*, *2004*(3), 2619–2624. https://doi.org/10.1109/robot.2004.1307456

Li, S., Wu, Y., Cui, X., Dong, H., Fang, F., & Russell, S. (2019). Robust Multi-Agent Reinforcement Learning via Minimax Deep Deterministic Policy Gradient. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*, 4213–4220. https://doi.org/10.1609/aaai.v33i01.33014213

Schulman, J., Levine, S., Moritz, P., Jordan, M., & Abbeel, P. (2015). Trust region policy optimization. *32nd International Conference on Machine Learning, ICML 2015*, *3*, 1889–1897.

Silver, D., & Tesauro, G. (2009). Monte-Carlo simulation balancing. *Proceedings of the 26th International Conference On Machine Learning, ICML 2009*, 945–952.

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Nerual Information Processing Systems*, 1057–1063.

Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, *8*(3), 229–256. https://doi.org/10.1023/A:1022672621406

Wu, J., & Li, H. (2020). Deep Ensemble Reinforcement Learning with Multiple Deep. *Mathematical Problems in Engineering*, *2020*.