# Assignment 9 by Haobin Tang

## 1.Write Proof (with precise notation) of the Policy Gradient Theorem

So far, in order to learn a policy, we have focused on value-based approaches where we nd the optimal state value function or state-action value function with parameters $\theta$,

$$V_\theta(s) = V^\pi(s)$$

In this setting, our goal is to directly nd the policy with the highest value function $V^\pi$, rather than rst nding the value-function of the optimal policy and then extracting the policy from it. Instead of the policy being a look-up table from states to actions, we will consider stochastic policies that are parameterized. Finding a good policy requires two parts:

1. Good policy parameterization: our function approximation and state/action representations must be expressive enough
2. Erffective search: we must be able to nd good parameters for our policy function approximation

Policy-based RL has a few advantages over value-based RL:

1. Better convergence properties
2. Effectiveness in high-dimensional or continuous action spaces, e.g. robotics.
3. Ability to learn stochastic policies.

The disadvantages of policy-based RL methods are:

1. They typically converge to locally rather than globally optimal policies, since they rely on gradient descent.
2. Evaluating a policy is typically data ine cient and high variance.

I use the sum notation for simplicity to prove instead of integral.

In an episodic environment, a natural measurement is the start value of the policy, which is the expected value of the start state:

$$J_1(\theta) = V^{\pi\theta}(s) = \mathbb{E}_{\pi\theta}[v_1]$$

Let us define $V(\theta)$ to be the objective function we wish to maximize over $\theta$. Policy gradient methods search for a local maximum in $V(\theta)$ by ascending the gradient of the policy, w.r.t parameters $\theta$

$$\Delta\theta = \alpha\nabla_\theta V(\theta)$$

Let us set the objective function $V(\theta)$ to be the expected rewards for an episode,

$$V(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi\theta}[\sum_{t=0}^{T} R(s_t, a_t)] = \mathbb{E}_{\tau\sim\pi\theta}[R(\tau)] = \sum_\tau P(\tau;\theta)R(\tau)$$

where $\tau$ is a trajectory,

$$\tau = ((s_0, a_0, r_0, \cdots, s_{T-1}, a_{T-1}, r_{T-1}, s_T))$$

If we can mathematically compute the policy gradient $\nabla_\theta\pi_\theta(a|s)$, then we can go right ahead and compute the gradient of this objective function with respect to $\theta$:

$$\begin{aligned}
\nabla_\theta V(\theta) &= \nabla_\theta \sum_\tau P(\tau;\theta)R(\tau) \\
&= \sum_\tau \nabla_\theta P(\tau;\theta)R(\tau) \\
&= \sum_\tau \frac{P(\tau;\theta)}{P(\tau;\theta)}\nabla_\theta P(\tau;\theta)R(\tau) \\
&= \sum_\tau P(\tau;\theta)\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)}R(\tau) \\
&= \sum_\tau P(\tau;\theta)R(\tau)\nabla_\theta log(P(\tau;\theta)) \\
&= \mathbb{E}_{\tau\sim\pi\theta}[R(\tau)\nabla_\theta log(P(\tau;\theta))]
\end{aligned}$$

Second, computing $\nabla_\theta log(P(\tau;\theta))$

$$\begin{aligned}\nabla_\theta log(P(\tau;\theta)) &= \nabla_\theta log[\mu(s_0)\prod_{t=0}^{T-1}\pi_\theta(a_t|s_t)P(s_{t+1}|a_t|s_t)] \\ &= \nabla_\theta[log\mu(s_0)\sum_{t=0}^{T-1}log\pi_\theta(a_t|s_t) + logP(s_{t+1}|a_t|s_t)] \\ &= \sum_{t=0}^{T-1}\nabla_\theta log\pi_\theta(a_t|s_t)\end{aligned}$$

Working with $log(P(\tau;\theta))$ instead of $(P(\tau;\theta))$ allows us to represent the gradient without reference to the initial state distribution, or even the environment dynamics model!

For now we get

$$\nabla_\theta V(\theta) = \mathbb{E}_{\pi\theta}[R(\tau)\nabla_\theta log\pi_\theta(a|s)]$$

Theorem For any di erentiable policy $\pi_\theta(a|s)$ and for any of the policy objective functions, the policy gradient is

$$\nabla_\theta V(\theta) = \mathbb{E}_{\pi\theta}[Q^{\pi\theta}(a|s)\nabla_\theta log\pi_\theta(a|s)]$$

Notice that the rewards $R(\tau^{(i)})$ are treated as a single number which is a function of an entire trajectory $\tau^{(i)}$. We can break this down into the sum of all the rewards encountered in the trajectory,

$$R(\tau) = \sum_{t=0}^{T-1} R(s_t, a_t)$$

Using this knowledge, we can derive the gradient estimate for a single reward term $r_t{}'$ in exactly the same way we derived equation :

$$\nabla_\theta\mathbb{E}_{\pi\theta}[r_t{}'] = \mathbb{E}_{\pi\theta}[r_t{}'\sum_{t=0}^{t'}\nabla_\theta log\pi_\theta(a_t|s_t)]$$

Since $\sum_{t'=t}^{T-1} r_{t'}$ is the return $G_t$, we can sum this up over all time steps for a trajectory to get

$$\begin{aligned}\nabla_\theta V(\theta) &= \mathbb{E}_{\tau\sim\pi\theta}[R(\tau)] \\ &= \mathbb{E}_{\pi\theta}[\sum_{t'=0}^{T-1} r_{t'}\sum_{t=0}^{t'}\nabla_\theta log(\pi_\theta(a_t|s_t)] \\ &= \mathbb{E}_{\pi\theta}[\sum_{t=0}^{T-1}\nabla_\theta log(\pi_\theta(a_t|s_t)\sum_{t'=t}^{T-1} r_{t'}] \\ &= \mathbb{E}_{\pi\theta}[\sum_{t=0}^{T-1}\nabla_\theta log(\pi_\theta(a_t|s_t)G_t]\end{aligned}$$

The main idea is that the policy's choice at a particular time step $t$ only a ects rewards received in later steps of the episode, and has no effect on rewards received in previous time steps. Our original expression in the above equation did not take this into account.

## 2.Derive the score function for softmax policy (for finite set of actions)

The expression $log(\pi_\theta(a|s))$ is known as the score function. The $\nabla_\theta\pi_\theta(a|s)$ for finite action spaces is

$$\nabla_\theta\pi_\theta(a|s) = \frac{e^{\theta^T * \phi(s,a)}}{\sum_b e^{\theta^T * \phi(s,b)}}$$

$$\begin{aligned}\nabla_\theta log(\pi_\theta(a|s)) &= \nabla_\theta(\theta^T * \phi(s,a) - \sum_b \theta^T * \phi(s,b)) \\ &= \phi(s,a) - \frac{\sum_b \phi(s,b) * e^{\theta^T * \phi(s,b)}}{\sum_b e^{\theta^T * \phi(s,b)}} \\ &= \phi(s,a) - \sum_b \pi_\theta(b|s) * \phi(s,b)\end{aligned}$$

We also can derive

$$\mathbb{E}_b[log(\pi_\theta(a|s))] = 0$$

## 3.Write code for the REINFORCE Algoithm (Monte-Carlo Policy Gradient Algorithm, i.e., no Critic)

In [2]:
```python
#This is a REINFORCE Algorithm framework I wrote as homework in CS234 using TensorFlow.

class REINFORCE(object):
    def __init__(self, env, logger=None):
        self.batch_size=1000
        self.episode_max_length=10
        self.num_seq_per_batch=50
        self.gamma=0.9
    def add_placeholders_op(self):
        """
        Add placeholders for observation, action, and advantage:
            self.observation_placeholder, type: tf.float32
            self.action_placeholder, type: depends on the self.discrete
            self.advantage_placeholder, type: tf.float32
        """
        self.observation_placeholder = tf.placeholder(tf.float32, shape = [None, self.observation_
dim])
        self.action_placeholder = tf.placeholder(tf.int64, shape = [None,])
        self.advantage_placeholder = tf.placeholder(tf.float32, shape = [None,])
    def build_policy_network_op(self, scope = "policy_network"):
        """
        Build the policy network, construct the tensorflow operation to sample
        actions from the policy network outputs, and compute the log probabilities
        of the actions taken (for computing the loss later). These operations are
        stored in self.sampled_action and self.logprob. Must handle both settings
        of self.discrete.

        Args:
              scope: the scope of the neural network
        """
        action_logits = build_mlp(self.observation_placeholder, self.action_dim, scope, self.pa.n_
layers, self.pa.layer_size)
        self.sampled_action = tf.squeeze(tf.multinomial(action_logits, 1), axis = 1)
        self.logprob = - tf.nn.sparse_softmax_cross_entropy_with_logits(labels = self.action_place
holder, logits = action_logits)
    def add_loss_op(self):
        """
        Compute the loss, averaged for a given batch.
        The update for REINFORCE with advantage:
        θ = θ + α ∇_θ log π_θ(a_t|s_t) A_t
        """
        self.loss =  -tf.reduce_mean(self.logprob * self.advantage_placeholder)

    def add_optimizer_op(self):
        """
        Set 'self.train_op' using AdamOptimizer
        """
        self.train_op = tf.train.AdamOptimizer(learning_rate = self.lr).minimize(self.loss)

    def add_baseline_op(self, scope = "baseline"):
        """
        Build the baseline network within the scope.
        Use build_mlp with the same parameters as the policy network to
        get the baseline estimate, and setup a target placeholder and
        an update operation so the baseline can be trained.

        Args:
          scope: the scope of the baseline network
        """
        self.baseline = tf.squeeze(build_mlp(self.observation_placeholder, 1, scope, self.pa.n_lay
ers, self.pa.layer_size))
        self.baseline_target_placeholder = tf.placeholder(tf.float32, shape = [None, ])
        loss = tf.losses.mean_squared_error(labels = self.baseline_target_placeholder, predictions
= self.baseline)
        self.update_baseline_op = tf.train.AdamOptimizer(learning_rate = self.lr).minimize(loss)

    def build(self):
        """
        Build the model by adding all necessary variables.
        """
        # add placeholders
        self.add_placeholders_op()
        # create policy net
        self.build_policy_network_op()
        # add square loss
        self.add_loss_op()
        # add optmizer for the main networks
        self.add_optimizer_op()

        # add baseline
        if self.pa.use_baseline:
```

```python
            self.add_baseline_op()

    def initialize(self):
        """
        Assumes the graph has been constructed (have called self.build())
        Creates a tf Session and run initializer of variables
        """
        # create tf session
        self.sess = tf.Session()
        # initiliaze all variables
        init = tf.global_variables_initializer()
        self.sess.run(init)


    def sample_path(self, env, num_episodes = None):
        """
        Sample trajectories from the environment.

        Args:
            env: a built simulator environment
            num_episodes: the number of episodes to be sampled
              if none, sample one batch (size indicated by config file)
        Returns:
          paths: a list of paths. Each path in paths is a dictionary with
              path["observation"] a numpy array of ordered observations in the path
              path["actions"] a numpy array of the corresponding actions in the path
              path["reward"] a numpy array of the corresponding rewards in the path
          total_rewards: the sum of all rewards encountered during this "path"
        """
        episode = 0
        episode_rewards = []
        paths = []
        t = 0

        while (num_episodes or t < self.batch_size):
            env.reset()
            states, actions, rewards= [], [], []
            state = env.observe()
            episode_reward = 0

            for step in range(self.episode_max_length):
                states.append(state)
                action = self.sess.run(self.sampled_action, feed_dict={self.observation_placeholde
r : states[-1][None]})[0]
                actions.append(action)
                state, reward, done = env.step(action)
                rewards.append(reward)
                episode_reward += reward
                t += 1
                if (done or step == self.episode_max_length-1):
                    episode_rewards.append(episode_reward)
                    break
                if (not num_episodes) and t == self.batch_size:
                    break

            path = {"observation" : np.array(states),
                            "reward" : np.array(rewards),
                            "action" : np.array(actions),
                            "info"   : info}
            paths.append(path)
            episode += 1
            if num_episodes and episode >= self.num_seq_per_batch:
                break

        return paths, episode_rewards
    def get_returns(self, paths):
        """
        Calculate the returns G_t for each timestep
        After acting in the environment, we record the observations, actions, and
        rewards. To get the advantages that we need for the policy update, we have
        to convert the rewards into returns, G_t, which are themselves an estimate
        of Q^π (s_t, a_t):
                    G_t = r_t + γ r_{t+1} + γ^2 r_{t+2} + ... + γ^{T-t} r_T
            where T is the last timestep of the episode.
        Args:
              paths: recorded sample paths.  See sample_path() for details.
        Return:
              returns: return G_t for each timestep
        """
        all_returns = []
        for path in paths:
            rewards = path["reward"]
```

```python
            returns = []
            T = len(rewards)
            for i in range(T):
                gammas = np.logspace(0, T - i, num = T - i, base = self.gamma, endpoint = False)
                r_t = np.dot(rewards[i:], gammas)
                returns.append(r_t)
            all_returns.append(returns)
        returns = np.concatenate(all_returns)
        return returns

    def train(self):
        """
        Performs training
        """
        last_eval = 0
        last_record = 0
        scores_eval = []

        self.init_averages()
        scores_eval = [] # list of scores computed at iteration time

        for t in range(self.pa.num_batches):

            # collect a minibatch of samples
            paths, total_rewards = self.sample_path(self.env)
            scores_eval = scores_eval + total_rewards
            observations = np.concatenate([path["observation"] for path in paths])
            actions = np.concatenate([path["action"] for path in paths])
            rewards = np.concatenate([path["reward"] for path in paths])
            # compute Q-val estimates (discounted future returns) for each time step
            returns = self.get_returns(paths)
            advantages = self.calculate_advantage(returns, observations)

            # run training operations
            if self.pa.use_baseline:
                self.update_baseline(returns, observations)
            self.sess.run(self.train_op, feed_dict={
                          self.observation_placeholder : observations,
                          self.action_placeholder : actions,
                          self.advantage_placeholder : advantages})

            # tf stuff
            if (t % self.pa.summary_freq == 0):
                self.update_averages(total_rewards, scores_eval)
                self.record_summary(t)

            # compute reward statistics for this batch and log
            avg_reward = np.mean(total_rewards)
            sigma_reward = np.sqrt(np.var(total_rewards) / len(total_rewards))
            msg = "Average reward: {:04.2f} +/- {:04.2f}".format(avg_reward, sigma_reward)
            self.logger.info(msg)

        self.logger.info("- Training done.")
        export_plot(scores_eval, "Score", self.pa.env_name, self.pa.plot_output)

    def run(self):
        """
        Apply procedures of training for a PG.
        """
        # initialize
        self.initialize()
        # model
        self.train()
```

## 4.Write Proof (with proper notation) of the Compatible Function Approximation Theorem

If the following two conditions are satisfied:

Critic gradient is compatible with the Actor score function

$$\nabla_w Q(s, a; w) = \nabla_\theta log\pi_(a, s; \theta)$$

Critic parameters w minimize the following mean-squared error:

$$\epsilon = \int_S \rho^\pi(s) \int_A \nabla_\theta \pi_(a, s; \theta)(Q^\pi(s, a) - Q(s, a; w)^2)da \cdot ds$$

Then the Policy Gradient using critic $Q(s, a; w)$ is exact:

$$\nabla_\theta J(\theta) = \int_S \rho^\pi(s) \int_A \nabla_\theta \pi_(a, s; \theta) Q(s, a; w) da \cdot ds$$

To minimize $\epsilon$

$$\epsilon = \int_S \rho^\pi(s) \int_A \pi_(a, s; \theta)(Q^\pi(s, a) - Q(s, a; w)^2)da \cdot ds$$

$$\int_S \rho^\pi(s) \int_A \pi_(a, s; \theta)(Q^\pi(s, a) - Q(s, a; w) \nabla_w Q(s, a; w))da \cdot ds = 0$$

Since in condition 1 :

$$\nabla_w Q(s, a; w) = \nabla_\theta log\pi_(a, s; \theta)$$

$$\int_S \rho^\pi(s) \int_A \pi_(a, s; \theta)(Q^\pi(s, a) - Q(s, a; w) \nabla_\theta log\pi_(a, s; \theta))da \cdot ds = 0$$

Therefore,

$$\begin{aligned} &\int_S \rho^\pi(s) \int_A \pi_(a, s; \theta) Q^\pi(s, a) \nabla_\theta log\pi_(a, s; \theta))da \cdot ds \\ &= \int_S \rho^\pi(s) \int_A \pi_(a, s; \theta) Q(s, a; w) \nabla_\theta log\pi_(a, s; \theta))da \cdot ds \\ &= \int_S \rho^\pi(s) \int_A Q(s, a; w) \nabla_\theta \pi_(a, s; \theta))da \cdot ds \end{aligned}$$

Proved

This means with conditions (1) and (2) of Compatible Function Approximation Theorem, we can use the critic func approx $Q(s, a; w)$ and still have the exact Policy Gradient.