Policy Gradient (Part 1)

Hong Xingxing

October 18, 2018

Outline

REINFORCE

Algorithm 1 REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π^*

Input:

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

while TRUE **do**
    Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(.|.,\theta)$
    for all steps in episode $t = 0, 1, ..., T-1$ **do**
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\theta \leftarrow \theta + \alpha\gamma^t G \nabla \ln \pi(A_t|S_t,\theta)$
    end for
end while

Policy Gradient

$$J(\theta) = \sum_{s \in \mathcal{S}} d^\pi(s) V^\pi(s) = \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q^\pi(s, a) \tag{1}$$

- where $d^\pi(s)$ is the **stationary distribution** of Markov chain for π_θ (on-policy state distribution under π)
- the θ parameter would be omitted for the policy π_θ when the policy is present in the subscript of other functions; for example, d^π and Q^π should be d^{π_θ} and Q^{π_θ} if written in full
- the probability of you ending up with one state **becomes unchanged** this is the **stationary probability** for π_θ.
  $d^\pi(s) = \lim_{t \to \infty} P(s_t = s | s_0, \pi_\theta)$

Policy Gradient Theorem

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s) \\ &\propto \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta\pi_\theta(a|s)\end{aligned}$$

policy gradient theorem provides a nice reformation of the derivative of the objective function to not involve the derivative of the state distribution $d^\pi(.)$

Proof of Policy Gradient Theorem

Start with the derivative of the state value function

$$
\begin{aligned}
&\nabla_\theta V^\pi(s) \\
=&\nabla_\theta\Big(\sum_{a\in\mathcal{A}}\pi_\theta(a|s)Q^\pi(s,a)\Big) \\
=&\sum_{a\in\mathcal{A}}\Big(\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a)+\pi_\theta(a|s)\nabla_\theta Q^\pi(s,a)\Big) && \text{; Derivative product rule.} \\
=&\sum_{a\in\mathcal{A}}\Big(\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a)+\pi_\theta(a|s)\nabla_\theta\sum_{s',r}P(s',r|s,a)(r+V^\pi(s'))\Big) && \text{; Extend } Q^\pi \text{ with future state value.} \\
=&\sum_{a\in\mathcal{A}}\Big(\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a)+\pi_\theta(a|s)\sum_{s',r}P(s',r|s,a)\nabla_\theta V^\pi(s')\Big) && P(s',r|s,a) \text{ or } r \text{ is not a func of } \theta \\
=&\sum_{a\in\mathcal{A}}\Big(\nabla_\theta\pi_\theta(a|s)Q^\pi(s,a)+\pi_\theta(a|s)\sum_{s'}P(s'|s,a)\nabla_\theta V^\pi(s')\Big) && \text{; Because } P(s'|s,a)=\textstyle\sum_r P(s',r|s,a)
\end{aligned}
$$

Proof of Policy Gradient Theorem

Now we have:

$$\nabla_\theta V^\pi(s) = \sum_{a \in \mathcal{A}} \Big(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^\pi(s') \Big) \tag{2}$$

- future state value function $V^\pi(s')$ can be repeated unrolled by following the same equation.
- Lets consider the following visitation sequence and label the probability of transitioning from state s to state x with policy π_θ after k step as $\rho^\pi(s \to x, k)$

$$s \xrightarrow{a \sim \pi_\theta(.|s)} s' \xrightarrow{a \sim \pi_\theta(.|s')} s'' \xrightarrow{a \sim \pi_\theta(.|s'')} \dots$$

Proof of Policy Gradient Theorem

$\rho^{\pi}(s \to x, k)$

- when $k = 0$: $\rho^{\pi}(s \to s, k = 0) = 1$
- When $k = 1$, we scan through all possible actions and sum up the transition probabilities to the target state: $\rho^{\pi}(s \to s', k = 1) = \sum_a \pi_\theta(a|s) P(s'|s, a)$
- update the visitation probability recursively: $\rho^{\pi}(s \to x, k + 1) = \sum_{s'} \rho^{\pi}(s \to s', k) \rho^{\pi}(s' \to x, 1)$

Proof of Policy Gradient Theorem

Unroll the recursive representation of $\nabla_\theta V^\pi(s)$. Let $\phi(s) = \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a)$ to simplify the maths.

$$
\begin{aligned}
&\nabla_\theta V^\pi(s) \\
=&\phi(s) + \sum_a \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^\pi(s') \\
=&\phi(s) + \sum_{s'} \sum_a \pi_\theta(a|s) P(s'|s,a) \nabla_\theta V^\pi(s') \\
=&\phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \nabla_\theta V^\pi(s') \\
=&\phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) [\phi(s') + \sum_{s''} \rho^\pi(s' \to s'', 1) \nabla_\theta V^\pi(s'')] \\
=&\phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2) \nabla_\theta V^\pi(s'') \text{ ; Consider } s' \text{ as the middle point for } s \to s'' \\
=&\phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2) \phi(s'') + \sum_{s'''} \rho^\pi(s \to s''', 3) \nabla_\theta V^\pi(s''') \\
=& \ldots \text{; Repeatedly unrolling the part of } \nabla_\theta V^\pi(.) \\
=& \sum_{x\in\mathcal{S}} \sum_{k=0}^{\infty} \rho^\pi(s \to x, k) \phi(x)
\end{aligned}
$$

Proof of Policy Gradient Theorem

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta V^\pi(s_0) && \text{; Starting from a random state } s_0 \\
&= \sum_s \sum_{k=0}^{\infty} \rho^\pi(s_0 \to s, k)\phi(s) && \text{; Let } \eta(s) = \textstyle\sum_{k=0}^{\infty} \rho^\pi(s_0 \to s, k) \\
&= \sum_s \eta(s)\phi(s) && \\
&= \left(\sum_s \eta(s)\right) \sum_s \frac{\eta(s)}{\sum_s \eta(s)}\phi(s) && \text{; Normalize } \eta(s), s \in \mathcal{S} \text{ to be a probability distribution.} \\
&\propto \sum_s \frac{\eta(s)}{\sum_s \eta(s)}\phi(s) && \textstyle\sum_s \eta(s) \text{ is a constant} \\
&= \sum_s d^\pi(s) \sum_a \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) && d^\pi(s) = \tfrac{\eta(s)}{\sum_s \eta(s)} \text{ is stationary distribution.}
\end{aligned}
$$

Proof of Policy Gradient Theorem

$$
\begin{aligned}
\nabla_\theta J(\theta) &\propto \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta \pi_\theta(a|s) \\
&= \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\
&= \mathbb{E}_{s\sim d^\pi, a\sim\pi_\theta}[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)] \qquad ; \text{Because } (\ln x)' = 1/x
\end{aligned}
$$

Algorithm 1: REINFORCE

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)] \\
&= \mathbb{E}_\pi[G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)] \qquad \text{; Because } Q^\pi(S_t,A_t)=\mathbb{E}_\pi[G_t|S_t,A_t]
\end{aligned}
$$

Algorithm 2: Actor-Critic

Two components:

- Actor: policy model, updates the policy parameters θ for $\pi_\theta(a|s)$. It select actions given the current environment state
- Critic: updates the value function parameters w and depending on the algorithm it could be action-value $Q_w(a|s)$ or state-value $V_w(s)$. It criticizes the actions made by the actor
- Actor-critic methods are TD methods

Algorithm 2: Actor-Critic

Algorithm 2 Action-value Actor-Critic Algorithm

Input:

Initialize s, θ at random; sample $a \sim \pi_\theta(a|s)$;

for $t = 1 \dots T$ **do**

  Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$;

  Then sample the next action $a' \sim \pi_\theta(a'|s')$;

  Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \ln \pi_\theta(a|s)$;

  Compute the correction (TD error) for action-value at time t:

  $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$

  and use it to update the parameters of action-value function:

  $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$;

  Update $a \leftarrow a'$ and $s \leftarrow s'$

end for

Off-Policy Policy Gradient

- REINFORCE and Actor-Critic are **on-policy**: training samples are collected according to the target policy the very same policy that we try to optimize for
- The behavior policy for collecting samples is a known policy (predefined just like a hyperparameter), labelled as $\beta(a|s)$

$$J(\theta) = \sum_{s \in \mathcal{S}} d^\beta(s) \sum_{a \in \mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s) = \mathbb{E}_{s \sim d^\beta}\Big[\sum_{a \in \mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s)\Big]$$

- $d^\beta(s)$ is the stationary distribution of the behavior policy β and $d^\beta(s) = \lim_{t\to\infty} P(S_t = s|S_0, \beta)$
- Q^π is the action-value function estimated with regard to the **target policy** π

Off-Policy Policy Gradient

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{s\sim d^\beta}\Big[\sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s)\Big] \\
&= \mathbb{E}_{s\sim d^\beta}\Big[\sum_{a\in\mathcal{A}} \big(Q^\pi(s,a)\nabla_\theta\pi_\theta(a|s) + \pi_\theta(a|s)\nabla_\theta Q^\pi(s,a)\big)\Big] && \text{; Derivative product rule.} \\
&\overset{(i)}{\approx} \mathbb{E}_{s\sim d^\beta}\Big[\sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta\pi_\theta(a|s)\Big] && \text{; Ignore the red part: } \pi_\theta(a|s)\nabla_\theta Q^\pi(s,a). \\
&= \mathbb{E}_{s\sim d^\beta}\Big[\sum_{a\in\mathcal{A}} \beta(a|s)\frac{\pi_\theta(a|s)}{\beta(a|s)} Q^\pi(s,a)\frac{\nabla_\theta\pi_\theta(a|s)}{\pi_\theta(a|s)}\Big] \\
&= \mathbb{E}_\beta\Big[\frac{\pi_\theta(a|s)}{\beta(a|s)} Q^\pi(s,a)\nabla_\theta \ln\pi_\theta(a|s)\Big] && \text{; The blue part is the importance weight.}
\end{aligned}
$$

$\frac{\pi_\theta(a|s)}{\beta(a|s)}$ is the importance weight. When applying policy gradient in the off-policy setting, we can simple adjust it with a weighted sum and the weight is the ratio of the target policy to the behavior policy

CartPole-v0 REINFORCE (Tensorflow)[1]

```
import numpy as np
import tensorflow as tf

np.random.seed(1)
tf.set_random_seed(1)

class PolicyGradient:
    def __init__(
            self,
            n_actions,
            n_features,
            learning_rate=0.01,
            reward_decay=0.95,
            output_graph=False,
    ):
        self.n_actions = n_actions
        self.n_features = n_features
        self.lr = learning_rate
        self.gamma = reward_decay

        self.ep_obs, self.ep_as, self.ep_rs = [], [], []

        self._build_net()
        self.sess = tf.Session()
        if output_graph:
            tf.summary.FileWriter("logs/", self.sess.graph)
        self.sess.run(tf.global_variables_initializer())
```

[1] https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/tree/master/contents/7_Policy_gradient_softmax

CartPole-v0 REINFORCE (Tensorflow)

```
    def _build_net(self):
        with tf.name_scope('inputs'):
            self.tf_obs = tf.placeholder(tf.float32, [None, self.n_features], name="observations")
            self.tf_acts = tf.placeholder(tf.int32, [None, ], name="actions_num")
            self.tf_vt = tf.placeholder(tf.float32, [None, ], name="actions_value")
        # fc1
        layer = tf.layers.dense(
            inputs=self.tf_obs,
            units=10,
            activation=tf.nn.tanh,  # tanh activation
            kernel_initializer=tf.random_normal_initializer(mean=0, stddev=0.3),
            bias_initializer=tf.constant_initializer(0.1),
            name='fc1'
        )
        # fc2
        all_act = tf.layers.dense(
            inputs=layer,
            units=self.n_actions,
            activation=None,
            kernel_initializer=tf.random_normal_initializer(mean=0, stddev=0.3),
            bias_initializer=tf.constant_initializer(0.1),
            name='fc2'
        )
        self.all_act_prob = tf.nn.softmax(all_act, name='act_prob')
        with tf.name_scope('loss'):
            neg_log_prob = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=all_act,
    ↪   labels=self.tf_acts)
            loss = tf.reduce_mean(neg_log_prob * self.tf_vt)
        with tf.name_scope('train'):
            self.train_op = tf.train.AdamOptimizer(self.lr).minimize(loss)
```

CartPole-v0 REINFORCE (Tensorflow)

```
    def choose_action(self, observation):
        prob_weights = self.sess.run(self.all_act_prob, feed_dict={self.tf_obs: observation[np.newaxis,
↪       :]})
        action = np.random.choice(range(prob_weights.shape[1]), p=prob_weights.ravel())
        # select action w.r.t the actions prob
        return action

    def store_transition(self, s, a, r):
        self.ep_obs.append(s)
        self.ep_as.append(a)
        self.ep_rs.append(r)

    def learn(self):
        # discount and normalize episode reward
        discounted_ep_rs_norm = self._discount_and_norm_rewards()

        # train on episode
        self.sess.run(self.train_op, feed_dict={
             self.tf_obs: np.vstack(self.ep_obs),  # shape=[None, n_obs]
             self.tf_acts: np.array(self.ep_as),  # shape=[None, ]
             self.tf_vt: discounted_ep_rs_norm,  # shape=[None, ]
        })

        self.ep_obs, self.ep_as, self.ep_rs = [], [], []    # empty episode data
        return discounted_ep_rs_norm
```

CartPole-v0 REINFORCE (Tensorflow)

```
def _discount_and_norm_rewards(self):
    # discount episode rewards
    discounted_ep_rs = np.zeros_like(self.ep_rs)
    running_add = 0
    for t in reversed(range(0, len(self.ep_rs))):
        running_add = running_add * self.gamma + self.ep_rs[t]
        discounted_ep_rs[t] = running_add

    # normalize episode rewards
    discounted_ep_rs -= np.mean(discounted_ep_rs)
    discounted_ep_rs /= np.std(discounted_ep_rs)
    return discounted_ep_rs
```

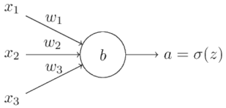
CartPole-v0 REINFORCE (Tensorflow)

```
import gym
from RL_brain import PolicyGradient
import matplotlib.pyplot as plt

DISPLAY_REWARD_THRESHOLD = 400  # renders environment if total episode reward is greater then this
 ↪    threshold
RENDER = False  # rendering wastes time
env = gym.make('CartPole-v0')
env.seed(1)     # reproducible, general Policy gradient has high variance
env = env.unwrapped

RL = PolicyGradient(
    n_actions=env.action_space.n,
    n_features=env.observation_space.shape[0],
    learning_rate=0.02,
    reward_decay=0.99,
    # output_graph=True,
)
```

CartPole-v0 REINFORCE (Tensorflow)

```
for i_episode in range(3000):
    observation = env.reset()

    while True:
        if RENDER: env.render()

        action = RL.choose_action(observation)
        observation_, reward, done, info = env.step(action)
        RL.store_transition(observation, action, reward)

        if done:
            ep_rs_sum = sum(RL.ep_rs)

            if 'running_reward' not in globals():
                running_reward = ep_rs_sum
            else:
                running_reward = running_reward * 0.99 + ep_rs_sum * 0.01
            if running_reward > DISPLAY_REWARD_THRESHOLD: RENDER = True     # rendering
            print("episode:", i_episode, "  reward:", int(running_reward))

            vt = RL.learn()

            if i_episode == 0:
                plt.plot(vt)    # plot the episode vt
                plt.xlabel('episode steps')
                plt.ylabel('normalized state-action value')
                plt.show()
            break

        observation = observation_
```

Cross Entropy



here $a = \sigma(z)$ and $z = \sum_j w_j x_j + b$. Then we define the cross entropy cost function as

$$C = -\frac{1}{n} \sum x \left[y \ln a + (1 - y) \ln(1 - a) \right] \tag{3}$$

We have $C >= 0$ and for any x, if the prediction a is equal to the real y, then C is equal 0. Thus the cross entropy is an appropriate cost function definition

Cross Entropy with Sigmoid

$$
\begin{aligned}
\frac{\partial C}{\partial w_j} &= -\frac{1}{n}\sum_x \left[\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right]\frac{\partial \sigma}{\partial w_j} \\
&= -\frac{1}{n}\sum_x \left[\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right]\sigma^{'}(z)x_j \\
&= \frac{1}{n}\sum_x \frac{\sigma^{'}(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z)-y)
\end{aligned}
$$

$\sigma(z) = 1/(1+e^{-z})$ and $\sigma^{'}(z) = \sigma(z)(1-\sigma(z))$ Thus the derivative of Cross Entropy Loss with sigmoid is

$$
\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z)-y) \tag{4}
$$

similarly we get $\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x(\sigma(z)-y)$

Cross Entropy with Softmax

$$L = -\sum_i y_i log(p_i)$$

$$\frac{\partial L}{\partial o_i} = -\sum_k y_k \frac{\partial log(p_k)}{\partial o_i}$$

$$= -\sum_k y_k \frac{\partial log(p_k)}{\partial p_k} \times \frac{\partial p_k}{\partial o_i}$$

$$= -\sum y_k \frac{1}{p_k} \times \frac{\partial p_k}{\partial o_i}$$

$$\frac{\partial L}{\partial o_i} = -y_i(1-p_i) - \sum_{k\neq i} y_k \frac{1}{p_k}(-p_k.p_i)$$

$$= -y_i(1-p_i) + \sum_{k\neq i} y_k.p_i$$

$$= -y_i + y_i p_i + \sum_{k\neq i} y_k.p_i$$

$$= p_i \left(y_i + \sum_{k\neq i} y_k \right) - y_i$$

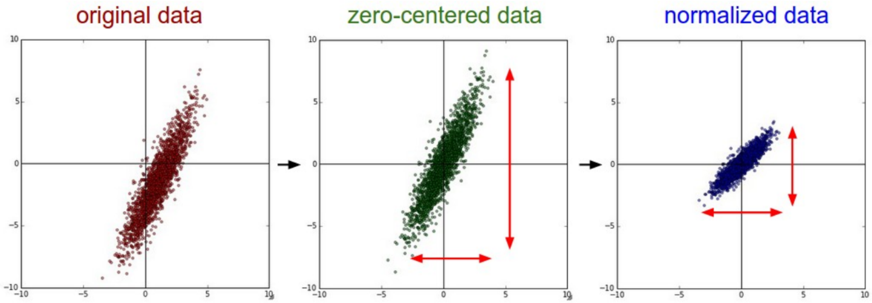
$$= p_i \left(y_i + \sum_{k\neq i} y_k \right) - y_i$$

$$= p_i - y_i$$

Data Preprocessing

- There are three common forms of data preprocessing a data matrix X, where we will assume that X is of size [N x D] (N is the number of data, D is their dimensionality).
- **Mean subtraction**: subtracting the mean across every individual feature in the data, and has the geometric interpretation of centering the cloud of data around the origin along every dimension. $(X- = np.mean(X))$
- **Normalization**: normalizing the data dimensions so that they are of approximately the same scale.
  - One is to divide each dimension by its standard deviation, once it has been zero-centered: $(X/ = np.std(X, axis = 0))$
  - Another form of this preprocessing normalizes each dimension so that the min and max along the dimension is -1 and 1 respectively.

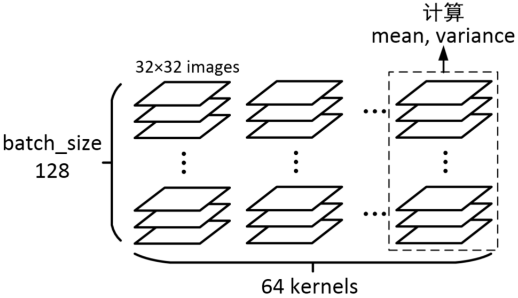
Data Preprocessing



Common data preprocessing pipeline. **Left**: Original toy, 2-dimensional input data. **Middle**: The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin. **Right**: Each dimension is additionally scaled by its standard deviation. The red lines indicate the extent of the data - they are of unequal length in the middle, but of equal length on the right.

Batch Normalization

tf.nn.batch_normalization(x, mean, variance, offset, scale, variance_epsilon, name=None)



$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}(x_i - \mu_{\mathcal{B}})^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathcal{BN}_{\gamma,\beta}(x_i)$$

Cross Entropy Functions

- tf.nn.sigmoid_cross_entropy_with_logits
- tf.nn.softmax_cross_entropy_with_logits
- tf.nn.sparse_softmax_cross_entropy_with_logits
- tf.nn.weighted_cross_entropy_with_logits

Cross Entropy Functions

tf.nn.sigmoid_cross_entropy_with_logits(_sentinel=None, labels=None, logits=None, name=None)

For brevity, let $x =$ logits, $z =$ labels. The logistic loss is

$$z * -log(sigmoid(x)) + (1 - z) * -log(1 - sigmoid(x))$$
$$= z * -log(1/(1 + exp(-x))) + (1 - z) * -log(exp(-x)/(1 + exp(-x)))$$
$$= z * log(1 + exp(-x)) + (1 - z) * (-log(exp(-x)) + log(1 + exp(-x)))$$
$$= z * log(1 + exp(-x)) + (1 - z) * (x + log(1 + exp(-x))$$
$$= (1 - z) * x + log(1 + exp(-x))$$
$$= x - x * z + log(1 + exp(-x))$$

Cross Entropy Functions

tf.nn.softmax_cross_entropy_with_logits(_sentinel=None, labels=None, logits=None, dim=-1, name=None)

- While the classes are mutually exclusive, their probabilities need not be. All that is required is that each row of labels is a **valid probability distribution**
- If using exclusive labels (wherein one and only one class is true at a time), see sparse_softmax_cross_entropy_with_logits
- This op expects **unscaled logits, since it performs a softmax on logits internally** for efficiency
- A common use case is to have logits and labels of shape $[batch\_size, num\_classes]$

Cross Entropy Functions

tf.nn.sparse_softmax_cross_entropy_with_logits(_sentinel=None, labels=None, logits=None, name=None)

- For this operation, the probability of a given label **is considered exclusive**. That is, **soft classes are not allowed**, and the labels vector must provide a **single specific index** for the true class for each row of logits (each minibatch entry)
- Each entry in labels must be an index in $[0, num\_classes)$

Cross Entropy Functions

tf.nn.weighted_cross_entropy_with_logits(targets, logits, pos_weight,name=None)

$targets * -log(sigmoid(logits)) * pos\_weight + (1 - targets) * -log(1 - sigmoid(logits))$

For brevity, let $x =$ logits, $z =$ targets, $q =$ pos_weight. The loss is:

$$
\begin{aligned}
& qz * -log(sigmoid(x)) + (1 - z) * -log(1 - sigmoid(x)) \\
&= qz * -log(1/(1 + exp(-x))) + (1 - z) * -log(exp(-x)/(1 + exp(-x))) \\
&= qz * log(1 + exp(-x)) + (1 - z) * (-log(exp(-x)) + log(1 + exp(-x))) \\
&= qz * log(1 + exp(-x)) + (1 - z) * (x + log(1 + exp(-x)) \\
&= (1 - z) * x + (qz + 1 - z) * log(1 + exp(-x)) \\
&= (1 - z) * x + (1 + (q - 1) * z) * log(1 + exp(-x))
\end{aligned}
$$

CartPole-v0 REINFORCE (PyTorch)[2]

```
import argparse
import gym
import numpy as np
from itertools import count
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical

parser = argparse.ArgumentParser(description='PyTorch REINFORCE example')
parser.add_argument('--gamma', type=float, default=0.99, metavar='G',
                    help='discount factor (default: 0.99)')
parser.add_argument('--seed', type=int, default=543, metavar='N',
                    help='random seed (default: 543)')
parser.add_argument('--render', action='store_true',
                    help='render the environment')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='interval between training status logs (default: 10)')
args = parser.parse_args()


env = gym.make('CartPole-v0')
env.seed(args.seed)
torch.manual_seed(args.seed)
```

[2] https://github.com/pytorch/examples/blob/master/reinforcement_learning/reinforce.py

CartPole-v0 REINFORCE (PyTorch)

```
class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.affine1 = nn.Linear(4, 128)
        self.affine2 = nn.Linear(128, 2)

        self.saved_log_probs = []
        self.rewards = []

    def forward(self, x):
        x = F.relu(self.affine1(x))
        action_scores = self.affine2(x)
        return F.softmax(action_scores, dim=1)

policy = Policy()
optimizer = optim.Adam(policy.parameters(), lr=1e-2)
eps = np.finfo(np.float32).eps.item()
```

CartPole-v0 REINFORCE (PyTorch)

```
def select_action(state):
    state = torch.from_numpy(state).float().unsqueeze(0)
    probs = policy(state)
    m = Categorical(probs)
    action = m.sample()
    policy.saved_log_probs.append(m.log_prob(action))
    return action.item()

def finish_episode():
    R = 0
    policy_loss = []
    rewards = []
    for r in policy.rewards[::-1]:
        R = r + args.gamma * R
        rewards.insert(0, R)
    rewards = torch.tensor(rewards)
    rewards = (rewards - rewards.mean()) / (rewards.std() + eps)
    for log_prob, reward in zip(policy.saved_log_probs, rewards):
        policy_loss.append(-log_prob * reward)
    optimizer.zero_grad()
    policy_loss = torch.cat(policy_loss).sum()
    policy_loss.backward()
    optimizer.step()
    del policy.rewards[:]
    del policy.saved_log_probs[:]
```

CartPole-v0 REINFORCE (PyTorch)

```
def main():
    running_reward = 10
    for i_episode in count(1):
        state = env.reset()
        for t in range(10000):  # Don't infinite loop while learning
            action = select_action(state)
            state, reward, done, _ = env.step(action)
            if args.render:
                env.render()
            policy.rewards.append(reward)
            if done:
                break

        running_reward = running_reward * 0.99 + t * 0.01
        finish_episode()
        if i_episode % args.log_interval == 0:
            print('Episode {}\tLast length: {:5d}\tAverage length: {:.2f}'.format(
                i_episode, t, running_reward))
        if running_reward > env.spec.reward_threshold:
            print("Solved! Running reward is now {} and "
                  "the last episode runs to {} time steps!".format(running_reward, t))
            break

if __name__ == '__main__':
    main()
```

CartPole-v0 Actor-Critic (Tensorflow)[3]

```
import numpy as np
import tensorflow as tf
import gym

np.random.seed(2)
tf.set_random_seed(2)  # reproducible

# Superparameters
OUTPUT_GRAPH = False
MAX_EPISODE = 3000
DISPLAY_REWARD_THRESHOLD = 200  # renders environment if total episode reward is greater then this
    threshold
MAX_EP_STEPS = 1000   # maximum time step in one episode
RENDER = False  # rendering wastes time
GAMMA = 0.9     # reward discount in TD error
LR_A = 0.001    # learning rate for actor
LR_C = 0.01     # learning rate for critic

env = gym.make('CartPole-v0')
env.seed(1)  # reproducible
env = env.unwrapped

N_F = env.observation_space.shape[0]
N_A = env.action_space.n
```

[3] http://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/blob/master/contents/8_Actor_Critic_Advantage/AC_CartPole.py

CartPole-v0 Actor (Tensorflow)

```
class Actor(object):
    def __init__(self, sess, n_features, n_actions, lr=0.001):
        self.sess = sess

        self.s = tf.placeholder(tf.float32, [1, n_features], "state")
        self.a = tf.placeholder(tf.int32, None, "act")
        self.td_error = tf.placeholder(tf.float32, None, "td_error")  # TD_error

        with tf.variable_scope('Actor'):
            l1 = tf.layers.dense(
                inputs=self.s,
                units=20,    # number of hidden units
                activation=tf.nn.relu,
                kernel_initializer=tf.random_normal_initializer(0., .1),    # weights
                bias_initializer=tf.constant_initializer(0.1),  # biases
                name='l1'
            )
```

CartPole-v0 Actor (Tensorflow)

```
        self.acts_prob = tf.layers.dense(
            inputs=l1,
            units=n_actions,    # output units
            activation=tf.nn.softmax,   # get action probabilities
            kernel_initializer=tf.random_normal_initializer(0., .1),  # weights
            bias_initializer=tf.constant_initializer(0.1),  # biases
            name='acts_prob'
        )

    with tf.variable_scope('exp_v'):
        log_prob = tf.log(self.acts_prob[0, self.a])
        self.exp_v = tf.reduce_mean(log_prob * self.td_error)  # advantage (TD_error) guided loss

    with tf.variable_scope('train'):
        self.train_op = tf.train.AdamOptimizer(lr).minimize(-self.exp_v)  # minimize(-exp_v) =
↪   maximize(exp_v)
```

CartPole-v0 Actor (Tensorflow)

```
def learn(self, s, a, td):
    s = s[np.newaxis, :]
    feed_dict = {self.s: s, self.a: a, self.td_error: td}
    _, exp_v = self.sess.run([self.train_op, self.exp_v], feed_dict)
    return exp_v

def choose_action(self, s):
    s = s[np.newaxis, :]
    probs = self.sess.run(self.acts_prob, {self.s: s})   # get probabilities for all actions
    return np.random.choice(np.arange(probs.shape[1]), p=probs.ravel())   # return a int
```

CartPole-v0 Critic (Tensorflow)

```
class Critic(object):
    def __init__(self, sess, n_features, lr=0.01):
        self.sess = sess

        self.s = tf.placeholder(tf.float32, [1, n_features], "state")
        self.v_ = tf.placeholder(tf.float32, [1, 1], "v_next")
        self.r = tf.placeholder(tf.float32, None, 'r')

        with tf.variable_scope('Critic'):
            l1 = tf.layers.dense(
                inputs=self.s,
                units=20,  # number of hidden units
                activation=tf.nn.relu,  # None
                # have to be linear to make sure the convergence of actor.
                # But linear approximator seems hardly learns the correct Q.
                kernel_initializer=tf.random_normal_initializer(0., .1),  # weights
                bias_initializer=tf.constant_initializer(0.1),  # biases
                name='l1'
            )
```

CartPole-v0 Critic (Tensorflow)

```
        self.v = tf.layers.dense(
            inputs=l1,
            units=1,  # output units
            activation=None,
            kernel_initializer=tf.random_normal_initializer(0., .1),  # weights
            bias_initializer=tf.constant_initializer(0.1),  # biases
            name='V'
        )

    with tf.variable_scope('squared_TD_error'):
        self.td_error = self.r + GAMMA * self.v_ - self.v
        self.loss = tf.square(self.td_error)    # TD_error = (r+gamma*V_next) - V_eval
    with tf.variable_scope('train'):
        self.train_op = tf.train.AdamOptimizer(lr).minimize(self.loss)
```

CartPole-v0 Critic (Tensorflow)

```
def learn(self, s, r, s_):
    s, s_ = s[np.newaxis, :], s_[np.newaxis, :]

    v_ = self.sess.run(self.v, {self.s: s_})
    td_error, _ = self.sess.run([self.td_error, self.train_op],
                                {self.s: s, self.v_: v_, self.r: r})
    return td_error
```

CartPole-v0 Actor-Critic (Tensorflow)

```
sess = tf.Session()

actor = Actor(sess, n_features=N_F, n_actions=N_A, lr=LR_A)
critic = Critic(sess, n_features=N_F, lr=LR_C)     # we need a good teacher, so the teacher should learn
 ↪    faster than the actor

sess.run(tf.global_variables_initializer())

if OUTPUT_GRAPH:
    tf.summary.FileWriter("logs/", sess.graph)
```

CartPole-v0 Actor-Critic (Tensorflow)

```
for i_episode in range(MAX_EPISODE):
    s = env.reset()
    t = 0
    track_r = []
    while True:
        if RENDER: env.render()

        a = actor.choose_action(s)

        s_, r, done, info = env.step(a)

        if done: r = -20

        track_r.append(r)

        td_error = critic.learn(s, r, s_)  # gradient = grad[r + gamma * V(s_) - V(s)]
        actor.learn(s, a, td_error)     # true_gradient = grad[logPi(s,a) * td_error]

        s = s_
        t += 1

        if done or t >= MAX_EP_STEPS:
            ep_rs_sum = sum(track_r)

            if 'running_reward' not in globals():
                running_reward = ep_rs_sum
            else:
                running_reward = running_reward * 0.95 + ep_rs_sum * 0.05
            if running_reward > DISPLAY_REWARD_THRESHOLD: RENDER = True  # rendering
            print("episode:", i_episode, "  reward:", int(running_reward))
            break
```

CartPole-v0 Actor-Critic (PyTorch)[4]

```
import argparse
import gym
import numpy as np
from itertools import count
from collections import namedtuple
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical

parser = argparse.ArgumentParser(description='PyTorch actor-critic example')
parser.add_argument('--gamma', type=float, default=0.99, metavar='G',
                    help='discount factor (default: 0.99)')
parser.add_argument('--seed', type=int, default=543, metavar='N',
                    help='random seed (default: 1)')
parser.add_argument('--render', action='store_true',
                    help='render the environment')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='interval between training status logs (default: 10)')
args = parser.parse_args()

env = gym.make('CartPole-v0')
env.seed(args.seed)
torch.manual_seed(args.seed)

SavedAction = namedtuple('SavedAction', ['log_prob', 'value'])
```

[4] https://github.com/pytorch/examples/blob/master/reinforcement_learning/actor_critic.py

CartPole-v0 Actor-Critic (PyTorch)

```
class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.affine1 = nn.Linear(4, 128)
        self.action_head = nn.Linear(128, 2)
        self.value_head = nn.Linear(128, 1)

        self.saved_actions = []
        self.rewards = []

    def forward(self, x):
        x = F.relu(self.affine1(x))
        action_scores = self.action_head(x)
        state_values = self.value_head(x)
        return F.softmax(action_scores, dim=-1), state_values


model = Policy()
optimizer = optim.Adam(model.parameters(), lr=3e-2)
eps = np.finfo(np.float32).eps.item()
```

CartPole-v0 Actor-Critic (PyTorch)

```
def select_action(state):
    state = torch.from_numpy(state).float()
    probs, state_value = model(state)
    m = Categorical(probs)
    action = m.sample()
    model.saved_actions.append(SavedAction(m.log_prob(action), state_value))
    return action.item()


def finish_episode():
    R = 0
    saved_actions = model.saved_actions
    policy_losses = []
    value_losses = []
    rewards = []
    for r in model.rewards[::-1]:
        R = r + args.gamma * R
        rewards.insert(0, R)
    rewards = torch.tensor(rewards)
    rewards = (rewards - rewards.mean()) / (rewards.std() + eps)
    for (log_prob, value), r in zip(saved_actions, rewards):
        reward = r - value.item()
        policy_losses.append(-log_prob * reward)
        value_losses.append(F.smooth_l1_loss(value, torch.tensor([r])))
    optimizer.zero_grad()
    loss = torch.stack(policy_losses).sum() + torch.stack(value_losses).sum()
    loss.backward()
    optimizer.step()
    del model.rewards[:]
    del model.saved_actions[:]
```

CartPole-v0 Actor-Critic (PyTorch)

```
def main():
    running_reward = 10
    for i_episode in count(1):
        state = env.reset()
        for t in range(10000):  # Don't infinite loop while learning
            action = select_action(state)
            state, reward, done, _ = env.step(action)
            if args.render:
                env.render()
            model.rewards.append(reward)
            if done:
                break

        running_reward = running_reward * 0.99 + t * 0.01
        finish_episode()
        if i_episode % args.log_interval == 0:
            print('Episode {}\tLast length: {:5d}\tAverage length: {:.2f}'.format(
                i_episode, t, running_reward))
        if running_reward > env.spec.reward_threshold:
            print("Solved! Running reward is now {} and "
                  "the last episode runs to {} time steps!".format(running_reward, t))
            break


if __name__ == '__main__':
    main()
```

References

- https://deepnotes.io/softmax-crossentropy
- https://hit-scir.gitbooks.io/neural-networks-and-deep-learning-zh_cn/content/chap3/c3s1.html
- http://cs231n.github.io/neural-networks-2/
- https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html