```
import random
import numpy as np
import tensorflow as tf

class PolicyGradientREINFORCE(object):

  def __init__(self, session,
                     optimizer,
                     policy_network,
                     state_dim,
                     num_actions,
                     init_exp=0.5,         # initial exploration prob
                     final_exp=0.0,        # final exploration prob
                     anneal_steps=10000,   # N steps for annealing exploration
                     discount_factor=0.99, # discount future rewards
                     reg_param=0.001,      # regularization constants
                     max_gradient=5,       # max gradient norms
                     summary_writer=None,
                     summary_every=100):

    # tensorflow machinery
    self.session        = session
    self.optimizer      = optimizer
    self.summary_writer = summary_writer

    # model components
    self.policy_network = policy_network

    # training parameters
    self.state_dim       = state_dim
    self.num_actions     = num_actions
    self.discount_factor = discount_factor
    self.max_gradient    = max_gradient
    self.reg_param       = reg_param

    # exploration parameters
    self.exploration  = init_exp
    self.init_exp     = init_exp
    self.final_exp    = final_exp
    self.anneal_steps = anneal_steps

    # counters
    self.train_iteration = 0

    # rollout buffer
    self.state_buffer  = []
    self.reward_buffer = []
    self.action_buffer = []

    # record reward history for normalization
    self.all_rewards = []
    self.max_reward_length = 1000000

    # create and initialize variables
    self.create_variables()
    var_lists = tf.get_collection(tf.GraphKeys.VARIABLES)
    self.session.run(tf.initialize_variables(var_lists))

    # make sure all variables are initialized
    self.session.run(tf.assert_variables_initialized())

    if self.summary_writer is not None:
      # graph was not available when journalist was created
      self.summary_writer.add_graph(self.session.graph)
```

```
      self.summary_every = summary_every

  def resetModel(self):
    self.cleanUp()
    self.train_iteration = 0
    self.exploration     = self.init_exp
    var_lists = tf.get_collection(tf.GraphKeys.VARIABLES)
    self.session.run(tf.initialize_variables(var_lists))

  def create_variables(self):

    with tf.name_scope("model_inputs"):
      # raw state representation
      self.states = tf.placeholder(tf.float32, (None, self.state_dim), name="states")

    # rollout action based on current policy
    with tf.name_scope("predict_actions"):
      # initialize policy network
      with tf.variable_scope("policy_network"):
        self.policy_outputs = self.policy_network(self.states)

      # predict actions from policy network
      self.action_scores = tf.identity(self.policy_outputs, name="action_scores")
      # Note 1: tf.multinomial is not good enough to use yet
      # so we don't use self.predicted_actions for now
      self.predicted_actions = tf.multinomial(self.action_scores, 1)

    # regularization loss
    policy_network_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
scope="policy_network")

    # compute loss and gradients
    with tf.name_scope("compute_pg_gradients"):
      # gradients for selecting action from policy network
      self.taken_actions = tf.placeholder(tf.int32, (None,), name="taken_actions")
      self.discounted_rewards = tf.placeholder(tf.float32, (None,), name="discounted_rewards")

      with tf.variable_scope("policy_network", reuse=True):
        self.logprobs = self.policy_network(self.states)

      # compute policy loss and regularization loss
      self.cross_entropy_loss = tf.nn.sparse_softmax_cross_entropy_with_logits(self.logprobs,
self.taken_actions)
      self.pg_loss            = tf.reduce_mean(self.cross_entropy_loss)
      self.reg_loss           = tf.reduce_sum([tf.reduce_sum(tf.square(x)) for x in
policy_network_variables])
      self.loss               = self.pg_loss + self.reg_param * self.reg_loss

      # compute gradients
      self.gradients = self.optimizer.compute_gradients(self.loss)

      # compute policy gradients
      for i, (grad, var) in enumerate(self.gradients):
        if grad is not None:
          self.gradients[i] = (grad * self.discounted_rewards, var)

      for grad, var in self.gradients:
        tf.histogram_summary(var.name, var)
        if grad is not None:
          tf.histogram_summary(var.name + '/gradients', grad)

      # emit summaries
      tf.scalar_summary("policy_loss", self.pg_loss)
      tf.scalar_summary("reg_loss", self.reg_loss)
      tf.scalar_summary("total_loss", self.loss)
```

```
    # training update
    with tf.name_scope("train_policy_network"):
      # apply gradients to update policy network
      self.train_op = self.optimizer.apply_gradients(self.gradients)

    self.summarize = tf.merge_all_summaries()
    self.no_op = tf.no_op()

  def sampleAction(self, states):
    # TODO: use this code piece when tf.multinomial gets better
    # sample action from current policy
    # actions = self.session.run(self.predicted_actions, {self.states: states})[0]
    # return actions[0]

    # temporary workaround
    def softmax(y):
      """ simple helper function here that takes unnormalized logprobs """
      maxy = np.amax(y)
      e = np.exp(y - maxy)
      return e / np.sum(e)

    # epsilon-greedy exploration strategy
    if random.random() < self.exploration:
      return random.randint(0, self.num_actions-1)
    else:
      action_scores = self.session.run(self.action_scores, {self.states: states})[0]
      action_probs  = softmax(action_scores) - 1e-5
      action = np.argmax(np.random.multinomial(1, action_probs))
      return action

  def updateModel(self):

    N = len(self.reward_buffer)
    r = 0 # use discounted reward to approximate Q value

    # compute discounted future rewards
    discounted_rewards = np.zeros(N)
    for t in reversed(xrange(N)):
      # future discounted reward from now on
      r = self.reward_buffer[t] + self.discount_factor * r
      discounted_rewards[t] = r

    # reduce gradient variance by normalization
    self.all_rewards += discounted_rewards.tolist()
    self.all_rewards = self.all_rewards[:self.max_reward_length]
    discounted_rewards -= np.mean(self.all_rewards)
    discounted_rewards /= np.std(self.all_rewards)

    # whether to calculate summaries
    calculate_summaries = self.train_iteration % self.summary_every == 0 and self.summary_writer is
not None

    # update policy network with the rollout in batches
    for t in xrange(N-1):

      # prepare inputs
      states  = self.state_buffer[t][np.newaxis, :]
      actions = np.array([self.action_buffer[t]])
      rewards = np.array([discounted_rewards[t]])

      # evaluate gradients
      grad_evals = [grad for grad, var in self.gradients]

      # perform one update of training
```

```
      _, summary_str = self.session.run([
        self.train_op,
        self.summarize if calculate_summaries else self.no_op
      ], {
        self.states:             states,
        self.taken_actions:      actions,
        self.discounted_rewards: rewards
      })

      # emit summaries
      if calculate_summaries:
        self.summary_writer.add_summary(summary_str, self.train_iteration)

    self.annealExploration()
    self.train_iteration += 1

    # clean up
    self.cleanUp()

  def annealExploration(self, stategy='linear'):
    ratio = max((self.anneal_steps - self.train_iteration)/float(self.anneal_steps), 0)
    self.exploration = (self.init_exp - self.final_exp) * ratio + self.final_exp

  def storeRollout(self, state, action, reward):
    self.action_buffer.append(action)
    self.reward_buffer.append(reward)
    self.state_buffer.append(state)

  def cleanUp(self):
    self.state_buffer  = []
    self.reward_buffer = []
    self.action_buffer = []
```