tensorflow / **agents**

Branch: **master** ▾ **agents** / agents / ppo / **algorithm.py** Find file Copy path

vincentvanhoucke Fix obesrv typo 104a68b 7 days ago

2 contributors

559 lines (493 sloc) 23.1 KB

```python
# Copyright 2017 The TensorFlow Agents Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Proximal Policy Optimization algorithm.

Based on John Schulman's implementation in Python and Theano:
https://github.com/joschu/modular_rl/blob/master/modular_rl/ppo.py
"""

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import collections

import tensorflow as tf

from agents.ppo import memory
from agents.ppo import normalize
from agents.ppo import utility


_NetworkOutput = collections.namedtuple(
    'NetworkOutput', 'policy, mean, logstd, value, state')


class PPOAlgorithm(object):
  """A vectorized implementation of the PPO algorithm by John Schulman."""

  def __init__(self, batch_env, step, is_training, should_log, config):
    """Create an instance of the PPO algorithm.

    Args:
      batch_env: In-graph batch environment.
      step: Integer tensor holding the current training step.
      is_training: Boolean tensor for whether the algorithm should train.
      should_log: Boolean tensor for whether summaries should be returned.
      config: Object containing the agent configuration as attributes.
    """
    self._batch_env = batch_env
    self._step = step
    self._is_training = is_training
    self._should_log = should_log
    self._config = config
    self._observ_filter = normalize.StreamingNormalize(
        self._batch_env.observ[0], center=True, scale=True, clip=5,
        name='normalize_observ')
    self._reward_filter = normalize.StreamingNormalize(
        self._batch_env.reward[0], center=False, scale=True, clip=10,
        name='normalize_reward')
    # Memory stores tuple of observ, action, mean, logstd, reward.
    template = (
        self._batch_env.observ[0], self._batch_env.action[0],
        self._batch_env.action[0], self._batch_env.action[0],
        self._batch_env.reward[0])
    self._memory = memory.EpisodeMemory(
        template, config.update_every, config.max_length, 'memory')
    self._memory_index = tf.Variable(0, False)
    use_gpu = self._config.use_gpu and utility.available_gpus()
    with tf.device('/gpu:0' if use_gpu else '/cpu:0'):
      # Create network variables for later calls to reuse.
      self._network(
          tf.zeros_like(self._batch_env.observ)[:, None],
          tf.ones(len(self._batch_env)), reuse=None)
      cell = self._config.network(self._batch_env.action.shape[1].value)
      with tf.variable_scope('ppo_temporary'):
        self._episodes = memory.EpisodeMemory(
            template, len(batch_env), config.max_length, 'episodes')
        self._last_state = utility.create_nested_vars(
            cell.zero_state(len(batch_env), tf.float32))
        self._last_action = tf.Variable(
            tf.zeros_like(self._batch_env.action), False, name='last_action')
        self._last_mean = tf.Variable(
            tf.zeros_like(self._batch_env.action), False, name='last_mean')
        self._last_logstd = tf.Variable(
            tf.zeros_like(self._batch_env.action), False, name='last_logstd')
    self._penalty = tf.Variable(
        self._config.kl_init_penalty, False, dtype=tf.float32)
    self._policy_optimizer = self._config.policy_optimizer(
        self._config.policy_lr, name='policy_optimizer')
    self._value_optimizer = self._config.value_optimizer(
        self._config.value_lr, name='value_optimizer')

  def begin_episode(self, agent_indices):
    """Reset the recurrent states and stored episode.

    Args:
      agent_indices: 1D tensor of batch indices for agents starting an episode.

    Returns:
      Summary tensor.
    """
    with tf.name_scope('begin_episode/'):
      reset_state = utility.reinit_nested_vars(self._last_state, agent_indices)
      reset_buffer = self._episodes.clear(agent_indices)
      with tf.control_dependencies([reset_state, reset_buffer]):
        return tf.constant('')

  def perform(self, observ):
    """Compute batch of actions and a summary for a batch of observation.

    Args:
      observ: Tensor of a batch of observations for all agents.

    Returns:
      Tuple of action batch tensor and summary tensor.
    """
    with tf.name_scope('perform/'):
      observ = self._observ_filter.transform(observ)
      network = self._network(
          observ[:, None], tf.ones(observ.shape[0]), self._last_state)
      action = tf.cond(
          self._is_training, network.policy.sample, lambda: network.mean)
      logprob = network.policy.log_prob(action)[:, 0]
      # pylint: disable=g-long-lambda
      summary = tf.cond(self._should_log, lambda: tf.summary.merge([
          tf.summary.histogram('mean', network.mean[:, 0]),
          tf.summary.histogram('std', tf.exp(network.logstd[:, 0])),
          tf.summary.histogram('action', action[:, 0]),
          tf.summary.histogram('logprob', logprob)]), str)
      # Remember current policy to append to memory in the experience callback.
      with tf.control_dependencies([
          utility.assign_nested_vars(self._last_state, network.state),
          self._last_action.assign(action[:, 0]),
          self._last_mean.assign(network.mean[:, 0]),
          self._last_logstd.assign(network.logstd[:, 0])]):
        return tf.check_numerics(action[:, 0], 'action'), tf.identity(summary)

  def experience(self, observ, action, reward, unused_done, unused_nextob):
    """Process the transition tuple of the current step.

    When training, add the current transition tuple to the memory and update
    the streaming statistics for observations and rewards. A summary string is
    returned if requested at this step.

    Args:
      observ: Batch tensor of observations.
      action: Batch tensor of actions.
      reward: Batch tensor of rewards.
      unused_done: Batch tensor of done flags.
      unused_nextob: Batch tensor of successor observations.

    Returns:
      Summary tensor.
    """
    with tf.name_scope('experience/'):
      return tf.cond(
          self._is_training,
          lambda: self._define_experience(observ, action, reward), str)

  def _define_experience(self, observ, action, reward):
    """Implement the branch of experience() entered during training."""
    update_filters = tf.summary.merge([
        self._observ_filter.update(observ),
        self._reward_filter.update(reward)])
    with tf.control_dependencies([update_filters]):
      if self._config.train_on_agent_action:
        # NOTE: Doesn't seem to change much.
        action = self._last_action
      batch = observ, action, self._last_mean, self._last_logstd, reward
      append = self._episodes.append(batch, tf.range(len(self._batch_env)))
    with tf.control_dependencies([append]):
      norm_observ = self._observ_filter.transform(observ)
      norm_reward = tf.reduce_mean(self._reward_filter.transform(reward))
      # pylint: disable=g-long-lambda
      summary = tf.cond(self._should_log, lambda: tf.summary.merge([
          update_filters,
          self._observ_filter.summary(),
          self._reward_filter.summary(),
          tf.summary.scalar('memory_size', self._memory_index),
          tf.summary.histogram('normalized_observ', norm_observ),
          tf.summary.histogram('action', self._last_action),
          tf.summary.scalar('normalized_reward', norm_reward)]), str)
      return summary

  def end_episode(self, agent_indices):
    """Add episodes to the memory and perform update steps if memory is full.

    During training, add the collected episodes of the batch indices that
    finished their episode to the memory. If the memory is full, train on it,
    and then clear the memory. A summary string is returned if requested at
    this step.

    Args:
      agent_indices: 1D tensor of batch indices for agents starting an episode.

```

```python
    Returns:
       Summary tensor.
    """
    with tf.name_scope('end_episode/'):
      return tf.cond(
          self._is_training,
          lambda: self._define_end_episode(agent_indices), str)

  def _define_end_episode(self, agent_indices):
    """Implement the branch of end_episode() entered during training."""
    episodes, length = self._episodes.data(agent_indices)
    space_left = self._config.update_every - self._memory_index
    use_episodes = tf.range(tf.minimum(
        tf.shape(agent_indices)[0], space_left))
    episodes = [tf.gather(elem, use_episodes) for elem in episodes]
    append = self._memory.replace(
        episodes, tf.gather(length, use_episodes),
        use_episodes + self._memory_index)
    with tf.control_dependencies([append]):
      inc_index = self._memory_index.assign_add(tf.shape(use_episodes)[0])
    with tf.control_dependencies([inc_index]):
      memory_full = self._memory_index >= self._config.update_every
      return tf.cond(memory_full, self._training, str)

  def _training(self):
    """Perform multiple training iterations of both policy and value baseline.

    Training on the episodes collected in the memory. Reset the memory
    afterwards. Always returns a summary string.

    Returns:
      Summary tensor.
    """
    with tf.name_scope('training'):
      assert_full = tf.assert_equal(
          self._memory_index, self._config.update_every)
      with tf.control_dependencies([assert_full]):
        data = self._memory.data()
      (observ, action, old_mean, old_logstd, reward), length = data
      with tf.control_dependencies([tf.assert_greater(length, 0)]):
        length = tf.identity(length)
      observ = self._observ_filter.transform(observ)
      reward = self._reward_filter.transform(reward)
      policy_summary = self._update_policy(
          observ, action, old_mean, old_logstd, reward, length)
      with tf.control_dependencies([policy_summary]):
        value_summary = self._update_value(observ, reward, length)
      with tf.control_dependencies([value_summary]):
        penalty_summary = self._adjust_penalty(
            observ, old_mean, old_logstd, length)
      with tf.control_dependencies([penalty_summary]):
        clear_memory = tf.group(
            self._memory.clear(), self._memory_index.assign(0))
      with tf.control_dependencies([clear_memory]):
        weight_summary = utility.variable_summaries(
            tf.trainable_variables(), self._config.weight_summaries)
        return tf.summary.merge([
            policy_summary, value_summary, penalty_summary, weight_summary])

  def _update_value(self, observ, reward, length):
    """Perform multiple update steps of the value baseline.

    We need to decide for the summary of one iteration, and thus choose the one
    after half of the iterations.

    Args:
      observ: Sequences of observations.
      reward: Sequences of reward.
      length: Batch of sequence lengths.

    Returns:
      Summary tensor.
    """
    with tf.name_scope('update_value'):
      loss, summary = tf.scan(
          lambda _1, _2: self._update_value_step(observ, reward, length),
          tf.range(self._config.update_epochs_value),
          [0., ''], parallel_iterations=1)
      print_loss = tf.Print(0, [tf.reduce_mean(loss)], 'value loss: ')
      with tf.control_dependencies([loss, print_loss]):
        return summary[self._config.update_epochs_value // 2]

  def _update_value_step(self, observ, reward, length):
    """Compute the current value loss and perform a gradient update step.

    Args:
      observ: Sequences of observations.
      reward: Sequences of reward.
      length: Batch of sequence lengths.

    Returns:
      Tuple of loss tensor and summary tensor.
    """
    loss, summary = self._value_loss(observ, reward, length)
    gradients, variables = (
        zip(*self._value_optimizer.compute_gradients(loss)))
    optimize = self._value_optimizer.apply_gradients(
        zip(gradients, variables))
    summary = tf.summary.merge([
        summary,
        tf.summary.scalar('gradient_norm', tf.global_norm(gradients)),
        utility.gradient_summaries(
            zip(gradients, variables), dict(value=r'.*'))])
    with tf.control_dependencies([optimize]):
      return [tf.identity(loss), tf.identity(summary)]
```

```python

  def _value_loss(self, observ, reward, length):
    """Compute the loss function for the value baseline.

    The value loss is the difference between empirical and approximated returns
    over the collected episodes. Returns the loss tensor and a summary strin.

    Args:
      observ: Sequences of observations.
      reward: Sequences of reward.
      length: Batch of sequence lengths.

    Returns:
      Tuple of loss tensor and summary tensor.
    """
    with tf.name_scope('value_loss'):
      value = self._network(observ, length).value
      return_ = utility.discounted_return(
          reward, length, self._config.discount)
      advantage = return_ - value
      value_loss = 0.5 * self._mask(advantage ** 2, length)
      summary = tf.summary.merge([
          tf.summary.histogram('value_loss', value_loss),
          tf.summary.scalar('avg_value_loss', tf.reduce_mean(value_loss))])
      value_loss = tf.reduce_mean(value_loss)
      return tf.check_numerics(value_loss, 'value_loss'), summary

  def _update_policy(
      self, observ, action, old_mean, old_logstd, reward, length):
    """Perform multiple update steps of the policy.

    The advantage is computed once at the beginning and shared across
    iterations. We need to decide for the summary of one iteration, and thus
    choose the one after half of the iterations.

    Args:
      observ: Sequences of observations.
      action: Sequences of actions.
      old_mean: Sequences of action means of the behavioral policy.
      old_logstd: Sequences of action log stddevs of the behavioral policy.
      reward: Sequences of rewards.
      length: Batch of sequence lengths.

    Returns:
      Summary tensor.
    """
    with tf.name_scope('update_policy'):
      return_ = utility.discounted_return(
          reward, length, self._config.discount)
      value = self._network(observ, length).value
      if self._config.gae_lambda:
        advantage = utility.lambda_return(
            reward, value, length, self._config.discount,
            self._config.gae_lambda)
      else:
        advantage = return_ - value
      mean, variance = tf.nn.moments(advantage, axes=[0, 1], keep_dims=True)
      advantage = (advantage - mean) / (tf.sqrt(variance) + 1e-8)
      advantage = tf.Print(
          advantage, [tf.reduce_mean(return_), tf.reduce_mean(value)],
          'return and value: ')
      advantage = tf.Print(
          advantage, [tf.reduce_mean(advantage)],
          'normalized advantage: ')
      # pylint: disable=g-long-lambda
      loss, summary = tf.scan(
          lambda _1, _2: self._update_policy_step(
              observ, action, old_mean, old_logstd, advantage, length),
          tf.range(self._config.update_epochs_policy),
          [0., ''], parallel_iterations=1)
      print_loss = tf.Print(0, [tf.reduce_mean(loss)], 'policy loss: ')
      with tf.control_dependencies([loss, print_loss]):
        return summary[self._config.update_epochs_policy // 2]

  def _update_policy_step(
      self, observ, action, old_mean, old_logstd, advantage, length):
    """Compute the current policy loss and perform a gradient update step.

    Args:
      observ: Sequences of observations.
      action: Sequences of actions.
      old_mean: Sequences of action means of the behavioral policy.
      old_logstd: Sequences of action log stddevs of the behavioral policy.
      advantage: Sequences of advantages.
      length: Batch of sequence lengths.

    Returns:
      Tuple of loss tensor and summary tensor.
    """
    network = self._network(observ, length)
    loss, summary = self._policy_loss(
        network.mean, network.logstd, old_mean, old_logstd, action,
        advantage, length)
    gradients, variables = (
        zip(*self._policy_optimizer.compute_gradients(loss)))
    optimize = self._policy_optimizer.apply_gradients(
        zip(gradients, variables))
    summary = tf.summary.merge([
        summary,
        tf.summary.scalar('gradient_norm', tf.global_norm(gradients)),
        utility.gradient_summaries(
            zip(gradients, variables), dict(policy=r'.*'))])
    with tf.control_dependencies([optimize]):
      return [tf.identity(loss), tf.identity(summary)]

  def _policy_loss(
      self, mean, logstd, old_mean, old_logstd, action, advantage, length):
    """Compute the policy loss composed of multiple components.

    1. The policy gradient loss is importance sampled from the data-collecting
       policy at the beginning of training.
    2. The second term is a KL penalty between the policy at the beginning of
       training and the current policy.
    3. Additionally, if this KL already changed more than twice the target
       amount, we activate a strong penalty discouraging further divergence.

    Args:
      mean: Sequences of action means of the current policy.
      logstd: Sequences of action log stddevs of the current policy.
      old_mean: Sequences of action means of the behavioral policy.
      old_logstd: Sequences of action log stddevs of the behavioral policy.
      action: Sequences of actions.
      advantage: Sequences of advantages.
      length: Batch of sequence lengths.

    Returns:
      Tuple of loss tensor and summary tensor.
    """
    with tf.name_scope('policy_loss'):
      entropy = utility.diag_normal_entropy(mean, logstd)
      kl = tf.reduce_mean(self._mask(utility.diag_normal_kl(
          old_mean, old_logstd, mean, logstd), length), 1)
      policy_gradient = tf.exp(
          utility.diag_normal_logpdf(mean, logstd, action) -
          utility.diag_normal_logpdf(old_mean, old_logstd, action))
      surrogate_loss = -tf.reduce_mean(self._mask(
          policy_gradient * tf.stop_gradient(advantage), length), 1)
      kl_penalty = self._penalty * kl
      cutoff_threshold = self._config.kl_target * self._config.kl_cutoff_factor
      cutoff_count = tf.reduce_sum(
          tf.cast(kl > cutoff_threshold, tf.int32))
      with tf.control_dependencies([tf.cond(
          cutoff_count > 0,
          lambda: tf.Print(0, [cutoff_count], 'kl cutoff! '), int)]):
        kl_cutoff = (
            self._config.kl_cutoff_coef *
            tf.cast(kl > cutoff_threshold, tf.float32) *
            (kl - cutoff_threshold) ** 2)
      policy_loss = surrogate_loss + kl_penalty + kl_cutoff
      summary = tf.summary.merge([
          tf.summary.histogram('entropy', entropy),
          tf.summary.histogram('kl', kl),
          tf.summary.histogram('surrogate_loss', surrogate_loss),
          tf.summary.histogram('kl_penalty', kl_penalty),
          tf.summary.histogram('kl_cutoff', kl_cutoff),
          tf.summary.histogram('kl_penalty_combined', kl_penalty + kl_cutoff),
          tf.summary.histogram('policy_loss', policy_loss),
          tf.summary.scalar('avg_surr_loss', tf.reduce_mean(surrogate_loss)),
          tf.summary.scalar('avg_kl_penalty', tf.reduce_mean(kl_penalty)),
          tf.summary.scalar('avg_policy_loss', tf.reduce_mean(policy_loss))])
      policy_loss = tf.reduce_mean(policy_loss, 0)
      return tf.check_numerics(policy_loss, 'policy_loss'), summary

  def _adjust_penalty(self, observ, old_mean, old_logstd, length):
    """Adjust the KL policy between the behavioral and current policy.

    Compute how much the policy actually changed during the multiple
    update steps. Adjust the penalty strength for the next training phase if we
    overshot or undershot the target divergence too much.

    Args:
      observ: Sequences of observations.
      old_mean: Sequences of action means of the behavioral policy.
      old_logstd: Sequences of action log stddevs of the behavioral policy.
      length: Batch of sequence lengths.

    Returns:
      Summary tensor.
    """
    with tf.name_scope('adjust_penalty'):
      network = self._network(observ, length)
      assert_change = tf.assert_equal(
          tf.reduce_all(tf.equal(network.mean, old_mean)), False,
          message='policy should change')
      print_penalty = tf.Print(0, [self._penalty], 'current penalty: ')
      with tf.control_dependencies([assert_change, print_penalty]):
        kl_change = tf.reduce_mean(self._mask(utility.diag_normal_kl(
            old_mean, old_logstd, network.mean, network.logstd), length))
        kl_change = tf.Print(kl_change, [kl_change], 'kl change: ')
        maybe_increase = tf.cond(
            kl_change > 1.3 * self._config.kl_target,
            # pylint: disable=g-long-lambda
            lambda: tf.Print(self._penalty.assign(
                self._penalty * 1.5), [0], 'increase penalty '),
            float)
        maybe_decrease = tf.cond(
            kl_change < 0.7 * self._config.kl_target,
            # pylint: disable=g-long-lambda
            lambda: tf.Print(self._penalty.assign(
                self._penalty / 1.5), [0], 'decrease penalty '),
            float)
      with tf.control_dependencies([maybe_increase, maybe_decrease]):
        return tf.summary.merge([
            tf.summary.scalar('kl_change', kl_change),
            tf.summary.scalar('penalty', self._penalty)])

  def _mask(self, tensor, length):
    """Set padding elements of a batch of sequences to zero.

    Useful to then safely sum along the time dimension.

    Args:
      tensor: Tensor of sequences.
      length: Batch of sequence lengths.

    Returns:
      Masked sequences.
    """
    with tf.name_scope('mask'):
      range_ = tf.range(tensor.shape[1].value)
      mask = tf.cast(range_[None, :] < length[:, None], tf.float32)
      masked = tensor * mask
      return tf.check_numerics(masked, 'masked')

  def _network(self, observ, length=None, state=None, reuse=True):
    """Compute the network output for a batched sequence of observations.

    Optionally, the initial state can be specified. The weights should be
    reused for all calls, except for the first one. Output is a named tuple
    containing the policy as a TensorFlow distribution, the policy mean and log
    standard deviation, the approximated state value, and the new recurrent
    state.

    Args:
      observ: Sequences of observations.
      length: Batch of sequence lengths.
      state: Batch of initial recurrent states.
      reuse: Python boolean whether to reuse previous variables.

    Returns:
      NetworkOutput tuple.
    """
    with tf.variable_scope('network', reuse=reuse):
      observ = tf.convert_to_tensor(observ)
      use_gpu = self._config.use_gpu and utility.available_gpus()
      with tf.device('/gpu:0' if use_gpu else '/cpu:0'):
        observ = tf.check_numerics(observ, 'observ')
        cell = self._config.network(self._batch_env.action.shape[1].value)
        (mean, logstd, value), state = tf.nn.dynamic_rnn(
            cell, observ, length, state, tf.float32, swap_memory=True)
      mean = tf.check_numerics(mean, 'mean')
      logstd = tf.check_numerics(logstd, 'logstd')
      value = tf.check_numerics(value, 'value')
      policy = tf.contrib.distributions.MultivariateNormalDiag(
          mean, tf.exp(logstd))
      return _NetworkOutput(policy, mean, logstd, value, state)
```