03_lunar_lander_deep_q_learning

September 29, 2021

Double Deep Q-Learning & Open AI Gym: Intro

1 The Open AI Lunar Lander environment

The OpenAI Gym is a RL platform that provides standardized environments to test and benchmark RL algorithms using Python. It is also possible to extend the platform and register custom environments.

The Lunar Lander (LL) environment requires the agent to control its motion in two dimensions, based on a discrete action space and low-dimensional state observations that include position, orientation, and velocity. At each time step, the environment provides an observation of the new state and a positive or negative reward. Each episode consists of up to 1,000 time steps. The following diagram shows selected frames from a successful landing after 250 episodes by the agent we will train:

More specifically, the agent observes eight aspects of the state, including six continuous and two discrete elements. Based on the observed elements, the agent knows its location, direction, speed of movement, and whether it has (partially) landed. However, it does not know where it should be moving using its available actions or observe the inner state of the environment in the sense of understanding the rules that govern its motion.

At each time step, the agent controls its motion using one of four discrete actions. It can do nothing (and continue on its current path), fire its main engine (to reduce downward motion), or steer to the left or right using the respective orientation engines. There are no fuel limitations.

The goal is to land the agent between two flags on a landing pad at coordinates (0, 0), but landing outside of the pad is possible. The agent accumulates rewards in the range of 100-140 for moving toward the pad, depending on the exact landing spot. However, moving away from the target negates the reward the agent would have gained by moving toward the pad. Ground contact by each leg adds ten points, and using the main engine costs -0.3 points.

An episode terminates if the agent lands or crashes, adding or subtracting 100 points, respectively, or after 1,000 time steps. Solving LL requires achieving a cumulative reward of at least 200 on average over 100 consecutive episodes.

2 Deep Q-Learning

Deep Q learning estimates the value of the available actions for a given state using a deep neural network. It was introduced by Deep Mind's Playing Atari with Deep Reinforcement Learning (2013), where RL agents learned to play games solely from pixel input.

The Deep Q-Learning algorithm approximates the action-value function q by learning a set of weights of a multi-layered Deep Q Network (DQN) that maps states to actions so that

$$q(s, a, \theta) \approx q^*(s, a)$$

The algorithm applies gradient descent to a loss function defined as the squared difference between the DQN's estimate of the target

$$y_i = \mathbb{E}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1} \mid s, a)]$$

and its estimate of the action-value of the current state-action pair to learn the network parameters:

$$L_i(\theta_i) = \mathbb{E}\left[\left(\overbrace{\underbrace{y_i}_{\text{Q Target}} - \underbrace{Q(s, a; \theta)}_{\text{Current Prediction}}}^{\text{TD Error}}\right)^2\right]$$

Both the target and the current estimate depend on the set of weights, underlining the distinction from supervised learning where targets are fixed prior to training.

2.1 Extensions

Several innovations have improved the accuracy and convergence speed of deep Q-Learning, namely: - **Experience replay** stores a history of state, action, reward, and next state transitions and randomly samples mini-batches from this experience to update the network weights at each time step before the agent selects an -greedy action. It increases sample efficiency, reduces the autocorrelation of samples, and limits the feedback due to the current weights producing training samples that can lead to local minima or divergence. - **Slowly-changing target network** weakens the feedback loop from the current network parameters on the neural network weight updates. Also invented by by Deep Mind in Human-level control through deep reinforcement learning (2015), it use a slowly-changing target network that has the same architecture as the Q-network, but its weights are only updated periodically. The target network generates the predictions of the next state value used to update the Q-Networks estimate of the current state's value. - **Double deep Q-learning** addresses the bias of deep Q-Learning to overestimate action values because it purposely samples the highest action value. This bias can negatively affect the learning process and the resulting policy if it does not apply uniformly , as shown by Hado van Hasselt in Deep Reinforcement Learning with Double Q-learning (2015). To decouple the estimation of action values from the selection of actions, Double Deep Q-Learning (DDQN) uses the weights, of one network to select the best action given the next state, and the weights of another network to provide the corresponding action value estimate.

3 Imports & Settings

> See the notebook `04_q_learning_for_trading.ipynb` for instructions on upgrading TensorFlow to version 2.2, required by the code below..

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```python
%matplotlib inline
from time import time
from pathlib import Path

import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2

# OpenAI Gym
import gym
from gym import wrappers

import matplotlib.pyplot as plt
import seaborn as sns
```

[3]:
```python
sns.set_style('whitegrid', {'axes.grid' : False})
```

[4]:
```python
gpu_devices = tf.config.experimental.list_physical_devices('GPU')
if gpu_devices:
    print('Using GPU')
    tf.config.experimental.set_memory_growth(gpu_devices[0], True)
else:
    print('Using CPU')
```

```
Using GPU
```

Set random seeds to ensure results can be reproduced:

[5]:
```python
np.random.seed(42)
tf.random.set_seed(42)
```

3.1 Result display helper functions

[6]:
```python
def format_time(t):
    m_, s = divmod(t, 60)
    h, m = divmod(m_, 60)
    return '{:02.0f}:{:02.0f}:{:02.0f}'.format(h, m, s)
```

3.2 Enable virtual display to run from docker container

This is only required if you run this on server that does not have a display.

[7]:
```python
# from pyvirtualdisplay import Display
# virtual_display = Display(visible=0, size=(1400, 900))
# virtual_display.start()
```

4 Define DDQN Agent

We will use TensorFlow to create our Double Deep Q-Network .

4.1 Replay Buffer

[8]:
```python
class Memory():
    def __init__(self, capacity, state_dims):
        self.capacity = capacity
        self.idx = 0

        self.state_memory = np.zeros(shape=(capacity, state_dims),
                                     dtype=np.float32)
        self.new_state_memory = np.zeros_like(self.state_memory)

        self.action_memory = np.zeros(capacity, dtype=np.int32)
        self.reward_memory = np.zeros_like(self.action_memory)
        self.done = np.zeros_like(self.action_memory)

    def store(self, state, action, reward, next_state, done):
        self.state_memory[self.idx, :] = state
        self.new_state_memory[self.idx, :] = next_state
        self.reward_memory[self.idx] = reward
        self.action_memory[self.idx] = action
        self.done[self.idx] = 1 - int(done)
        self.idx += 1

    def sample(self, batch_size):
        batch = np.random.choice(self.idx, batch_size, replace=False)

        states = self.state_memory[batch]
        next_states = self.new_state_memory[batch]
        rewards = self.reward_memory[batch]
        actions = self.action_memory[batch]
        done = self.done[batch]
        return states, actions, rewards, next_states, done
```

4.2 Agent Class

[9]:
```
class DDQNAgent:
    def __init__(self,
                 state_dim,
                 num_actions,
                 gamma,
                 epsilon_start,
                 epsilon_end,
                 epsilon_decay_steps,
                 epsilon_exponential_decay,
                 learning_rate,
                 architecture,
                 l2_reg,
                 replay_capacity,
                 tau,
                 batch_size,
                 results_dir,
                 log_every=10):

        self.state_dim = state_dim
        self.num_actions = num_actions

        self.architecture = architecture
        self.l2_reg = l2_reg
        self.learning_rate = learning_rate
        self.experience = Memory(replay_capacity,
                                 state_dim)
        self.gamma = gamma
        self.tau = tau
        self.batch_size = batch_size
        self.idx = np.arange(batch_size, dtype=np.int32)

        self.online_network = self.build_model()
        self.target_network = self.build_model(trainable=False)
        self.optimizer = Adam(lr=learning_rate)
        self.update_target()

        self.epsilon = epsilon_start
        self.epsilon_decay_steps = epsilon_decay_steps
        self.epsilon_decay = (epsilon_start - epsilon_end) / epsilon_decay_steps
        self.epsilon_exponential_decay = epsilon_exponential_decay
        self.epsilon_history = []

        self.total_steps = self.train_steps = 0
        self.episodes = self.episode_length = self.train_episodes = 0
        self.steps_per_episode = []
```

```
        self.episode_reward = 0
        self.rewards_history = []

        self.results_dir = results_dir
        self.experiment = experiment
        self.log_every = log_every

        self.summary_writer = (tf.summary
                               .create_file_writer(results_dir.as_posix()))
        self.start = time()
        self.train = True

    def build_model(self, trainable=True):
        layers = []
        for i, units in enumerate(self.architecture, 1):
            layers.append(Dense(units=units,
                                input_dim=self.state_dim if i == 1 else None,
                                activation='relu',
                                kernel_regularizer=l2(self.l2_reg),
                                trainable=trainable))
        layers.append(Dense(units=self.num_actions,
                            trainable=trainable))
        return Sequential(layers)

    def update_target(self):
        self.target_network.set_weights(self.online_network.get_weights())

    # @tf.function
    def epsilon_greedy_policy(self, state):
        self.total_steps += 1
        if np.random.rand() <= self.epsilon:
            return np.random.choice(self.num_actions)
        q = self.online_network.predict(state)
        return np.argmax(q, axis=1).squeeze()

    # @tf.function
    def decay_epsilon(self):
        if self.train:
            if self.episodes < self.epsilon_decay_steps:
                self.epsilon -= self.epsilon_decay
            else:
                self.epsilon *= self.epsilon_exponential_decay

    def log_progress(self):
        self.rewards_history.append(self.episode_reward)
        self.steps_per_episode.append(self.episode_length)
```

```
        avg_steps_100 = np.mean(self.steps_per_episode[-100:])
        avg_steps_10 = np.mean(self.steps_per_episode[-10:])
        max_steps_10 = np.max(self.steps_per_episode[-10:])
        avg_rewards_100 = np.mean(self.rewards_history[-100:])
        avg_rewards_10 = np.mean(self.rewards_history[-10:])
        max_rewards_10 = np.max(self.rewards_history[-10:])

        with self.summary_writer.as_default():
            tf.summary.scalar('Episode Reward', self.episode_reward, step=self.
↪episodes)
            tf.summary.scalar('Episode Rewards (MA 100)', avg_rewards_100,␣
↪step=self.episodes)
            tf.summary.scalar('Episode Steps', self.episode_length, step=self.
↪episodes)
            tf.summary.scalar('Epsilon', self.epsilon, step=self.episodes)

        if self.episodes % self.log_every == 0:
            template = '{:03} | {} | Rewards {:4.0f} {:4.0f} {:4.0f} | ' \
                       'Steps: {:4.0f} {:4.0f} {:4.0f} | Epsilon: {:.4f}'
            print(template.format(self.episodes, format_time(time() - self.
↪start),
                                  avg_rewards_100, avg_rewards_10,␣
↪max_rewards_10,
                                  avg_steps_100, avg_steps_10, max_steps_10,
                                  self.epsilon))

    def memorize_transition(self, s, a, r, s_prime, done):
        self.experience.store(s, a, r, s_prime, done)
        self.episode_reward += r
        self.episode_length += 1

        if done:
            self.epsilon_history.append(self.epsilon)
            self.decay_epsilon()
            self.episodes += 1
            self.log_progress()
            self.episode_reward = 0
            self.episode_length = 0

    def experience_replay(self):
        # not enough experience yet
        if self.batch_size > self.experience.idx:
            return

        # sample minibatch
        states, actions, rewards, next_states, done = self.experience.
↪sample(self.batch_size)
```

```
        # select best next action (online)
        next_action = tf.argmax(self.online_network.predict(next_states, self.
↪batch_size), axis=1, name='next_action')
        # predict next q values (target)
        next_q_values = self.target_network.predict(next_states, self.
↪batch_size)
        # get q values for best next action
        target_q = (tf.math.reduce_sum(next_q_values *
                                       tf.one_hot(next_action,
                                                  self.num_actions),
                                       axis=1, name='target_q'))
        # compute td target
        td_target = rewards + done * self.gamma * target_q

        with tf.GradientTape() as tape:
            q_values = self.online_network(states)
            q_values = tf.math.reduce_sum(q_values * tf.one_hot(actions, self.
↪num_actions), axis=1, name='q_values')
            loss = tf.math.reduce_mean(tf.square(td_target - q_values))

        # run back propagation
        variables = self.online_network.trainable_variables
        gradients = tape.gradient(loss, variables)

        self.optimizer.apply_gradients(zip(gradients, variables))
        with self.summary_writer.as_default():
            tf.summary.scalar('Loss', loss, step=self.train_steps)
        self.train_steps += 1

        if self.total_steps % self.tau == 0:
            self.update_target()

    def store_results(self):
        result = pd.DataFrame({'Rewards': self.rewards_history,
                               'Steps'  : self.steps_per_episode,
                               'Epsilon': self.epsilon_history},
                              index=list(range(1, len(self.rewards_history) +␣
↪1)))

        result.to_csv(self.results_dir / 'results.csv', index=False)
```

5 Run Experiment

[10]:
```
experiment = 0
```

[11]:
```
results_dir = Path('results', 'lunar_lander', 'experiment_{}'.
 ↪format(experiment))
if not results_dir.exists():
    results_dir.mkdir(parents=True)
```

5.1 Set up OpenAI Gym Lunar Lander Environment

We will begin by instantiating and extracting key parameters from the LL environment:

[12]:
```
env = gym.make('LunarLander-v2')
state_dim = env.observation_space.shape[0]  # number of dimensions in state
num_actions = env.action_space.n  # number of actions
max_episode_steps = env.spec.max_episode_steps  # max number of steps per␣
 ↪episode
env.seed(42)
```

[12]:
```
[42]
```

We will also use the built-in wrappers that permit the periodic storing of videos that display the agent's performance:

[13]:
```
monitor_path = results_dir / 'monitor'
video_freq = 500
```

[14]:
```
env = wrappers.Monitor(env,
                       directory=monitor_path.as_posix(),
                       video_callable=lambda count: count % video_freq == 0,
                      force=True)
```

5.2 Define hyperparameters

The agent's performance is quite sensitive to several hyperparameters. We will start with the discount before setting the Q-Network, replay buffer, and -greedy policy parameters.

5.2.1 Discount Factor

[15]:
```
gamma = .99
```

5.2.2 Q-Network Parameters

[16]:
```
learning_rate = 0.0001
```

[17]:
```
architecture = (256, 256)  # units per layer
l2_reg = 1e-6  # L2 regularization
```

We will update the target network every 100 time steps, store up to 1 m past episodes in the replay memory, and sample mini-batches of 1,024 from memory to train the agent:

5.2.3 Replay Buffer Parameters

[18]:
```
tau = 100  # target network update frequency
replay_capacity = int(1e6)
batch_size = 1024
```

5.2.4 -greedy Policy

The -greedy policy starts with pure exploration at =1 and linearly decays every step to 0.01 over 100 episodes, followed by exponential decay at rate 0.99:

[19]:
```
epsilon_start = 1.0
epsilon_end = 0.01
epsilon_decay_steps = 100
epsilon_exponential_decay = .99
```

5.3 Instantiate DDQN Agent

[20]:
```
agent = DDQNAgent(state_dim=state_dim,
                  num_actions=num_actions,
                  learning_rate=learning_rate,
                  gamma=gamma,
                  epsilon_start=epsilon_start,
                  epsilon_end=epsilon_end,
                  epsilon_decay_steps=epsilon_decay_steps,
                  epsilon_exponential_decay=epsilon_exponential_decay,
                  replay_capacity=replay_capacity,
                  architecture=architecture,
                  l2_reg=l2_reg,
                  tau=tau,
                  batch_size=batch_size,
                  results_dir=results_dir)
```

5.4 Train & test agent

[21]:
```
tf.keras.backend.clear_session()
```

[22]:
```
max_episodes = 2500
test_episodes = 0
```

Besides the episode number an elapsed time, we log the moving averages for the last 100 and last 10 rewards and episode lengths, as well as their respective maximum values over the last 10 iterations. We also track the decay of epsilon.

[23]:
```
while agent.episodes < max_episodes:
    this_state = env.reset()
    done = False
    while not done:
        action = agent.epsilon_greedy_policy(this_state.reshape(-1, state_dim))
        next_state, reward, done, _ = env.step(action)
        agent.memorize_transition(this_state, action, reward, next_state, done)
        agent.experience_replay()
        this_state = next_state
    if np.mean(agent.rewards_history[-100:]) > 200:
        break

agent.store_results()
env.close()
```

```
010 | 00:00:03 | Rewards -193 -193  -70 | Steps:   93   93  143 | Epsilon:
0.9010
020 | 00:00:29 | Rewards -157 -122  -59 | Steps:   95   97  153 | Epsilon:
0.8020
030 | 00:00:59 | Rewards -140 -105  -43 | Steps:   95   96  129 | Epsilon:
0.7030
040 | 00:01:37 | Rewards -129  -98  -29 | Steps:  101  116  188 | Epsilon:
0.6040
050 | 00:02:25 | Rewards -114  -51   25 | Steps:  108  137  277 | Epsilon:
0.5050
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-23-b27f98ec606e> in <module>
      6         next_state, reward, done, _ = env.step(action)
      7         agent.memorize_transition(this_state, action, reward,␣
 ↪next_state, done)
----> 8         agent.experience_replay()
      9         this_state = next_state
     10     if np.mean(agent.rewards_history[-100:]) > 200:

<ipython-input-9-a46dafd8269c> in experience_replay(self)
    136         next_action = tf.argmax(self.online_network.predict(next_states ␣
 ↪self.batch_size), axis=1, name='next_action')
    137         # predict next q values (target)
--> 138         next_q_values = self.target_network.predict(next_states, self.
 ↪batch_size)
    139         # get q values for best next action
    140         target_q = (tf.math.reduce_sum(next_q_values *
```

```
~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
↪tensorflow/python/keras/engine/training.py in _method_wrapper(self, *args,␣
↪**kwargs)
     86 from tensorflow.tools.docs import doc_controls
     87
---> 88
     89 # pylint: disable=g-import-not-at-top
     90 try:

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
↪tensorflow/python/keras/engine/training.py in predict(self, x, batch_size,␣
↪verbose, steps, callbacks, max_queue_size, workers, use_multiprocessing)
   1238         """Runs an evaluation execution with multiple steps."""
   1239         for _ in math_ops.range(self._steps_per_execution):
-> 1240           outputs = step_function(self, iterator)
   1241         return outputs
   1242

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
↪tensorflow/python/keras/engine/data_adapter.py in __init__(self, x, y,␣
↪sample_weight, batch_size, steps_per_epoch, initial_epoch, epochs, shuffle,␣
↪class_weight, max_queue_size, workers, use_multiprocessing, model)
   1098
   1099     adapter_cls = select_data_adapter(x, y)
-> 1100     self._adapter = adapter_cls(

   1101         x,
   1102         y,

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
↪tensorflow/python/keras/engine/data_adapter.py in __init__(self, x, y,␣
↪sample_weights, sample_weight_modes, batch_size, epochs, steps, shuffle,␣
↪**kwargs)
    360       dataset = dataset.map(shuffle_batch)
    361
--> 362     self._dataset = dataset
    363
    364   def slice_inputs(self, indices_dataset, inputs):

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
↪tensorflow/python/data/ops/dataset_ops.py in flat_map(self, map_func)
   1650       Dataset: A `Dataset`.
   1651
-> 1652     Raises:
   1653       ValueError: If a component has an unknown rank, and  the␣
↪`padded_shapes`
   1654         argument is not set.
```

```
~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/tensorflow/python/data/ops/dataset_ops.py in __init__(self, input_dataset, map_func)
   4068                           (value, output_type))
   4069     return value
-> 4070
   4071
   4072 def _padding_values_or_default(padding_values, input_dataset):

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/tensorflow/python/data/ops/dataset_ops.py in __init__(self, func, transformation_name, dataset, input_classes, input_shapes, input_types, input_structure, add_to_graph, use_legacy_function, defun_kwargs)
   3219
   3220
-> 3221 class _NestedVariant(composite_tensor.CompositeTensor):
   3222
   3223   def __init__(self, variant_tensor, element_spec, dataset_shape):

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/tensorflow/python/eager/function.py in get_concrete_function(self, *args, **kwargs)
   2529     """Returns a string summarizing this function's signature.
   2530
-> 2531     Args:
   2532       default_values: If true, then include default values in the signature.
   2533

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/tensorflow/python/eager/function.py in _get_concrete_function_garbage_collected(self, *args, **kwargs)
   2494     return self._args_to_indices
   2495
-> 2496   @property
   2497   def kwargs_to_include(self):
   2498     return self._kwargs_to_include

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/tensorflow/python/eager/function.py in _maybe_define_function(self, args, kwargs)
   2775
   2776   try:
-> 2777     flatten_inputs = nest.flatten_up_to(

   2778         input_signature,
   2779         inputs[:len(input_signature)],
```

```
~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
↪tensorflow/python/eager/function.py in _create_graph_function(self, args,␣
↪kwargs, override_flat_arg_shapes)
   2655               if i not in self._arg_indices_to_default_values
   2656           ]
-> 2657           raise TypeError("{} missing required arguments: {}".format(

   2658               self.signature_summary(), ", ".join(missing_args)))
   2659

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
↪tensorflow/python/framework/func_graph.py in func_graph_from_py_func(name,␣
↪python_func, args, kwargs, signature, func_graph, autograph,␣
↪autograph_options, add_control_dependencies, arg_names, op_return_value,␣
↪collections, capture_by_value, override_flat_arg_shapes)
    979               raise
    980
--> 981         # Wrapping around a decorator allows checks like tf_inspect.
↪getargspec
    982         # to be accurate.
    983         converted_func = tf_decorator.make_decorator(original_func,␣
↪wrapper)

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
↪tensorflow/python/data/ops/dataset_ops.py in wrapper_fn(*args)
   3212
   3213   def _inputs(self):
-> 3214     return []
   3215
   3216   @property

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
↪tensorflow/python/data/ops/dataset_ops.py in _wrapper_helper(*args)
   3154     """See `Dataset.from_tensor_slices()` for details."""
   3155     element = structure.normalize_element(element)
-> 3156     batched_spec = structure.type_spec_from_value(element)
   3157     self._tensors = structure.to_batched_tensor_list(batched_spec,␣
↪element)
   3158     self._structure = nest.map_structure(

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
↪tensorflow/python/autograph/impl/api.py in wrapper(*args, **kwargs)
    260     # dealing with the extra loop increment operation that the for
    261     # canonicalization creates.
--> 262     node = continue_statements.transform(node, ctx)
    263     node = return_statements.transform(node, ctx)
    264     if ctx.user.options.uses(converter.Feature.LISTS):
```

```
~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/autograph/impl/api.py in converted_call(f, args, kwargs,␣
 ↪caller_fn_scope, options)
    490          ' @tf.autograph.experimental.do_not_convert')
    491    if isinstance(exc, errors.UnsupportedLanguageElementError):
--> 492      if not conversion.is_in_allowlist_cache(f, options):
    493        logging.warn(warning_template, f, '', exc)
    494    else:

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/autograph/impl/api.py in _call_unconverted(f, args, kwargs,␣
 ↪options, update_cache)
    344      if caller_fn_scope is None:
    345        raise ValueError('either caller_fn_scope or options must have a␣
 ↪value')
--> 346      options = caller_fn_scope.callopts
    347
    348    if conversion.is_in_allowlist_cache(f, options):

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/keras/engine/data_adapter.py in slice_batch_indices(indices
    345              indices, [num_in_full_batch], [self._partial_batch_size]))
    346          flat_dataset = flat_dataset.concatenate(index_remainder)
--> 347
    348        if shuffle == "batch":
    349          # 1024 is a magic constant that has not been properly evaluated

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/ops/array_ops.py in slice(input_, begin, size, name)
   1035        packed_begin = packed_end = packed_strides = var_empty
   1036      return strided_slice(
-> 1037          tensor,
   1038          packed_begin,
   1039          packed_end,

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/ops/gen_array_ops.py in _slice(input, begin, size, name)
   9092        _ops.raise_from_not_ok_status(e, name)
   9093      except _core._FallbackException:
-> 9094        pass
   9095      try:
   9096        return scatter_nd_non_aliasing_add_eager_fallback(

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/framework/op_def_library.py in␣
 ↪_apply_op_helper(op_type_name, name, **keywords)
    463                                  "earlier arguments." %
    464                                  (prefix, dtype.name))
```

```
--> 465             else:
    466               raise TypeError("%s that don't all match." % prefix)
    467           else:

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/framework/ops.py in convert_to_tensor(value, dtype, name,␣
 ↪as_ref, preferred_dtype, dtype_hint, ctx, accepted_result_types)
   1339
   1340 @tf_export("convert_to_tensor", v1=[])
-> 1341 @dispatch.add_dispatch_support
   1342 def convert_to_tensor_v2_with_dispatch(
   1343     value, dtype=None, dtype_hint=None, name=None):

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/framework/constant_op.py in␣
 ↪_constant_tensor_conversion_function(v, dtype, name, as_ref)
    319         x = _eager_fill(shape.as_list(), _eager_identity(t, ctx), ctx)
    320       return _eager_identity(x, ctx)
--> 321     else:
    322       return _eager_fill(shape.as_list(), t, ctx)
    323   raise TypeError("Eager execution of tf.constant with unsupported shap␣
 ↪"

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/framework/constant_op.py in constant(value, dtype, shape,␣
 ↪name)
    259
    260   Raises:
--> 261     TypeError: if shape is incorrectly specified or unsupported.
    262     ValueError: if called on a symbolic tensor.
    263   """

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/framework/constant_op.py in _constant_impl(value, dtype,␣
 ↪shape, name, verify_shape, allow_broadcast)
    296   return const_tensor
    297
--> 298
    299 def _constant_eager_impl(ctx, value, dtype, shape, verify_shape):
    300   """Implementation of eager constant."""

~/.pyenv/versions/miniconda3-latest/envs/ml4t-dl/lib/python3.8/site-packages/
 ↪tensorflow/python/framework/tensor_util.py in make_tensor_proto(values, dtype␣
 ↪shape, verify_shape, allow_broadcast)
    430           dtypes.qint8, dtypes.quint8, dtypes.qint16, dtypes.quint16,
    431           dtypes.qint32
--> 432       ])

    433
```

```
    434     if _is_array_like(values):

KeyboardInterrupt:
```

5.5 Evaluate Results

[ ]:
```
results = pd.read_csv(results_dir / 'results.csv')
results['MA100'] = results.rolling(window=100, min_periods=25).Rewards.mean()
```

[ ]:
```
fig, axes = plt.subplots(ncols=2, figsize=(16, 4), sharex=True)
results[['Rewards', 'MA100']].plot(ax=axes[0])
axes[0].set_ylabel('Rewards')
axes[0].set_xlabel('Episodes')
axes[0].axhline(200, c='k', ls='--', lw=1)
results[['Steps', 'Epsilon']].plot(secondary_y='Epsilon', ax=axes[1]);
axes[1].set_xlabel('Episodes')
fig.suptitle('Double Deep Q-Network Agent | Lunar Lander', fontsize=16)
fig.tight_layout()
fig.subplots_adjust(top=.9)
fig.savefig(results_dir / 'trading_agent_2ed', dpi=300)
```

5.6 Tensorboard

[ ]:
```
%load_ext tensorboard
```

[ ]:
```
%tensorboard --logdir results/lunar_lander/experiment_0
```

[ ]:
```

```