Deep Reinforcement Learning: Value- and Policy-Based Methods

Machine Learning, CS 5333/7333

In this project you will implement and experiment with (a) Q-learning with neural net based function approximation for continuous spaces to solve the "cartpole" problem and (b) REINFORCE algorithms to solve the "inverted pendulum" problem in "gym" package developed by openAI. You also try the variants of these methods (e.g., Double Deep Q-Learning (DQN) or Actor-Critic) to obtain faster and/or more stable learning performance.

Submit your code and report analyzing learning performance over episodes, in terms of cumulative utility obtained per episode. You should report results from experiments varying the NN architecture and learning rate, activation function, etc.

1 Q-learning for discrete action, continuous state spaces

You will implement a Q-learning agent to solve the "CartPole-v1" problem. A brief description of the environment could be found here: `https://gym.openai.com/envs/CartPole-v1/`. In "CartPole-v1", the state space is 4-dimensional and continuous, while the action space is 1-dimensional and discrete, with two actions: 0 and 1. You would use artificial neural networks to learn the Q-values for state-action pairs. The maximum number of steps in this game is 500.

2 REINFORCE for continuous action and state spaces

You will implement a policy-based agent to solve the "InvertedPendulum-v2" problem, which is similar to the previous environment but with 1D continuous action space. The visualization of this environment can be found here: `https://gym.openai.com/envs/InvertedPendulum-v2/`. You would use policy networks to build a agent. The maximum number of steps in this game is 1000, and the action space interval is $[-3.0, 3.0]$.

3 Implementation Tips

- You don't need and suppose not to understand what the values in state/action variable refer to.
- The documentation of "gym" is here: `https://gym.openai.com/docs/`, which presents examples about how to run an environment, and how to get the state variable and rewards.
- If you want to find some basic information for an environment (e.g., maximum number of steps), you could get it with: *gym.spec("environment_name")*
- if you want some information about action/state space (e.g., range of the values), you could find it with: *env.action_space* or *env.observation_space*
- Around 2000 episodes/games and relatively simple networks with naive methods would train a decent agent, which only takes about 10-20 minutes when training on Google Colab. Sometimes only few hundreds of episodes are sufficient as the naive methods are relatively unstable and the agent will perform worse if it's trained with more episodes, so you could use early stopping and/or learning rate decay in this case.
- For continuous action with constrains, the output for the policy network could be the entire or partial set of parameters for some distribution, then sample one or more instances with this parameter set. Next, you could use tanh or sigmoid function to map the sampled instances into the action space.

4 Q-learning with function approximation on Colab

Run the following command to setup the environment:

```
!sudo apt install python3.8 python3.8-dev python3-pip
!update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.8 1
!update-alternatives --list python3
!sudo update-alternatives --set python3 /usr/bin/python3.8
!python3 --version
!python3.8 -m pip install --upgrade pip
!pip3.8 -V
!pip3.8 install gym tensorflow pygame
```

Create a python script named "q_learning.py", and add the following code to file:

```
from gym.version import VERSION
print(VERSION) # make sure the new version of gym is loaded

import gym
import numpy as np
import tensorflow as tf

class Agent:
	def __init__(self, obs_shape, act_size):
		self.obs_shape = obs_shape
		self.act_size = act_size

	def network(self, train=True):
		inputs = tf.keras.Input(shape=(self.obs_shape,), name="input")
		x = tf.keras.layers.Dense(64, activation=tf.keras.layers.LeakyReLU(), name="
		    ↪ dense_1")(inputs)
		x = tf.keras.layers.Dense(64, activation=tf.keras.layers.LeakyReLU(), name="
		    ↪ dense_2")(x)
		outputs = tf.keras.layers.Dense(self.act_size, name="output")(x)
		model = tf.keras.models.Model(inputs=inputs, outputs=outputs, name="nn",
		    ↪ trainable=train)
		return model


class Util:
	def __init__(self):
		self.optimizer = tf.keras.optimizers.Adam(learning_rate=tf.keras.optimizers.
		    ↪ schedules.ExponentialDecay(0.01, decay_steps=50, decay_rate=0.9))
		self.history = []

	def record_history(self, current_state, action, reward, next_state):
		self.history.append([current_state, action, reward, next_state])

	def td_loss(self, nn, discount=0.99):
		loss = []
		for current_state, action, reward, next_state in self.history:
			binary_action = [0.0] * nn.output.shape[1]
			binary_action[action] = 1.0
			binary_action = tf.constant([binary_action])
			q_current = nn(tf.convert_to_tensor([current_state]))
			max_q_next = tf.math.reduce_max(nn(tf.convert_to_tensor([next_state])))
			loss.append(tf.math.square((reward + discount * max_q_next - q_current) *
			    ↪ binary_action))
```

```
                return tf.math.reduce_mean(loss, axis=0)

        def update_model(self, nn):
                with tf.GradientTape() as tape:
                        loss = self.td_loss(nn)
                grads = tape.gradient(loss, nn.trainable_variables)
                self.optimizer.apply_gradients(zip(grads, nn.trainable_variables))
                self.history = []


env = gym.make('CartPole-v1')
agent = Agent(4, 2).network()
utility = Util()

# train
epsilon = 0.3
i, early_stop = 0, 0
n_game = 2000
while i < n_game:
        current_state = env.reset()
        step = 0
        while True:
                if np.random.uniform() < epsilon:
                        action = env.action_space.sample()
                else:
                        action = tf.math.argmax(tf.reshape(agent(tf.convert_to_tensor([
                            ↪ current_state])), [-1])).numpy()
                next_state, reward, done, info = env.step(action)
                step += 1
                utility.record_history(current_state, action, reward, next_state)
                current_state = next_state
                if len(utility.history) == 50:
                        utility.update_model(agent)
                epsilon = max(epsilon * 0.99, 0.05)
                if done:
                        print(i, step)
                        i += 1
                        if step >= 500:
                                early_stop += 1
                        else:
                                early_stop = 0
                        if early_stop >= 10:
                                i = n_game
                        break


# test
for i in range(10):
        env.close()
        env = gym.make('CartPole-v1')
        state = env.reset()
        step = 0
        while True:
                action = tf.math.argmax(tf.reshape(agent(tf.convert_to_tensor([state])), [-1])).
                    ↪ numpy()
                state, reward, done, info = env.step(action)
```

```
            step += 1
            if done:
                print(i, step)
                break
env.close()

# save agent
# agent.save("cartpole_dql")
```

Then, go back to Colab and run:

```
!python3.8 q_learning.py
```

5 REINFORCE on Colab

Run the following command to load environment:
(from https://gist.github.com/BuildingAtom/3119ac9c595324c8001a7454f23bf8c8)

```
#Include this at the top of your colab code
import os
if not os.path.exists('.mujoco_setup_complete'):
  # Get the prereqs
  !apt-get -qq update
  !apt-get -qq install -y libosmesa6-dev libgl1-mesa-glx libglfw3 libgl1-mesa-dev libglew-dev
      ↪ patchelf
  # Get Mujoco
  !mkdir ~/.mujoco
  !wget -q https://mujoco.org/download/mujoco210-linux-x86_64.tar.gz -O mujoco.tar.gz
  !tar -zxf mujoco.tar.gz -C "$HOME/.mujoco"
  !rm mujoco.tar.gz
  # Add it to the actively loaded path and the bashrc path (these only do so much)
  !echo 'export␣LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/.mujoco/mujoco210/bin' >> ~/.bashrc
  !echo 'export␣LD_PRELOAD=$LD_PRELOAD:/usr/lib/x86_64-linux-gnu/libGLEW.so' >> ~/.bashrc
  # THE ANNOYING ONE, FORCE IT INTO LDCONFIG SO WE ACTUALLY GET ACCESS TO IT THIS SESSION
  !echo "/root/.mujoco/mujoco210/bin" > /etc/ld.so.conf.d/mujoco_ld_lib_path.conf
  !ldconfig
  # Install Mujoco-py
  !pip3 install -U 'mujoco-py<2.2,>=2.1'
  # run once
  !touch .mujoco_setup_complete

try:
  if _mujoco_run_once:
    pass
except NameError:
  _mujoco_run_once = False
if not _mujoco_run_once:
  # Add it to the actively loaded path and the bashrc path (these only do so much)
  try:
    os.environ['LD_LIBRARY_PATH']=os.environ['LD_LIBRARY_PATH'] + ':/root/.mujoco/mujoco210/bin
        ↪ '
  except KeyError:
    os.environ['LD_LIBRARY_PATH']='/root/.mujoco/mujoco210/bin'
  try:
    os.environ['LD_PRELOAD']=os.environ['LD_PRELOAD'] + ':/usr/lib/x86_64-linux-gnu/libGLEW.so'
  except KeyError:
```

```
    os.environ['LD_PRELOAD']='/usr/lib/x86_64-linux-gnu/libGLEW.so'
  # presetup so we don't see output on first env initialization
  import mujoco_py
  _mujoco_run_once = True
```

Then run the following code for a simply policy gradient agent:

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


class Agent:
	def __init__(self, obs_shape, act_size):
		self.obs_shape = obs_shape
		self.act_size = act_size

	def network(self, train=True):
		inputs = tf.keras.Input(shape=(self.obs_shape,), name="input")
		x = tf.keras.layers.Dense(256, activation=tf.keras.layers.LeakyReLU(), name="
		    ↪ dense_1")(inputs)
		outputs = tf.keras.layers.Dense(2 * self.act_size, name="output")(x)
		model = tf.keras.models.Model(inputs=inputs, outputs=outputs, name="nn",
		    ↪ trainable=train)
		return model

	@staticmethod
	def action(obs, nn):
		mean, log_std = nn(tf.convert_to_tensor([obs])).numpy().flatten()
		sample = np.random.normal(mean, np.exp(log_std))
		return sample, 3 * np.tanh(sample)


class Util:
	def __init__(self):
		self.optimizer = tf.keras.optimizers.Adam(learning_rate=0.0005)
		self.state, self.sample, self.reward = [], [], []
		self.data = {"state": [], "sample": [], "reward": []}

	def history_for_one_game(self, state, sample, reward):
		self.state.append(state)
		self.sample.append(sample)
		self.reward.append(reward)

	def discount_reward(self, gamma=0.99):
		discount_reward = []
		for index, item in enumerate(self.reward[::-1]):
			if index == 0:
				discount_reward.append(item)
			else:
				discount_reward.append(item + gamma * discount_reward[-1])
		return discount_reward[::-1]

	def dataset(self):
		self.data["state"] += self.state
		self.data["sample"] += self.sample
```

```
                self.data["reward"] += self.discount_reward()
                self.state, self.sample, self.reward = [], [], []

        def update_model(self, model):
                self.dataset()
                with tf.GradientTape() as tape:
                        reward = tf.cast(tf.convert_to_tensor(self.data["reward"]), dtype=tf.
                            ↪ float32)
                        pred = model(tf.convert_to_tensor(self.data["state"]))
                        mean, std = pred[:, 0], tf.math.exp(pred[:, 1])
                        log_prob = tf.math.log(tf.math.exp(-((tf.cast(tf.convert_to_tensor(self.
                            ↪ data["sample"]), dtype=tf.float32) - mean) / std) ** 2 / 2) / (std
                            ↪  * tf.math.sqrt(2 * math.pi)))
                        loss = - reward * log_prob
                grads = tape.gradient(loss, model.trainable_variables)
                self.optimizer.apply_gradients(zip(grads, model.trainable_variables))
                self.data = {"state": [], "sample": [], "reward": []}


env = gym.make('InvertedPendulum-v2')
agent = Agent(4, 1).network()
utility = Util()

# train
i, early_stop = 0, 0
n_game = 2000
while i < n_game:
        state = env.reset()
        step = 0
        while True:
                sample, action = Agent.action(state, agent)
                next_state, reward, done, info = env.step([action])
                step += 1
                utility.history_for_one_game(state, sample, reward)
                state = next_state
                if done:
                        utility.update_model(agent)
                        print(i, step)
                        i += 1
                        if step >= 1000:
                                early_stop += 1
                        else:
                                early_stop = 0
                        if early_stop >= 20:
                                i = n_game
                        break

# test
for i in range(10):
        env.close()
        env = gym.make('InvertedPendulum-v2')
        state = env.reset()
        step = 0
        while True:
                _, action = Agent.action(state, agent)
                state, reward, done, info = env.step([action])
```

```
            step += 1
            if done:
                print(i, step)
                break

env.close()

# save agent
# agent.save("InvertedPendulum_policy")
```