**14**

**An Introduction to Reinforcement Learning**

**Reinforcement learning is the training of machine learning models to solve a problem by making a sequence of decisions. The three components of reinforcement learning are agent, environment, and actions. An** environment **is a problem to be solved. An** agent **is an algorithm that interacts with the environment to solve a problem.** Actions **are an agent’s interactions with the environment.**

**An agent receives observations from an environment and takes actions. The environment uses rewards and punishments as signals for positive or negative behavior based on the actions taken by the agent. So the agent learns how to solve a problem by trial and error interactions with the environment. The objective is to make decisions that maximizes expected rewards over time.**

**Although the designer sets the reward policy, he (or she) gives the learning model no hints or suggestions on how to solve the problem. The model learns** on its own **how to maximize reward through trial and error interactions between the agent and environment.**

**A reward policy is the set of rules that maximizes the reward function for a given reinforcement learning environment. So a** reward function **stipulates what the designer wants the agent to accomplish.**

# ****Challenges of Reinforcement Learning****

**The main challenge of reinforcement learning is** creating the environment**. Creating an effective environment has three issues.**

**First, the environment is highly dependent on the problem to be solved. So it is contextually specific. The context (of the problem domain) drives design of each reinforcement learning environment to solve a specific set of tasks. For instance, an environment for a self-driving car is not transferable to one for a self-flying drone.**

**Second, the reward policy is finite and structured. Environments for chess, Go, and Atari games are relatively simple. No matter how complex these games might appear to be, their rules are structured and finite. Even an extremely complex environment for a self-driving car is finite and structured. Although such an environment must deal with many unknowns, its reward policy is created by designers. Even the most brilliant designers on the planet can’t create an infinite reward policy.**

**Third, the room for error where safety is a concern is pretty much zero. Reinforcement learning experiments in the health care industry come to mind. Such models must be tested and tweaked in many stages before they are ready for even simple testing. Transferring the model out of the testing phase into to the real world is where the real work begins.**

**Scaling and tweaking the neural network controlling the agent is another challenge. The only way to communicate with the network is through its system of rewards and penalties. With this single conduit comes the possibility of catastrophic forgetting (or catastrophic interference). Catastrophic forgetting is the tendency of a neural network to completely and abruptly forget previously learned information upon learning new information. So when a neural network acquires new information some of the old information is erased from the network. Catastrophic forgetting occurs because many of the weights (where information is stored) of the neural network are changed when new information is learned, which makes it less likely that prior knowledge is kept intact. During sequential learning, new inputs coming into a neural network can erase original input weights.**

**Another challenge is reaching a local optimum. A local optimum is the best solution to a problem within a small neighborhood of possible solutions. In contrast, a global optimum is the optimal solution when** every **possible solution is considered. A local optimum would be walking when the goal is to learn how to move (walk, run, and jump). In this case, the agent performs the task in a suboptimal (locally optimal) way, but not in an (globally) optimal way.**

**A final challenge is securing talented designers. A competent data scientist may not be hard to find, but the average salary can be well over $150,000 per year. Moreover, it may be difficult to find one with experience in creating a specific type of environment because environments are contextually specific.**

**As reinforcement learning quickly evolves, more challenges continue to arise. The good news is that researchers continue to work hard to mitigate issues with current reinforcement learning models as they surface.**

Notebooks for chapters are located at the following URL:

https://github.com/paperd/deep-learning-models

**We demonstrate reinforcement learning with a code experiment using a** very simple **environment.** Begin setting up the Colab ecosystem by importing the main TensorFlow library and instantiating the GPU.

# Import the TensorFlow Library

Import the library and alias it as **tf**:

import tensorflow as tf

Aliasing the TensorFlow library as tf is common practice.

# GPU Hardware Accelerator

As a convenience, we provide the steps to enable the GPU in a Colab notebook:

1. click *Runtime* in the top left menu
2. click *Change runtime type* from the drop-down menu
3. choose *GPU* from the *Hardware accelerator* drop-down menu
4. click *SAVE*

Verify that the GPU is active:

tf.\_\_version\_\_, tf.test.gpu\_device\_name()

If '/device:GPU:0' is displayed, the GPU is active. If '..' is displayed, the regular CPU is active.

**Note:** If you get the error name 'tf' is not defined, re-execute the code to import the TensorFlow library!

# Reinforcement Learning Experiment

We demonstrate a simple reinforcement learning experiment with Cart-Pole. Cart-Pole is a game in which a pole is attached by an unactuated joint to a cart that moves along a frictionless track. The goal of the game is to keep the pole vertically upright. The starting state is randomly initialized between -0.05 and 0.05. The starting state includes the cart position, cart velocity, pole angle, and pole velocity. The pole velocity is measure at the tip of the pole. The Cart-Pole game is in 2D space. So the cart can only move left and right to balance the pole.

For the experiment, we need a working environment to train an agent. Instead of creating our own environment, we can use OpenAI Gym. OpenAI Gym is a toolkit that provides a wide variety of simulated environments including Atari games, board games, and (2D and 3D) physical simulations.

## Install and Configure OpenAI Gym on Colab

Most of the requirements of python packages (for OpenAI Gym) are already fulfilled on Colab. But we still need to install prerequisites:

!pip install gym

!apt-get install python-opengl -y

!apt install xvfb -y

We also need to install libraries for displaying output rendered from the environment:

!pip install pyvirtualdisplay

!pip install piglet

The pyvirtualdisplay library enables virtual display. The piglet library provides an object-oriented API for the creation of games and other multimedia applications.

## Import Libraries

Import and activate the virtual display library:

import pyvirtualdisplay

display = pyvirtualdisplay.Display(

    visible=0, size=(1400, 900)).start()

We used 1400 x 900 for our virtual display. Feel free to experiment with the size of the display.

Import the gym library:

import gym

The gym library is a collection of environments designed for testing and developing reinforcement learning algorithms. It saves us from having to create complicated environments on our own.

## Create an Environment

Create the Cart-Pole environment:

env = gym.make('CartPole-v1')

The Cart-Pole environment is a 2D simulation that accelerates a cart left or right to balance a pole placed on top of it. A pole is attached by an unactuated joint to a cart that moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum (or pole) starts upright and the goal is to prevent it from falling over.

Initialize the environment:

env.seed(0)

obs = env.reset()

obs

Initialize the environment with the reset method. After it initializes, the method returns an observation.

Observations vary depending on the environment. In this case, an observation is a 1D NumPy array composed of 4 floats that represent the cart's horizontal position, velocity, angle of the pole (0 = vertical), and angular velocity. Any positive number indicates movement to the right for angle of the pole and angular velocity. Any negative number indicates movement to the left. For horizontal position, a negative number means that the pole is tilting left and a positive number tilting right. For velocity, a positive number means the cart is speeding up and a negative number slowing down.

The initial rendering state (with seed of 0) produces the 1D NumPy array:

array([-0.04456399, 0.04653909, 0.01326909, -0.02099827])

So the initial state of the pole is not completely horizontal (obs[0] is slightly negative), its velocity is slowly increasing (obs[1] is slightly positive), the pole is angled slightly to the right (obs[2] is slightly positive), and the angular velocity is going toward the left (obs[3] is slightly negative).

An environment can be visualized by calling its render method and we can pick the rendering mode (the rendering options depend on the environment). Display the rendering environment at its initial state:

env.render()

The initial state is True because we just initialized it.

## Display the Rendering from the Environment

Set mode='rgb\_array' to get an image of the environment as a NumPy array:

img = env.render(mode='rgb\_array')

img.shape

The image shape is rendered from the Cart-Pole environment.

Create a function to display the rendered image of the pole position on the cart from the environment as configured in Listing 14-1.

def plot\_environment(env, figsize=(5,4)):

  plt.figure(figsize=figsize)

  img = env.render(mode='rgb\_array')

  plt.imshow(img)

  plt.axis('off')

  return img

Listing 14-1. Function to Display the Rendered Environment

Display:

import matplotlib.pyplot as plt

plot\_environment(env)

plt.show()

We see the rendering from the environment in it initialized state. So the pole is very nearly vertical on the cart, but tilted slightly to the right.

## Display Actions

Let's see how to interact with the environment we created. The agent needs to select an action from an action space. An **action space** is the set of possible actions that an agent can take.

Ask the environment about possible actions:

env.action\_space

Discrete(2) means that the possible actions for the Cart-Pole environment are integers 0 and 1. Accelerating left is 0 and accelerating right is 1. So the environment's action space has two possible actions, which means that the agent can accelerate towards the left or towards the right. Of course, other environments may have additional discrete actions or other kinds of actions like continuous ones.

**Note:** The Cart-Pole environment is the simplest one that can be created! Real world reinforcement learning environment have enormous action spaces containing many possible actions.

Reset the environment and see how the pole is leaning by looking at its angle:

env.seed(0)

obs = env.reset()

indx = 2

obs[indx]

The third position (index of 2) in the obs array is the angle of the pole. If the value is below 0, the pole angles to the left. If above 0, it angles to the right. The value is barely over zero. So the pole is tilted slightly toward the right because obs[2] is > 0.

**Note:** We didn’t have to reset the environment again because we did so in the previous section. But an environment should be reset at the beginning of an experiment. So we did this again just to instill a good reinforcement learning habit.

As we already know, the Cart-Pole environment only has two actions, left (0) or right (1). Let's accelerate the cart toward the right by setting action=1:

action = 1

obs, reward, done, info = env.step(action)

print ('obs array:', obs)

print ('reward:', reward)

print ('done:', done)

print ('info:', info)

The step method executes the given action and returns four values. obs is the new observation. The cart is now moving toward the right because obs[1] > 0. The pole is still tilted toward the right because obs[2] > 0, but its angular velocity is now negative because obs[3] < 0. So it will likely be tilted toward the left after the next step.

In this simplest of environments, reward is always 1.0 at every step. So the goal is to keep the episode running as long as possible. The done value is True when the episode is over. The episode is over if the pole tilts too much, goes off the screen or we win the game. The info value provides extra information. In this case, there is no extra information. Once we finish using an environment, call the close method to free resources.

The environment tells the agent each new observation, the reward, when the game is over, and information it got during the last step. Display the pole position:

plot\_environment(env)

plt.show()

The pole is still tilted toward the right.

Display the reward that the agent received in the last step:

reward

Of course, the reward is 1.0 because it is always 1.0 in this simplest of experiments.

Test if the game is over:

done

Since the value is False, the game is not over.

An **episode** is the sequence of steps between the moment the environment is reset until it is done. At the end of an episode (i.e., when the step method returns done=True), reset the environment before continuing to use it.

To automatically reset when an episode is over:

if done:

  obs = env.reset()

else:

  print ('game is not over!')

## Simple Neural Network Reward Policy

How can we make the poll remain upright? We need to define a reward policy. In action, a reward policy is simply the strategy the agent uses to select an action at each step. The agent can use all past actions and observations to decide what to do.

Let's create a neural network that takes observations as inputs and output the action to take for each observation. To choose an action, our network estimates a probability for each action and randomly selects an action based on the estimated probabilities. In the case of the Cart-Pole environment, there are just two possible actions (left or right). So we only need one output neuron that outputs the probability p of action 0 (left) and the probability 1 - p of action 1 (right).

Clear previous models and generate a seed:

import numpy as np

tf.keras.backend.clear\_session()

tf.random.set\_seed(0)

np.random.seed(0)

Determine the observation space:

obs\_space = env.observation\_space.shape

obs\_space

The observation space is another term for the reward policy. The observation space (as shown earlier in the observation array) is a 1D NumPy array composed of 4 floats that represent the cart's horizontal position, velocity, angle of the pole (0 = vertical), and angular velocity. So the observation space is 4.

**Note:** The terms policy, reward policy, observation array, and observation space are interchangeable.

Set the number of inputs for the policy network:

n\_inputs = env.observation\_space.shape[0]

n\_inputs

Create the policy network:

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

model = Sequential([

  Dense(5, activation='elu', input\_shape=[n\_inputs]),

  Dense(1, activation='sigmoid')

])

The policy network is a simple Sequential neural network. The number of inputs is the size of the observation space, which is 4 in our case. We include only 5 neurons in the first layer because this is such a simple problem. We only need to output a single probability (the probability of going left). So we use sigmoid activation in the output layer to generate a single output neuron as a logit. We only need to output a single probability p because we can get the probability of going right with 1 - p. If we had more than two possible actions, we would still use one output neuron per action and substitute softmax activation in the output layer.

In this particular environment, past actions and observations can safely be ignored because each observation contains the environment's full state. If there were some hidden state, we might need to consider past actions and observations to try to infer the hidden state of the environment. For example, if the environment only revealed the position of the cart but not its velocity, we have to consider not only the current observation but also the previous observation in order to estimate the current velocity. Another example is if the observations are noisy, we might want to use the past few observations to estimate the most likely current state. Our problem is as simple as can be because the current observation is noise-free and contains the environment's full state.

Why do we pick a random action based on the probability given by the policy network rather than just picking the action with the highest probability? Because this approach lets the agent find the right balance between exploring new actions and exploiting the actions that are known to work well.

## Model Predictions

Create a function that runs the model to play one episode and return the frames so we can display an animation as shown in Listing 14-2.

def render\_policy\_net(model, n\_max\_steps=200, seed=0):

  frames = []

  env = gym.make('CartPole-v1')

  env.seed(seed)

  np.random.seed(seed)

  obs = env.reset()

  for step in range(n\_max\_steps):

    frames.append(env.render(mode='rgb\_array'))

    left\_proba = model.predict(obs.reshape(1, -1))

    action = int(np.random.rand() > left\_proba)

    obs, reward, done, info = env.step(action)

    if done:

      break

  env.close()

  return frames

Listing 14-2. Function to Return the Frames from One Episode

Establish the Cart-Pole environment and reset it. Create a loop to run a number of steps until the episode is over. Begin each step by appending the visualization of the environment rendering to the frames list. Continue by making an action prediction from the model. Next, establish an action based on the prediction. Execute the step method based on the action. Continue looping until the episode is over. End by returning the list of frames.

Create functions to show animation of the frames as shown in Listing 14-3.

import matplotlib.animation as animation

import matplotlib as mpl

def update\_scene(num, frames, patch):

  patch.set\_data(frames[num])

  return patch,

def plot\_animation(frames, repeat=False, interval=40):

  fig = plt.figure()

  patch = plt.imshow(frames[0])

  plt.axis('off')

  anim = animation.FuncAnimation(

      fig, update\_scene, fargs=(frames, patch), blit=True,

      frames=len(frames), repeat=repeat, interval=interval)

  plt.close()

  return anim

Listing 14-3. Functions to Animate Frames

The plot\_animation function makes an animation by repeatedly calling the update\_scene function. The plot\_animation function accepts the frames list. The function continues by extracting an image (patch) from the frames list. It then calls upadate\_scene to set the x and y coordinates for the patch. The animation is returned batched on the patch coordinates for each frame image.

Create the frames list based on the policy network:

frames = render\_policy\_net(model)

## Animate

Create the animation:

anim = plot\_animation(frames, interval=100)

Experiment with the interval parameter to see its impact on the animation. We set interval size to 100 just because we liked the result.

Render and display the animation. We show two ways to accomplish this. The first method uses the HTML library to display HTML elements:

from IPython.display import HTML

method1 = HTML(anim.to\_html5\_video())

method1

The animation is rendered to html5 video with the to\_html5\_video method and displayed with HTML module.

The second method uses the runtime configuration library:

from matplotlib import rc

method2 = rc('animation', html='html5')

To implement the second method, just run the animation object:

anim

Ugh! The pole is falling to the left! The reason is that we’ve yet to implement a reward policy.

## Implement a Basic Reward Policy

In the previous section, we just had the policy network run random predictions with no interaction with the agent. If the pole is tilting left, the agent has no way of knowing whether this is a bad or good action. We need to develop an environment with a reward policy that the agent can interact with to learn how to balance the pole on the cart.

Create a diverse environment space as shown in Listing 14-4.

n\_environments = 50

n\_iterations = 5000

envs = [gym.make(

    'CartPole-v1') for \_ in range(n\_environments)]

for index, env in enumerate(envs):

  env.seed(index)

np.random.seed(0)

observations = [env.reset() for env in envs]

optimizer = tf.keras.optimizers.RMSprop()

loss\_fn = tf.keras.losses.binary\_crossentropy

for iteration in range(n\_iterations):

  target\_probas = np.array(

      [([1.] if obs[2] < 0 else [0.])

      for obs in observations])

  with tf.GradientTape() as tape:

    left\_probas = model(np.array(observations))

    loss = tf.reduce\_mean(

        loss\_fn(target\_probas, left\_probas))

    print('\rIteration: {}, Loss: {:.3f}'.\

          format(iteration, loss.numpy()), end='')

    grads = tape.gradient(loss, model.trainable\_variables)

    optimizer.apply\_gradients(

        zip(grads, model.trainable\_variables))

    actions = (np.random.rand(n\_environments, 1) >\

               left\_probas.numpy()).astype(np.int32)

  for env\_index, env in enumerate(envs):

    obs, reward, done, info = env.step(

        actions[env\_index][0])

    observations[env\_index] = obs if not done else env.reset()

for env in envs:

  env.close()

Listing 14-4. Environment Space for the Experiment

We create an environment space with 50 different environments in parallel to provide a diverse training batch at each step in the network. We train the network for 5,000 iterations. Of course, you can tweak the number of environments and iterations, but be cognizant of available computer resources.

We use the RMSProp optimizer because it seems to work pretty well. You can experiment with different optimizers to see how well the agent learns the task. We use binary cross-entropy for the loss function because we only have two discrete possible actions (left or right). The first line in the iteration loop checks the angle of the pole. If the angle < 0, the target action is left (proba(left) = 1.). Otherwise, the target action is right (proba(left) = 0).

The gradient tape section records agent’s actions during training. The tilt of the pole is determined. The agent takes actions to steady the pole on the cart. The actions are then fed to the 50 environments that determine the rewards given to the agent. This process repeats 5,000 times.

Finally, the environments are reset to free resources. We train with a custom training loop so we can easily use the predictions at each training step to advance the environments.

Create the frames for an animation:

frames = render\_policy\_net(model)

We now have the frames based on what the agent learned during training. We use the frames to create an animation.

Animate:

anim = plot\_animation(frames, repeat=True, interval=100)

anim

Voilà! We taught the agent how to balance the pole on the cart. But can we do even better?

## Reinforce Policy Gradient Algorithm

We have yet to demonstrate the real breakthrough of reinforcement learning. In the previous section, we taught the agent with a reward policy. But can the agent can learn a better policy on its own?

We can use a reinforce policy gradient algorithm to automate agent learning. Policy gradients optimize the parameters of a policy by following the gradients toward higher rewards.

### How Do Policy Gradients Work?

To use policy gradients, let the neural network policy play the game several times. At each step, compute the gradients that make the chosen action even more likely. But don’t apply the gradients at this point.

After running several episodes, compute each action's advantage with a discount factor at each step. A discount factor is computed by evaluating an action based on the sum of all rewards that come after the action. If an action's advantage is positive, the action was probably good. So **now** apply the gradients to make the action more likely to be chosen in the future. If it is negative, apply the opposite gradients to make the action less likely to be chosen. Finally, compute the mean of all resultant gradient vectors and use it to perform a Gradient Descent step. The mean of all resultant gradient vectors is calculated by taking the mean of each gradient (or opposite gradient) multiplied by its action advantage.

### Train the Model with Policy Gradients

Begin by creating functions to play a step, play multiple episodes, discount rewards, and normalize discounted rewards.

Create a function that plays one step as shown in Listing 14-5.

def play\_one\_step(env, obs, model, loss\_fn):

  with tf.GradientTape() as tape:

    left\_proba = model(obs[np.newaxis])

    action = (tf.random.uniform([1, 1]) > left\_proba)

    y\_target = tf.constant(

        [[1.]]) - tf.cast(action, tf.float32)

    loss = tf.reduce\_mean(loss\_fn(

        y\_target, left\_proba))

  grads = tape.gradient(loss, model.trainable\_variables)

  obs, reward, done, info = env.step(

      int(action[0, 0].numpy()))

  return obs, reward, done, grads

Listing 14-5. Function to Play a Single Step

In the GradientTape block, call the model with a single observation. Reshape the observation so that it becomes a batch containing a single instance (the model expects a batch). Get a probability of going left by sampling a random float between 0 and 1 and checking if it is greater than the probability. The action is False if the probability is left\_proba and True if the probability is 1 - left\_proba. Cast this Boolean (True or False) to a number of 0 (left) or 1 (right) with the appropriate probabilities. We then define the target probabilities of going left (1 - the action) or right (the action). If the action is 0 (left), the target probability of going left is 1. If the action is 1 (right), the target probability is 0.

Continue by computing the loss and use the tape to compute the gradient of the loss with regard to the model's trainable variables. Tweak the gradients later depending on how good or bad that action turned out to be. Finally, play the selected action and return the new observation, reward, whether the episode is over or not, and the gradients.

Create a function to play multiple episodes and return the rewards and gradients for each episode and each step as shown in Listing 14-6.

def play\_multiple\_episodes(

    env, n\_episodes, n\_max\_steps, model, loss\_fn):

  all\_rewards = []

  all\_grads = []

  for episode in range(n\_episodes):

    current\_rewards = []

    current\_grads = []

    obs = env.reset()

    for step in range(n\_max\_steps):

      obs, reward, done, grads = play\_one\_step(

          env, obs, model, loss\_fn)

      current\_rewards.append(reward)

      current\_grads.append(grads)

      if done:

        break

    all\_rewards.append(current\_rewards)

    all\_grads.append(current\_grads)

  return all\_rewards, all\_grads

Listing 14-6. Play Multiple Episodes Function

The function returns a list of reward lists by calling the play\_one\_step function for the number of desired steps. This list contains one reward list per episode. Each reward list contains one reward per step. The function also returns a list of gradient lists. This list contains one gradient list per episode. Each gradient list contains one tuple of gradients per step. Each tuple contains one gradient tensor per trainable variable.

Simply, the policy gradient algorithm uses the play\_multiple\_episodes function to play the game several times. It then goes back and looks at all the rewards to discount and normalize them.

### Discount and Normalize the Rewards

To discount and normalize rewards, we discount the rewards and normalize them. The first function discounts rewards as shown in Listing 14-7.

def discount\_rewards(rewards, discount\_rate):

  discounted = np.array(rewards)

  for step in range(len(rewards) - 2, -1, -1):

    discounted[step] += discounted[step + 1] \* discount\_rate

  return discounted

Listing 14-7. Discount Rewards Function

Verify that the function works:

discount\_rewards([10, 0, -50], discount\_rate=0.8)

We give the function 3 actions. After each action, there is a reward. The first reward is 10, the second is 0, and the third is -50. We use a discount factor of 80%. So the third action gets -50 (full credit for the last reward). But the second action only gets -40 (80% credit for the last reward). Finally, the first action gets 80% of -40 (-32) plus full credit for the first reward (+10) leading to a discounted reward of -22.

The second function normalizes discounted rewards as shown in Listing 14-8.

def discount\_and\_normalize\_rewards(

    all\_rewards, discount\_rate):

  all\_discounted\_rewards =\

    [discount\_rewards(rewards, discount\_rate)

    for rewards in all\_rewards]

  flat\_rewards = np.concatenate(all\_discounted\_rewards)

  reward\_mean = flat\_rewards.mean()

  reward\_std = flat\_rewards.std()

  return [(discounted\_rewards - reward\_mean) / reward\_std

          for discounted\_rewards in all\_discounted\_rewards]

Listing 14-8. Normalize Discounted Rewards Function

To normalize all discounted rewards across all episodes, compute the mean and standard deviation of all the discounted rewards. Subtract the mean from each discounted reward and divide by the standard deviation. Let’s try this function with two episodes:

discount\_and\_normalize\_rewards(

    [[10, 0, -50], [10, 20]], discount\_rate=0.8)

All actions from the first episode (first array) are considered bad because normalized advantages are all negative. This makes sense because the sum of the rewards is -40 (10 + 0 + -50). Conversely, the second episode actions (second array) are good because normalized advantages are positive. The sum of the rewards is 30 (10 + 20).

### Train the Learner

As we know, reinforcement learning uses a learning model and an environment space to enable an agent to learn how to solve a problem. So the agent learns how to solve a problem during the training process. So we can think of the process as training the learner.

Begin by defining a set of hyperparameters:

n\_iterations = 150

n\_episodes\_per\_update = 10

n\_max\_steps = 200

discount\_rate = 0.95

We set 150 training iterations. Of course, you can tweak this number and the other values. Play 10 episodes of the game per iteration, make each episode last at most 200 steps, and use a discount rate of 0.95.

Define an optimizer and loss function for the policy network:

optimizer = tf.keras.optimizers.Adam(lr=0.01)

loss\_fn = tf.keras.losses.binary\_crossentropy

Use binary cross\_entropy because we are training a binary classifier (two possible actions: left or right).

Generate a seed, clear previous models, can create a simple policy network:

tf.keras.backend.clear\_session()

np.random.seed(0)

tf.random.set\_seed(0)

model = Sequential([

  Dense(5, activation='elu', input\_shape=[4]),

  Dense(1, activation='sigmoid'),

])

A policy network in reinforcement learning is very simple. Creating an environment space is the difficult part.

Train the learner as shown in Listing 14-9.

env = gym.make('CartPole-v1')

env.seed(42);

for iteration in range(n\_iterations):

  all\_rewards, all\_grads = play\_multiple\_episodes(

      env, n\_episodes\_per\_update, n\_max\_steps,

      model, loss\_fn)

  total\_rewards = sum(map(sum, all\_rewards))

  print('\rIteration: {}, mean rewards: {:.1f}'.format(

      iteration, total\_rewards / n\_episodes\_per\_update),

      end='')

  all\_final\_rewards = discount\_and\_normalize\_rewards(

      all\_rewards, discount\_rate)

  all\_mean\_grads = []

  for var\_index in range(len(model.trainable\_variables)):

    mean\_grads = tf.reduce\_mean(

        [final\_reward \* all\_grads[episode\_index][step][var\_index]

         for episode\_index, final\_rewards in enumerate(

             all\_final\_rewards)

           for step, final\_reward in enumerate(

               final\_rewards)], axis=0)

    all\_mean\_grads.append(mean\_grads)

  optimizer.apply\_gradients(

      zip(all\_mean\_grads, model.trainable\_variables))

env.close()

Listing 14-9. Train the Learner

Call play\_multiple\_episodes (at each training iteration) to play the game 10 times and return all the rewards and gradients for every episode and step. Call discount\_and\_normalize\_rewards to compute each action's normalized advantage, which gives us a measure of how good or bad each action actually was in hindsight. For each trainable variable, compute the weighted mean of the gradients over all episodes and all steps weighted by the final\_reward. The final\_reward is each action's normalized advantage. End by applying the mean gradients using the optimizer, which tweaks the model's trainable variables to hopefully make the policy a bit better.

**Note:** Training the learner takes some time. So be patient.

### Render the Frames from the Reinforce Policy Gradient Algorithm

Render the frames:

frames\_ra = render\_policy\_net(model)

### Animate the Policy

Animate:

anim = plot\_animation(frames\_ra, repeat=True, interval=100)

anim

The pole appears to be a bit less wobbly. It’s kind of amazing that the agent learned a better policy on its own!

# Summary

We introduced the concept of reinforcement learning and demonstrated it with a simple experiment. The code in the experiment is quite complex even for the simplest of environment spaces. The evolution of reinforcement learning applied to real world problems lies in the ability of designers to create environment spaces that realistically represent the world we live in.