Machine learning is usually classified into three different paradigms: **supervised learning**, **unsupervised learning**, and **reinforcement learning (RL)**. Supervised learning requires labeled data and has been the most popularly used machine learning paradigm so far. However, applications based on unsupervised learning, which does not require labels, have been steadily on the rise, especially in the form of generative models.

An RL, on the other hand, is a different branch of machine learning that is considered to be the closest we have reached in terms of emulating how humans learn. It is an area of active research and development and is in its early stages, with some promising results. A prominent example is the famous AlphaGo model, built by Google's DeepMind, that defeated the world's best Go player.

In supervised learning, we usually feed the model with atomic input-output data pairs and hope for the model to learn the output as a function of the input. In RL, we are not keen on learning such individual input to individual output functions. Instead, we are interested in learning a strategy (or policy) that enables us to take a sequence of steps (or actions), starting from the input (state), in order to obtain the final output or achieve the final goal.

Looking at a photo and deciding whether it's a cat or a dog is an atomic input-output learning task that can be solved through supervised learning. However, looking at a chess board and deciding the next move with the aim of winning the game requires strategy, and we need RL for complex tasks like these.

In the previous chapters, we came across examples of supervised learning such as building a classifier to classify handwritten digits using the MNIST dataset. We also explored unsupervised learning while building a text generation model using an unlabeled text corpus.

In this chapter, we will uncover some of the basic concepts of RL and **deep reinforcement learning (DRL)**. We will then focus on a specific and popular type of DRL model – the **deep Q-learning Network** (**DQN**) model. Using PyTorch, we will build a DRL application. We will train a DQN model to learn how to play the game of Pong against a computer opponent (bot).

By the end of this chapter, you will have all the necessary context to start working on your own DRL project in PyTorch. Additionally, you will have hands-on experience of building a DQN model for a real-life problem. The skills you'll have gained in this chapter will be useful for working on other such RL problems.

This chapter is broken down into the following topics:

- Reviewing reinforcement learning concepts
- Discussing Q-learning
- Understanding deep Q-learning
- Building a DQN model in PyTorch

Reviewing reinforcement learning concepts

In a way, RL can be defined as learning from mistakes. Instead of getting the feedback for every data instance, as is the case with supervised learning, the feedback is received after a sequence of actions. The following diagram shows the high-level schematic of an RL system:

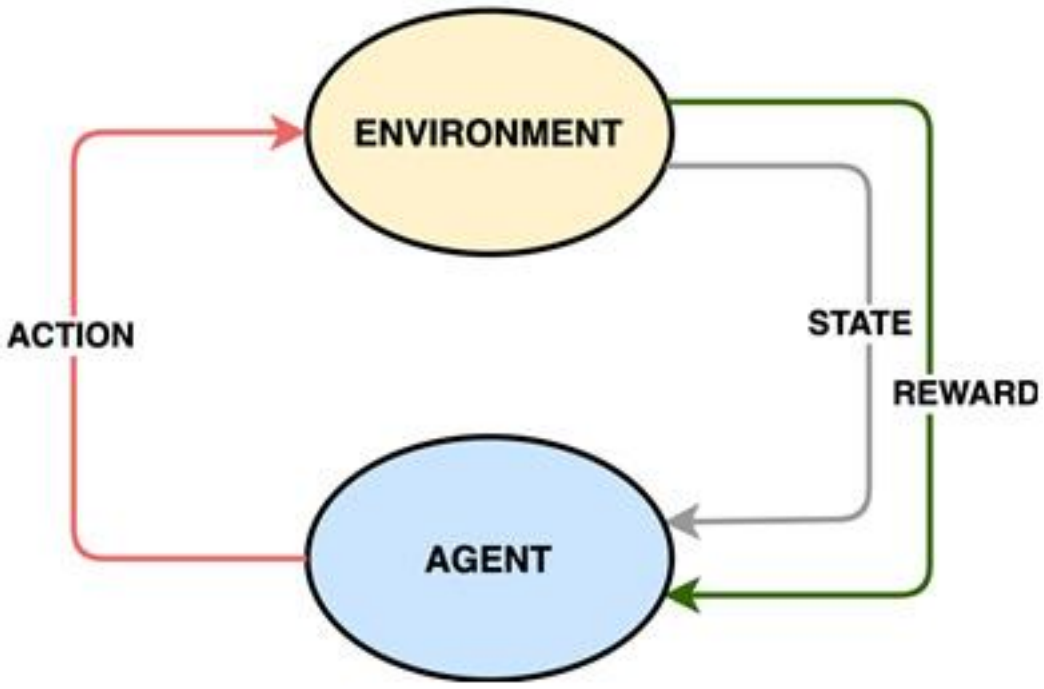

Figure 11. 1 – Reinforcement learning schematic

In an RL setting, we usually have an **agent**, which does the learning. The agent learns to make decisions and take **actions** according to these decisions. The agent operates within a provided **environment**. This environment can be thought of as a confined world where the agent lives, takes actions, and learns from its actions. An action here is simply the implementation of the decision the agent makes based on what it has learned.

We mentioned earlier that unlike supervised learning, RL does not have an output for each and every input; that is, the agent does not necessarily receive a feedback for each and every action. Instead, the agent works in **states**. Suppose it starts at an initial state, $S0$. It then takes an action, say $a0$. This action transitions the state of the agent from $S0$ to $S1$, after which the agent takes another action, $a1$, and the cycle goes on.

Occasionally, the agent receives **rewards** based on its state. The sequence of states and actions that the agent traverses is also known as a **trajectory**. Let's say the agent received a reward at state $S2$. In that case, the trajectory that resulted in this reward would be $S0$, $a0$, $S1$, $a1$, $S2$.

Note
The rewards could either be positive or negative.

Based on the rewards, the agent learns to adjust its behavior so that it takes actions in a way that maximizes the long-term rewards. This is the essence of RL. The agent learns a strategy regarding how to act optimally (that is, to maximize the reward) based on the given state and reward.

This learned strategy, which is basically actions expressed as a function of states and rewards, is called the **policy** of the agent. The ultimate goal of RL is to compute a policy that enables the agent to always receive the maximum reward from the given

situation the agent is placed in.

Video games are one of the best examples to demonstrate RL. Let's use the video game Pong as an example, which is a virtual version of table tennis. The following is a snapshot of this game:

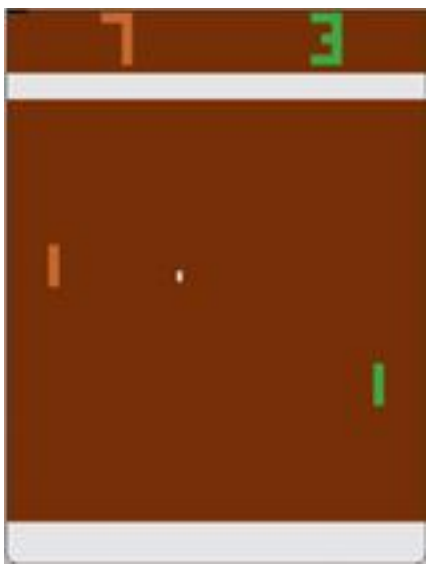

Figure 11. 2 – Pong video game

Consider that the player to the right is the agent, which is represented by a short vertical line. Notice that there is a well-defined environment here. The environment consists of the playing area, which is denoted by the brown pixels. The environment also consists of a ball, which is denoted by a white pixel. As well as this, the environment consists of the boundaries of the playing area, denoted by the gray stripes and edges that the ball may bounce off. Finally, and most importantly, the environment includes an opponent, which looks like the agent but is placed on the left-hand side, opposite the agent.

Usually, in an RL setting, the agent at any given state has a finite set of possible actions, referred to as a discrete action space (as opposed to a continuous action space). In this example, the agent has two possible actions at all states – move up or move down, but with two exceptions. First, it can only move down when it is at the top-most position (state), and second, it can only move up when it is at the bottom-most position (state).

The concept of reward in this case can be directly mapped to what happens in an actual table tennis game. If you miss the ball, your opponent gains a point. Whoever scores 21 points first wins the game and receives a positive reward. Losing a game means negative rewards. Scoring a point or losing a point also results in smaller intermediate positive and negative rewards, respectively. A sequence of play starting from score 0-0 and leading to either of the players scoring 21 points is called an **episode**.

Training our agent for a Pong game using RL is equivalent to training someone to play table tennis from scratch. Training results in a policy that the agent follows while playing the game. In any given situation – which includes the position of the ball, the

position of the opponent, the scoreboard, as well as the previous reward – a successfully trained agent moves up or down to maximize its chances of winning the game.

So far, we have discussed the basic concepts behind RL by providing an example. In doing so, we have repeatedly mentioned terms such as strategy, policy, and learning. But how does the agent actually learn the policy? The answer is through an RL model, which works based on a pre-defined algorithm. Next, we will explore the different kinds of RL algorithms.

Types of reinforcement learning algorithms

In this section, we will look at the types of RL algorithms, as per the literature. We will then explore some of the subtypes within these types. Broadly speaking, RL algorithms can be categorized as either of the following:

- **Model-based**
- **Model-free**

Let's look at these one by one.

Model-based

As the name suggests, in model-based algorithms, the agent knows about the model of the environment. The model here refers to the mathematical formulation of a function that can be used to estimate rewards and how the states transition within the environment. Because the agent has some idea about the environment, it helps reduce the sample space to choose the next action from. This helps with the efficiency of the learning process.

However, in reality, a modeled environment is not directly available most of the time. If we, nonetheless, want to use the model-based approach, we need to have the agent learn the environment model with its own experience. In such cases, the agent is highly likely to learn a biased representation of the model and perform poorly in the real environment. For this reason, model-based approaches are less frequently used for implementing RL systems. We will not be discussing models based on this approach in detail in this book, but here are some examples:

- **Model-Based DRL with Model-Free Fine-Tuning (MBMF).**
- **Model-Based Value Estimation (MBVE)** for efficient Model-Free RL.

Imagination-Augmented Agents (I2A) for DRL.

AlphaZero, the famous AI bot that defeated Chess and Go champions.

Now, let's look at the other set of RL algorithms that work with a different philosophy.

Model-free

The model-free approach works without any model of the environment and is currently more popularly used for RL research and development. There are primarily two ways of training the agent in a model-free RL setting:

Policy optimization

Q-learning

Policy optimization

In this method, we formulate the policy in the form of a function of an action, given the current state, as demonstrated in the following equation:

$$Policy = F_{\beta}(a \mid S)$$

– Equation 11.1

Here, β represents the internal parameters of this function, which is updated to optimize the policy function via gradient ascent. The objective function is defined using the policy function and the rewards. An approximation of the objective function may also be used in some cases for the optimization process. Furthermore, in some cases, an approximation of the policy function could be used instead of the actual policy function for the optimization process.

Usually, the optimizations that are performed under this approach are **on-policy**, which means that the parameters are updated based on the data gathered using the latest policy version. Some examples of policy optimization-based RL algorithms are as follows:

Policy gradient: This is the most basic policy optimization method where we directly optimize the policy function using gradient ascent. The policy function outputs the probabilities of different actions to be taken next, at each time step.

Actor-critic: Because of the on-policy nature of optimization under the policy gradient algorithm, every iteration of the algorithm needs the policy to be

updated. This takes a lot of time. The actor-critic method introduces the use of a value function, as well as a policy function. The actor models the policy function and the critic models the value function.

By using a critic, the policy update process becomes faster. We will discuss the value function in more detail in the next section. However, we will not go into the mathematical details of the actor-critic method in this book.

Trust region policy optimization (TRPO): Like the policy gradient method, TRPO consists of an on-policy optimization approach. In the policy-gradient approach, we use the gradient for updating the policy function parameters, β. Since the gradient is a first-order derivative, it can be noisy for sharp curvatures in the function. This may lead us to making large policy changes that may destabilize the learning trajectory of the agent.

To avoid that, TRPO proposes a trust region. It defines an upper limit on how much the policy may change in a given update step. This ensures the stability of the optimization process.

Proximal policy optimization (PPO): Similar to TRPO, PPO aims to stabilize the optimization process. During gradient ascent, an update is performed per data sample in the policy gradient approach. PPO, however, uses a surrogate objective function, which facilitates updates over batches of data samples. This results in estimating gradients more conservatively, thereby improving the chances of the gradient ascent algorithm converging.

Policy optimization functions directly work on optimizing the policy and hence are extremely intuitive algorithms. However, due to the on-policy nature of most of these algorithms, data needs to be resampled at each step after the policy is updated. This can be a limiting factor in solving RL problems. Next, we will discuss the other kind of model-free algorithm that is more sample-efficient, known as Q-learning.

Q-learning

Contrary to policy optimization algorithms, **Q-learning** relies on a value function instead of a policy function. From here on, this chapter will focus on Q-learning. We will explore the fundamentals of Q-learning in detail in the next section.

Discussing Q-learning

The key difference between policy optimization and Q-learning is the fact that in the latter, we are not directly optimizing the policy. Instead, we optimize a value function. What is a **value function**? We have already learned that RL is all about an agent learning to gain the maximum overall rewards while traversing a trajectory of states and actions. A value function is a function of a given state the agent is currently at, and this function outputs the expected sum of rewards the agent will receive by the end of the current episode.

In Q-learning, we optimize a specific type of value function, known as the **action-value function**, which depends on both the current state and the action. At a given state, *S*, the action-value function determines the long-term rewards (rewards until the end of the episode) the agent will receive for taking action *a*. This function is usually expressed as *Q(S, a)*, and hence is also called the Q-function. The action-value is also referred to as the **Q-value**.

The Q-values for every (state, action) pair can be stored in a table where the two dimensions are state and action. For example, if there are four possible states, *S*1, *S*2, *S*3, and *S*4, and two possible actions, *a*1 and *a*2, then the eight Q-values will be stored in a 4x2 table. The goal of Q-learning, therefore, is to create this table of Q-values. Once the table is available, the agent can look up the Q-values for all possible actions from the given state and take the action with the maximum Q-value. However, the question is, where do we get the Q-values from? The answer lies in the **Bellman equation**, which is mathematically expressed as follows:

$$Q\left(S_t, a_t\right) = R + \gamma * Q\left(S_{t+1}, a_{t+1}\right)$$

– Equation 11.2

The Bellman equation is a recursive way of calculating Q-values. *R* in this equation is the reward received by taking action *a*t at state *S*t, while γ (gamma) is the **discount factor**, which is a scalar value between *0* and *1*. Basically, this equation states that the Q-value for the current state, *S*t, and action, *a*t, is equal to the reward, *R*, received by taking action *a*t at state *S*t, plus the Q-value resulting from the most optimal action, *a*t*+1*, taken from the next state, *S*t*+1*, multiplied by a discount factor. The discount factor defines how much weightage is to be given to the immediate reward versus the long-term future rewards.

Now that we have defined most of the underlying concepts of Q-learning, let's walk through an example to demonstrate how Q-learning exactly works. The following diagram shows an environment that consists of five possible states:

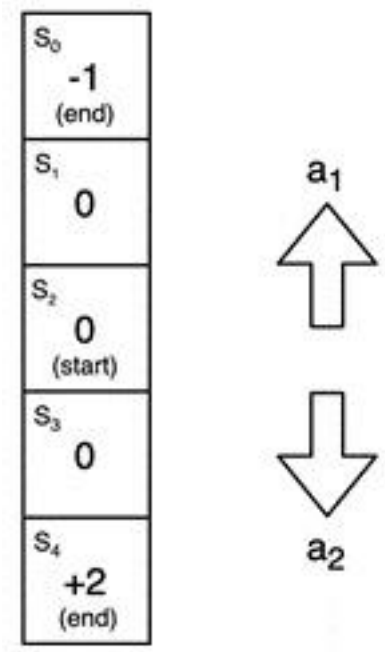

Figure 11. 3 – Q-learning example environment

There are two different possible actions – moving up (**a1**) or down (**a2**). There are different rewards at different states ranging from **+2** at state **S4** to **-1** at state **S0**. Every episode in this environment starts from state **S2** and ends at either **S0** or **S4**. Because there are five states and two possible actions, the Q-values can be stored in a 5x2 table. The following code snippet shows how rewards and Q-values can be written in Python:

```
rwrds = [-1, 0, 0, 0, 2]
Qvals = [[0.0, 0.0],
         [0.0, 0.0],
         [0.0, 0.0],
         [0.0, 0.0],
         [0.0, 0.0]]
```

We initialize all the Q-values to zero. Also, because there are two specific end states, we need to specify those in the form of a list, as shown here:

```
end_states = [1, 0, 0, 0, 1]
```

This basically indicates that states **S0** and **S4** are end states. There is one final piece we need to look at before we can run the complete Q-learning loop. At each step of Q-learning, the agent has two options with regards to taking the next action:

Take the action that has the highest Q-value.

Randomly choose the next action.

Why would the agent choose an action randomly?

Remember that in *Chapter 7 , Music and Text Generation with PyTorch*, in the *Text generation* section, we discussed how greedy search or beam search results in repetitive results, and hence introducing randomness helps in producing better results. Similarly , if the agent always chooses the next action based on Q-values, then it might get stuck choosing an action repeatedly that gives an immediate high reward in the short term. Hence, taking actions randomly once in a while will help the agent get out of such sub-optimal conditions.

Now that we've established that the agent has two possible ways of taking an action at each step, we need to decide which way the agent goes. This is where the **epsilon-greedy-action** mechanism comes into play. The following diagram shows how it works:

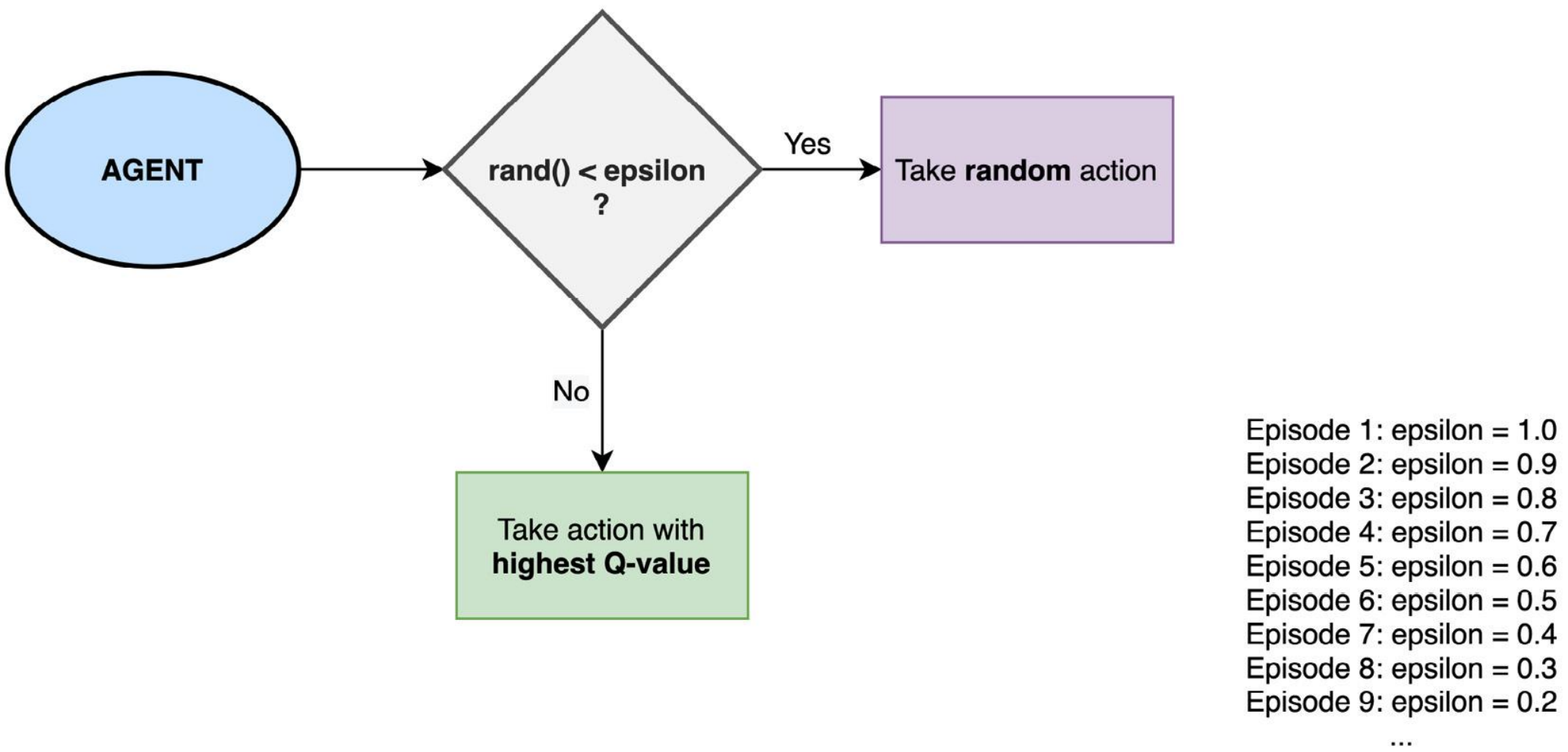

Figure 11. 4 – Epsilon-greedy-action mechanism

Under this mechanism, at each episode, an epsilon value is pre-decided, which is a scalar value between `0` and `1`. In a given episode, for taking each next action, the agent generates a random number between `0` to `1`. If the generated number is less than the pre-defined epsilon value, the agent chooses the next action randomly from the available set of next actions. Otherwise, the Q-values for each of the next

possible actions are retrieved from the Q-value table, and the action with the highest Q-value is chosen. The Python code for the epsilon-greedy-action mechanism is as follows:

Copy Explain

```
def eps_greedy_action_mechanism(eps, S):
  rnd = np.random.uniform()
  if rnd < eps:
    return np.random.randint(0, 2)
  else:
    return np.argmax(Qvals[S])
```

Typically, we start with an epsilon value of 1 at the first episode and then linearly decrease it as the episodes progress. The idea here is that we want the agent to explore different options initially. However, as the learning process progresses, the agent is less susceptible to getting stuck collecting short-term rewards and hence it can better exploit the Q-values table.

We are now in a position to write the Python code for the main Q-learning loop, which will look as follows:

Copy Explain

```
n_epsds = 100
eps = 1
gamma = 0.9
for e in range(n_epsds):
  S_initial = 2 # start with state S2
  S = S_initial
  while not end_states[S]:
    a = eps_greedy_action_mechanism(eps, S)
    R, S_next = take_action(S, a)
    if end_states[S_next]:
      Qvals[S][a] = R
    else:
      Qvals[S][a] = R + gamma * max(Qvals[S_next])
    S = S_next
  eps = eps - 1/n_epsds
```

First, we define that the agent shall be trained for 100 episodes. We begin with an epsilon value of 1 and we define the discounting factor (gamma) as 0.9. Next, we run the Q-learning loop, which loops over the number of episodes. In each iteration of this loop, we run through an entire episode. Within the episode, we first initialize the state of the agent to S2.

Thereon, we run another internal loop, which only breaks if the agent reaches an end state. Within this internal loop, we decide on the next action for the agent using the epsilon-greedy-action mechanism. The agent then takes the action, which transitions the agent to a new state and may possibly yield a reward. The implementation for the `take_action` function is as follows:

Copy Explain

```
def take_action(S, a):
  if a == 0: # move up
    S_next = S - 1
  else:
    S_next = S + 1
  return rwrds[S_next], S_next
```

Once we obtain the reward and the next state, we update the Q-value for the current state-action pair using equation 11.2 . The next state now becomes the current state and the process repeats. At the end of each episode, the epsilon value is reduced linearly. Once the entire Q-learning loop is over, we obtain a Q-values table. This table is essentially all that the agent needs to operate in this environment in order to gain the maximum long-term rewards.

Ideally, a well-trained agent for this example would always move downward to receive the maximum reward of *+2* at *S*4, and would avoid going toward *S*0, which contains a negative reward of *-1*.

This completes our discussion on Q-learning. The preceding code should help you get started with Q-learning in simple environments such as the one provided here. For more complex and realistic environments, such as video games, this approach will not work. Why?

We have noticed that the essence of Q-learning lies in creating the Q-values table. In our example, we only had 5 states and 2 actions, and therefore the table was of size 10, which is manageable. But in video games such as Pong, there are far too many possible states. This explodes the Q-values table's size, which makes our Q-learning algorithm extremely memory intensive and impractical to run.

Thankfully, there is a solution where we can still use the concept of Q-learning without having our machines run out of memory. This solution combines the worlds of Q-learning and deep neural networks and provides the extremely popular RL algorithm known as **DQN**. In the next section, we will discuss the basics of DQN and some of its novel characteristics.

Understanding deep Q-learning

Instead of creating a Q-values table, **DQN** uses a **deep neural network** (**DNN**) that outputs a Q-value for a given state-action pair. DQN is used with complex environments such as video games, where there are far too many states for them to be managed in a Q-values table. The current image frame of the video game is used to represent the current state and is fed as input to the underlying DNN model, together with the current action.

The DNN outputs a scalar Q-value for each such input. In practice, instead of just passing the current image frame, *N* number of neighboring image frames in a given time window are passed as input to the model.

We are using a DNN to solve an RL problem. This has an inherent concern. While working with DNNs, we have always worked with **independent and identically distributed** (**iid**) data samples. However, in RL, every current output impacts the next input. For example, in the case of Q-learning, the Bellman equation itself suggests that the Q-value is dependent on another Q-value; that is, the Q-value of the next state-action pair impacts the Q-value of the current-state pair.

This implies that we are working with a constantly moving target and there is a high correlation between the target and the input. DQN addresses these issues with two novel features:

Using two separate DNNs

Experience replay buffer

Let's look at these in more detail.

Using two separate DNNs

Let's rewrite the Bellman equation for DQNs:

$$Q\left(S_t, a_t, \theta\right) = R + \gamma * Q\left(S_{t+1}, a_{t+1}, \theta\right)$$

– Equation 11.3

This equation is mostly the same as for Q-learning except for the introduction of a new term,

θ

(theta).

θ

represents the weights of the DNN that the DQN model uses to get Q-values. But something is odd with this equation. Notice that

θ

is placed on both the left hand-side and the right-hand side of the equation. This means that at every step, we are using the same neural network for getting the Q-values of the current state-action, pair as well as the next state-action pair. This means that we are chasing a non-stationary target because every step,

θ

, will be updated, which will change both the left-hand side as well as the right-hand side of the equation for the next step, causing instability in the learning process. This can be more clearly seen by looking at the loss function, which the DNN will be trying to minimize using gradient descent. The loss function is as follows:

$$L = E[(R + \gamma * Q(S_{t+1}, a_{t+1}, \theta) - Q(S_t, a_t, \theta))^2]$$

– Equation 11.4

Keeping *R* (reward) aside for a moment, having the exact same network producing Q-values for current and next state-action pairs will lead to volatility in the loss function as both terms will be constantly changing. To address this issue, DQN uses two separate networks – a main DNN and a target DNN. Both DNNs have the exact same architecture.

The main DNN is used for computing the Q-values of the current state-action pair, while the target DNN is used for computing the Q-values of the next (or target) state-action pair. However, although the weights of the main DNN are updated at every learning step, the weights of the target DNN are frozen. After every *K* gradient descent iterations, the weights of the main network are copied to the target network. This mechanism keeps the training procedure relatively stable. The weights-copying mechanism ensures accurate predictions from the target network.

Experience replay buffer

Because the DNN expects iid data as input, we simply cache the last *X* number of steps (frames of the video game) into a buffer memory and then randomly sample batches of data from the buffer. These batches are then fed as inputs to the DNN. Because the batches consist of randomly sampled data, the distribution looks similar to that of iid data samples. This helps stabilize the DNN training process.

Note
Without the buffer trick, the DNN would receive correlated data, which would result in poor optimization results.

These two tricks have proven significant in contributing to the success of DQNs. Now that we have a basic understanding of how DQN models work and their novel characteristics, let's move on to the final section of this chapter, where we will implement our own DQN model. Using PyTorch, we will build a CNN-based DQN model that will learn to play the Atari video game known as Pong and potentially learn to win the game against the computer opponent.

Building a DQN model in PyTorch

We discussed the theory behind DQNs in the previous section. In this section, we will take a hands-on approach. Using PyTorch, we will build a CNN-based DQN model that will train an agent to play the video game known as Pong. The goal of this exercise is to demonstrate how to develop DRL applications using PyTorch. Let's get straight into the exercise.

Initializing the main and target CNN models

In this exercise, we will only show the important parts of the code for demonstration purposes. In order to access the full code, visit our github repository [11.1] . Follow these steps:

1. First, we need to import the necessary libraries:

Copy Explain

```
# general imports
import cv2
import math
import numpy as np
import random
# reinforcement learning related imports
import re
import atari_py as ap
from collections import deque
from gym import make, ObservationWrapper, Wrapper
from gym.spaces import Box
# pytorch imports
import torch
import torch.nn as nn
from torch import save
from torch.optim import Adam
```

In this exercise, besides the usual Python- and PyTorch-related imports, we are also using a Python library called `gym`. It is a Python library produced by OpenAI [11.2] that provides a set of tools for building DRL applications. Essentially, importing `gym` does away with the need of writing all the scaffolding code for the internals of an RL system. It also consists of built-in environments, including one for the video game Pong, which we will use in this exercise.

1. After importing the libraries, we must define the CNN architecture for the DQN model. This CNN model essentially takes in the current state input and outputs the probability distribution over all possible actions. The action with the highest probability gets chosen as the next action by the agent. Instead of using a regression model to predict the Q-values for each state-action pair, we cleverly turn this into a classification problem.

The Q-value regression model will have to be run separately for all possible actions, and we will choose the action with the highest predicted Q-value. But using this classification model combines the task of calculating Q-values and predicting the best next action into one:

```
class ConvDQN(nn.Module):
    def __init__(self, ip_sz, tot_num_acts):
        super(ConvDQN, self).__init__()
        self._ip_sz = ip_sz
        self._tot_num_acts = tot_num_acts
        self.cnv1 = nn.Conv2d(ip_sz[0], 32, kernel_size=8, stride=4)
        self.rl = nn.ReLU()
        self.cnv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.cnv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.fc1 = nn.Linear(self.feat_sz, 512)
        self.fc2 = nn.Linear(512, tot_num_acts)
```

As we can see, the model consists of three convolutional layers – `cnv1`, `cnv2`, and `cnv3` – with ReLU activations in-between them, followed by two fully connected layers. Now, let's look at what a forward pass through this model entails:

```
    def forward(self, x):
        op = self.cnv1(x)
        op = self.rl(op)
        op = self.cnv2(op)
        op = self.rl(op)
        op = self.cnv3(op)
        op = self.rl(op).view(x.size()[0], -1)
        op = self.fc1(op)
        op = self.rl(op)
        op = self.fc2(op)
        return op
```

The `forward` method simply demonstrates a forward pass by the model, where the input is passed through the convolutional layers, flattened, and finally fed to the fully connected layers. Finally, let's look at the other model methods:

```
    @property
    def feat_sz(self):
        x = torch.zeros(1, *self._ip_sz)
        x = self.cnv1(x)
        x = self.rl(x)
        x = self.cnv2(x)
        x = self.rl(x)
        x = self.cnv3(x)
        x = self.rl(x)
        return x.view(1, -1).size(1)
    def perf_action(self, stt, eps, dvc):
        if random.random() > eps:
            stt=torch.from_numpy(np.float32(stt)).unsqueeze(0).to(dvc)
            q_val = self.forward(stt)
            act = q_val.max(1)[1].item()
        else:
            act = random.randrange(self._tot_num_acts)
        return act
```

In the preceding code snippet, the `feat_size` method is simply meant to calculate the size of the feature vector after flattening the last convolutional layer output. Finally, the `perf_action` method is the same as the `take_action` method we discussed previously in the *Discussing Q-learning* section.

1. In this step, we define a function that instantiates the main neural network and the target neural network:

```
def models_init(env, dvc):
    mdl = ConvDQN(env.observation_space.shape, env.action_space.n).to(dvc)
    tgt_mdl = ConvDQN(env.observation_space.shape, env.action_space.n).to(dvc)
    return mdl, tgt_mdl
```

These two models are instances of the same class and hence share the same architecture. However, they are two separate instances and hence will evolve differently with different sets of weights.

Defining the experience replay buffer

As we discussed in the *Understanding deep Q-learning* section, the experience replay buffer is a significant feature of DQNs. With the help of this buffer, we can store several thousand transitions (frames) of a game and then randomly sample those video frames to train the CNN model. The following is the code for defining the replay buffer:

```
class RepBfr:
    def __init__(self, cap_max):
        self._bfr = deque(maxlen=cap_max)
    def push(self, st, act, rwd, nxt_st, fin):
        self._bfr.append((st, act, rwd, nxt_st, fin))
    def smpl(self, bch_sz):
        idxs = np.random.choice(len(self._bfr), bch_sz, False)
        bch = zip(*[self._bfr[i] for i in idxs])
        st, act, rwd, nxt_st, fin = bch
        return (np.array(st), np.array(act), np.array(rwd,
dtype=np.float32),np.array(nxt_st), np.array(fin, dtype=np.uint8))
    def __len__(self):
        return len(self._bfr)
```

Here, `cap_max` is the defined buffer size; that is, the number of video game state transitions that shall be stored in the buffer. The `smpl` method is used during the CNN training loop to sample the stored transitions and generate batches of training data.

Setting up the environment

So far, we have mostly focused on the neural network side of DQNs. In this section, we will focus on building one of the foundational aspects in an RL problem – the environment. Follow these steps:

1. First, we must define some video game environment initialization-related functions:

```
def gym_to_atari_format(gym_env):
    ...
def check_atari_env(env):
    ...
```

Using the `gym` library, we have access to a pre-built Pong video game environment. But here, we will augment the environment in a series of steps, which will include downsampling the video game image frames, pushing image frames to the experience replay buffer, converting images into PyTorch tensors, and so on.

1. The following are the defined classes that implement each of the environment control steps:

```
class CCtrl(Wrapper):
    ...
class FrmDwSmpl(ObservationWrapper):
    ...
class MaxNSkpEnv(Wrapper):
    ...
class FrRstEnv(Wrapper):
    ...
class FrmBfr(ObservationWrapper):
    ...
class Img2Trch(ObservationWrapper):
    ...
class NormFlts(ObservationWrapper):
    ...
```

These classes will now be used for initializing and augmenting the video game environment.

1. Once the environment-related classes have been defined, we must define a final method that takes in the raw Pong video game environment as input and augments the environment, as follows:

```
def wrap_env(env_ip):
    env = make(env_ip)
    is_atari = check_atari_env(env_ip)
    env = CCtrl(env, is_atari)
    env = MaxNSkpEnv(env, is_atari)
    try:
        env_acts = env.unwrapped.get_action_meanings()
        if "FIRE" in env_acts:
            env = FrRstEnv(env)
    except AttributeError:
        pass
    env = FrmDwSmpl(env)
    env = Img2Trch(env)
    env = FrmBfr(env, 4)
    env = NormFlts(env)
    return env
```

Some of the code in this step have been omitted as our focus is on the PyTorch aspect of this exercise. Please refer to this book's GitHub repository [11.3] for the full code.

Defining the CNN optimization function

In this section, we will define the loss function for training our DRL model, as well as define what needs to be done at the end of each model training iteration. Follow these steps:

1. We initialized our main and target CNN models in *step 2* of the *Initializing the main and target CNN models* section. Now that we have defined the model architecture, we shall define the `loss` function, which the model will be trained to minimize:

```
def calc_temp_diff_loss(mdl, tgt_mdl, bch, gm, dvc):
    st, act, rwd, nxt_st, fin = bch        st =
torch.from_numpy(np.float32(st)).to(dvc)
    nxt_st =      torch.from_numpy(np.float32(nxt_st)).to(dvc)
    act = torch.from_numpy(act).to(dvc)
    rwd = torch.from_numpy(rwd).to(dvc)
    fin = torch.from_numpy(fin).to(dvc)     q_vals = mdl(st)
    nxt_q_vals = tgt_mdl(nxt_st)        q_val = q_vals.gather(1,
act.unsqueeze(-1)).squeeze(-1)
    nxt_q_val = nxt_q_vals.max(1)[0]
    exp_q_val = rwd + gm * nxt_q_val * (1 - fin)        loss = (q_val -
exp_q_val.data.to(dvc)).pow(2).   mean()
    loss.backward()
```

The loss function defined here is derived from equation 11.4. . This loss is known as the **time/temporal difference loss** and is one of the foundational concepts of DQNs.

1. Now that the neural network architecture and loss function are in place, we shall define the model `updation` function, which is called at every iteration of neural network training:

```
def upd_grph(mdl, tgt_mdl, opt, rpl_bfr, dvc, log):
    if len(rpl_bfr) > INIT_LEARN:
        if not log.idx % TGT_UPD_FRQ:
            tgt_mdl.load_state_dict(mdl.state_dict())
        opt.zero_grad()
        bch = rpl_bfr.smpl(B_S)
        calc_temp_diff_loss(mdl, tgt_mdl, bch, G, dvc)
        opt.step()
```

This function samples a batch of data from the experience replay buffer, computes the time difference loss on this batch of data, and also copies the weights of the main neural network to the target neural network once every `TGT_UPD_FRQ` iterations. `TGT_UPD_FRQ` will be assigned a value later.

Managing and running episodes

Now, let's learn how to define the epsilon value:

1. First, we will define a function that will update the epsilon value after each episode:

```
def upd_eps(epd):
    last_eps = EPS_FINL
    first_eps = EPS_STRT
    eps_decay = EPS_DECAY
    eps = last_eps + (first_eps - last_eps) * math.exp(-1 * ((epd + 1) / eps_decay))
    return eps
```

This function is the same as the epsilon update step in our Q-learning loop, as discussed in the *Discussing Q-learning* section. The goal of this function is to linearly reduce the epsilon value per episode.

1. The next function is to define what happens at the end of an episode. If the overall reward that's scored in the current episode is the best we've achieved so far, we save the CNN model weights and print the reward value:

```
def fin_epsd(mdl, env, log, epd_rwd, epd, eps):
    bst_so_far = log.upd_rwds(epd_rwd)
    if bst_so_far:
        print(f"checkpointing current model weights. highest running_average_reward of\
{round(log.bst_avg, 3)} achieved!")
        save(mdl.state_dict(), f"{env}.dat")
    print(f"episode_num {epd}, curr_reward: {epd_rwd},       best_reward:
{log.bst_rwd},\running_avg_reward: {round(log.avg, 3)}, curr_epsilon: {round(eps, 4)}")
```

At the end of each episode, we also log the episode number, the reward at the end of the current episode, a running average of reward values across the past few episodes, and finally, the current epsilon value.

1. We have finally reached one of the most crucial function definitions of this exercise. Here, we must specify the DQN loop. This is where we define the steps that shall be executed in an episode:

```
def run_epsd(env, mdl, tgt_mdl, opt, rpl_bfr, dvc, log, epd):
    epd_rwd = 0.0
    st = env.reset()
    while True:
        eps = upd_eps(log.idx)
        act = mdl.perf_action(st, eps, dvc)
        env.render()
        nxt_st, rwd, fin, _ = env.step(act)
        rpl_bfr.push(st, act, rwd, nxt_st, fin)
        st = nxt_st
        epd_rwd += rwd
        log.upd_idx()
        upd_grph(mdl, tgt_mdl, opt, rpl_bfr, dvc, log)
        if fin:
            fin_epsd(mdl, ENV, log, epd_rwd, epd, eps)
            break
```

The rewards and states are reset at the beginning of the episode. Then, we run an endless loop that only breaks if the agent reaches one of the end states. Within this loop, in each iteration, the following steps are executed:

i) First, the epsilon value is modified as per the *linear depreciation scheme*.

ii) The next action is predicted by the main CNN model. This action is executed, resulting in the next state and a reward. This state transition is recorded in the experience replay buffer.

iii) The next state now becomes the current state and we calculate the time difference loss, which is used to update the main CNN model while keeping the target CNN model frozen.

iv) If the new current state is an end state, then we break the loop (that is, end the episode) and log the results for this episode.

1. We have mentioned logging results throughout the training process. In order to store the various metrics around rewards and model performance, we must define a training metadata class, which will consist of various metrics as attributes:

```
class TrMetadata:
    def __init__(self):
        self._avg = 0.0
        self._bst_rwd = -float("inf")
        self._bst_avg = -float("inf")
        self._rwds = []
        self._avg_rng = 100
        self._idx = 0
```

We will use these metrics to visualize model performance later in this exercise, once we've trained the model.

1. We store the model metric attributes in the previous step as private members and publicly expose their corresponding getter functions instead:

```
    @property
    def bst_rwd(self):
        ...
    @property
    def bst_avg(self):
        ...
    @property
    def avg(self):
        ...
    @property
    def idx(self):
        ...
    ...
```

The `idx` attribute is critical for deciding when to copy the weights from the main CNN to the target CNN, while the `avg` attribute is useful for computing the running average of rewards that have been received in the past few episodes.

Training the DQN model to learn Pong

Now, we have all the necessary ingredients to start training the DQN model. Let's get started:

1. The following is a training wrapper function that will do everything we need it to do:

Copy Explain

```
def train(env, mdl, tgt_mdl, opt, rpl_bfr, dvc):
    log = TrMetadata()
    for epd in range(N_EPDS):
        run_epsd(env, mdl, tgt_mdl, opt, rpl_bfr, dvc, log, epd)
```

Essentially, we initialize a logger and just run the DQN training system for a predefined number of episodes.

1. Before we actually run the training loop, we need to define the hyperparameter values, which are as follows:

i) The batch size for each iteration of gradient descent to tune the CNN model

ii) The environment, which in this case is the Pong video game

iii) The epsilon value for the first episode

iv) The epsilon value for the last episode

v) The rate of depreciation for the epsilon value

vi) Gamma; that is, the discounting factor

vii) The initial number of iterations that are reserved just for pushing data to the replay buffer

viii) The learning rate

ix) The size or capacity of the experience replay buffer

x) The total number of episodes to train the agent for

xi) The number of iterations after which we copy the weights from the main CNN to the target CNN

We can instantiate all of these hyperparameters in the following piece of code:

Copy Explain

```
B_S = 64
ENV = "Pong-v4"
EPS_STRT = 1.0
EPS_FINL = 0.005
EPS_DECAY = 100000
G = 0.99
INIT_LEARN = 10000
LR = 1e-4
MEM_CAP = 20000
N_EPDS = 2000
TGT_UPD_FRQ = 1000
```

These values are experimental, and I encourage you to try changing them and observe the impact they have on the results.

1. This is the last step of the exercise and is where we actually execute the DQN training routine, as follows:

i) First, we instantiate the game environment.

ii) Then, we define the device that the training will happen on – either CPU or GPU, based on availability.

iii) Next, we instantiate the main and target CNN models. We also define *Adam* as the optimizer for the CNN models.

iv) We then instantiate an experience replay buffer.

v) Finally, we begin training the main CNN model. Once the training routine finishes, we close the instantiated environment.

The code for this is as follows:

Copy Explain

```
env = wrap_env(ENV)
dvc = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
mdl, tgt_mdl = models_init(env, dvc)
opt = Adam(mdl.parameters(), lr=LR)
rpl_bfr = RepBfr(MEM_CAP)
train(env, mdl, tgt_mdl, opt, rpl_bfr, dvc)
env.close()
```

This should give us the following output:

```
episode_num 0, curr_reward: -20.0, best_reward: -20.0, running_avg_reward: -20.0, curr_epsilon: 0.9971
checkpointing current model weights. highest running_average_reward of -19.5 achieved!
episode_num 1, curr_reward: -19.0, best_reward: -19.0, running_avg_reward: -19.5, curr_epsilon: 0.9937
episode_num 2, curr_reward: -21.0, best_reward: -19.0, running_avg_reward: -20.0, curr_epsilon: 0.991
episode_num 3, curr_reward: -21.0, best_reward: -19.0, running_avg_reward: -20.25, curr_epsilon: 0.9881
episode_num 4, curr_reward: -19.0, best_reward: -19.0, running_avg_reward: -20.0, curr_epsilon: 0.9846
episode_num 5, curr_reward: -20.0, best_reward: -19.0, running_avg_reward: -20.0, curr_epsilon: 0.9811
                                        .
                                        .
                                        .
episode_num 500, curr_reward: -13.0, best_reward: -11.0, running_avg_reward: -16.52, curr_epsilon: 0.1053
episode_num 501, curr_reward: -20.0, best_reward: -11.0, running_avg_reward: -16.52, curr_epsilon: 0.1049
episode_num 502, curr_reward: -19.0, best_reward: -11.0, running_avg_reward: -16.59, curr_epsilon: 0.1041
episode_num 503, curr_reward: -12.0, best_reward: -11.0, running_avg_reward: -16.53, curr_epsilon: 0.1034
checkpointing current model weights. highest running_average_reward of -16.51 achieved!
episode_num 504, curr_reward: -13.0, best_reward: -11.0, running_avg_reward: -16.51, curr_epsilon: 0.1026
checkpointing current model weights. highest running_average_reward of -16.5 achieved!
episode_num 505, curr_reward: -18.0, best_reward: -11.0, running_avg_reward: -16.5, curr_epsilon: 0.1019
checkpointing current model weights. highest running_average_reward of -16.46 achieved!
                                        .
                                        .
                                        .
episode_num 1000, curr_reward: -4.0, best_reward: 13.0, running_avg_reward: -6.64, curr_epsilon: 0.0059
checkpointing current model weights. highest running_average_reward of -6.61 achieved!
episode_num 1001, curr_reward: -9.0, best_reward: 13.0, running_avg_reward: -6.61, curr_epsilon: 0.0059
episode_num 1002, curr_reward: -15.0, best_reward: 13.0, running_avg_reward: -6.72, curr_epsilon: 0.0059
episode_num 1003, curr_reward: -3.0, best_reward: 13.0, running_avg_reward: -6.66, curr_epsilon: 0.0059
episode_num 1004, curr_reward: -7.0, best_reward: 13.0, running_avg_reward: -6.72, curr_epsilon: 0.0059
episode_num 1005, curr_reward: -12.0, best_reward: 13.0, running_avg_reward: -6.69, curr_epsilon: 0.0059
                                        .
                                        .
                                        .
episode_num 1500, curr_reward: 11.0, best_reward: 17.0, running_avg_reward: -0.22, curr_epsilon: 0.005
checkpointing current model weights. highest running_average_reward of -0.05 achieved!
episode_num 1501, curr_reward: 7.0, best_reward: 17.0, running_avg_reward: -0.05, curr_epsilon: 0.005
checkpointing current model weights. highest running_average_reward of 0.01 achieved!
episode_num 1502, curr_reward: -1.0, best_reward: 17.0, running_avg_reward: 0.01, curr_epsilon: 0.005

checkpointing current model weights. highest running_average_reward of 0.11 achieved!
episode_num 1503, curr_reward: 3.0, best_reward: 17.0, running_avg_reward: 0.11, curr_epsilon: 0.005
checkpointing current model weights. highest running_average_reward of 0.2 achieved!
episode_num 1504, curr_reward: 2.0, best_reward: 17.0, running_avg_reward: 0.2, curr_epsilon: 0.005
episode_num 1505, curr_reward: -8.0, best_reward: 17.0, running_avg_reward: 0.19, curr_epsilon: 0.005
                                        .
                                        .
                                        .
episode_num 1000, curr_reward: -4.0, best_reward: 13.0, running_avg_reward: -6.64, curr_epsilon: 0.0059
checkpointing current model weights. highest running_average_reward of -6.61 achieved!
episode_num 1001, curr_reward: -9.0, best_reward: 13.0, running_avg_reward: -6.61, curr_epsilon: 0.0059
episode_num 1002, curr_reward: -15.0, best_reward: 13.0, running_avg_reward: -6.72, curr_epsilon: 0.0059
episode_num 1003, curr_reward: -3.0, best_reward: 13.0, running_avg_reward: -6.66, curr_epsilon: 0.0059
episode_num 1004, curr_reward: -7.0, best_reward: 13.0, running_avg_reward: -6.72, curr_epsilon: 0.0059
episode_num 1005, curr_reward: -12.0, best_reward: 13.0, running_avg_reward: -6.69, curr_epsilon: 0.0059
```

Figure 11. 5 – DQN training logs

Furthermore, the following graph shows the progression of the current rewards, best rewards, and average rewards, as well as the epsilon values against the progression of the episodes:

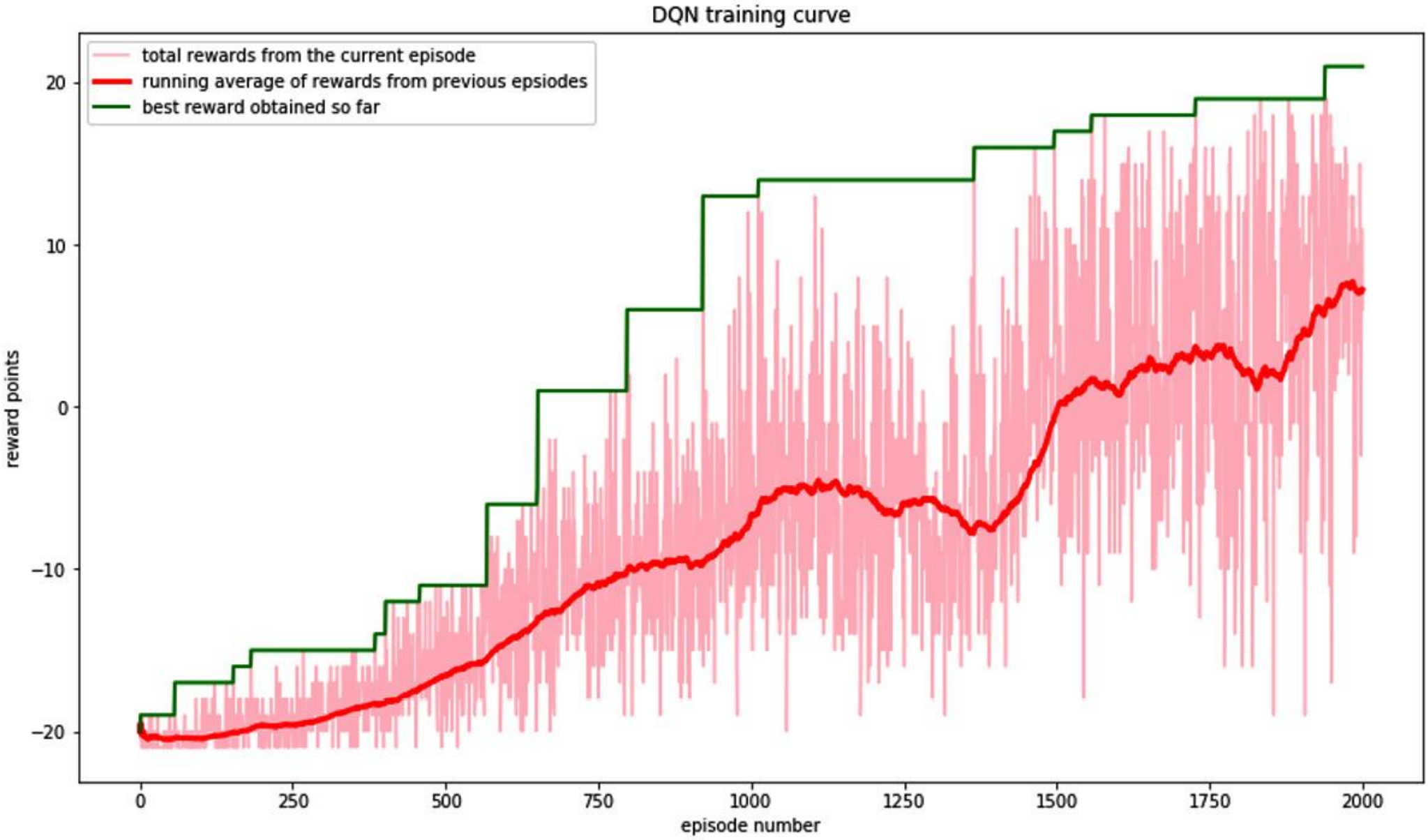

Figure 11. 6 – DQN training curves

The following graph shows how the epsilon value decreases over episodes during the training process:

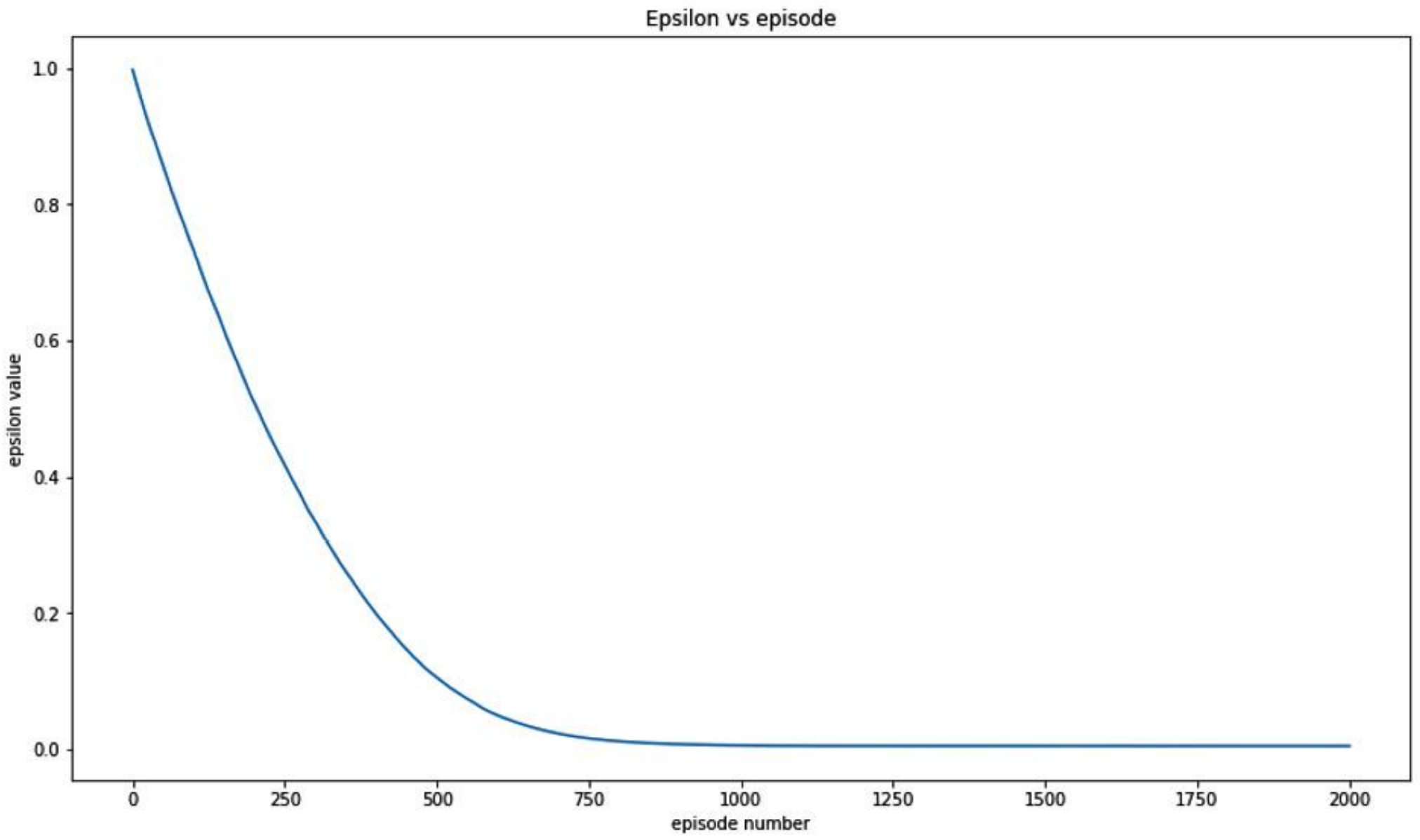

Figure 11. 7 – Epsilon variation over episodes

Notice that in *Figure 11. 6*, the running average value of rewards in an episode (red curve) starts at **-20**, which is the scenario where the agent scores **0** points in a game and the opponent scores all **20** points. As the episodes progress, the average rewards keep increasing and by episode number **1500**, it crosses the zero mark. This means that after **1500** episodes of training, the agent has leveled up against the opponent.

From here onward, the average rewards are positive, which indicates that the agent is winning against the opponent on average. We have only trained until **2000** episodes, which already results in the agent winning by a margin of over **7** average points against the opponent. I encourage you to train it for longer and see if the agent can absolutely crush the opponent by always scoring all the points and winning by a margin of **20** points.

This concludes our deep dive into the implementation of a DQN model. DQN has been vastly successful and popular in the field of RL and is definitely a great starting point for those interested in exploring the field further. PyTorch, together with the gym library, is a great resource that enables us to work in various RL environments and work with different kinds of DRL models.

In this chapter, we have only focused on DQNs, but the lessons we've learned can be transferred to working with other variants of Q-learning models and other DRL algorithms.

Summary

RL is one of the fundamental branches of machine learning and is currently one of the hottest, if not the hottest, areas of research and development. RL-based AI breakthroughs such as AlphaGo from Google's DeepMind have further increased enthusiasm and interest in the field. This chapter provided an overview of RL and DRL and walked us through a hands-on exercise of building a DQN model using PyTorch.

RL is a vast field and one chapter is not enough to cover everything. I encourage you to use the high-level discussions from this chapter to explore the details around those discussions. From the next chapter onward, we will focus on the practical aspects of working with PyTorch, such as model deployment, parallelized training, automated machine learning, and so on. In the next chapter, we will start by discussing how to effectively use PyTorch to put trained models into production systems.

Previous Chapter

Next Chapter