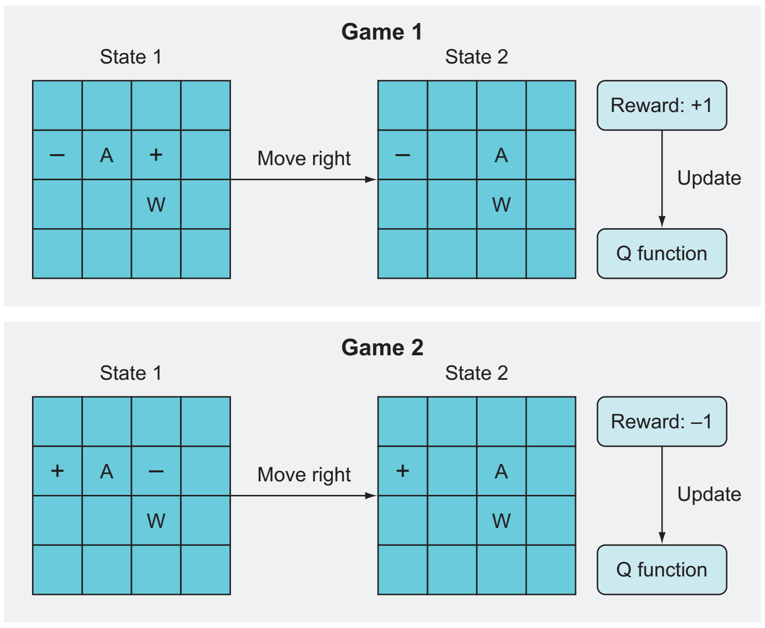

Figure 3.12 The idea of catastrophic forgetting is that when two game states are very similar and yet lead to very different outcomes, the Q function will get "confused" and won't be able to learn what to do. In this example, the catastrophic forgetting happens because the Q function learns from game 1 that moving right leads to a +1 reward, but in game 2, which looks very similar, it gets a reward of –1 after moving right. As a result, the algorithm forgets what it previously learned about game 1, resulting in essentially no significant learning at all.

3.3.2 Experience replay

Catastrophic forgetting is probably not something we have to worry about with the first variant of our game because the targets are always stationary, and indeed the model successfully learned how to play it. But with the random mode, it's something we need to consider, and that is why we need to implement something called *experience replay*. Experience replay basically gives us batch updating in an online learning scheme. It's not a big deal to implement

Here's how experience replay works (figure 3.13):

1. In state s, take action a, and observe the new state s_{t+1} and reward r_{t+1}.
2. Store this as a tuple (s, a, s_{t+1}, r_{t+1}) in a list.
3. Continue to store each experience in this list until you have filled the list to a specific length (this is up to you to define).
4. Once the experience replay memory is filled, randomly select a subset (again, you need to define the subset size).
5. Iterate through this subset and calculate value updates for each subset; store these in a target array (such as Y) and store the state, s, of each memory in X.

6 Use *X* and *Y* as a mini-batch for batch training. For subsequent epochs where the array is full, just overwrite old values in your experience replay memory array.

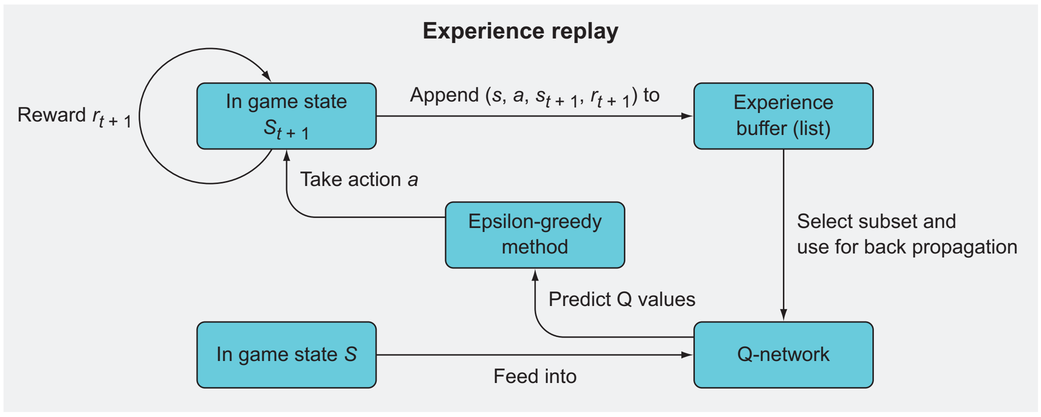

Figure 3.13 This is the general overview of experience replay, a method for mitigating a major problem with online training algorithms: catastrophic forgetting. The idea is to employ mini-batching by storing past experiences and then using a random subset of these experiences to update the Q-network, rather than using just the single most recent experience.

Thus, in addition to learning the action value for the action you just took, you're also going to use a random sample of past experiences to train on, to prevent catastrophic forgetting.

Listing 3.5 shows the same training algorithm from listing 3.4, except with experience replay added. Remember, this time we're training it on the harder variant of the game, where all the board pieces are randomly placed on the grid.

Listing 3.5 DQN with experience replay

```
from collections import deque
epochs = 5000
losses = []
mem_size = 1000                    # Sets the total size of the experience replay memory
batch_size = 200                   # Sets the mini-batch size
replay = deque(maxlen=mem_size)    # Creates the memory replay as a deque list
max_moves = 50                     # Sets the maximum number of moves before game is over
h = 0
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state1_ = game.board.render_np().reshape(1,64) +
     np.random.rand(1,64)/100.0
    state1 = torch.from_numpy(state1_).float()
    status = 1
    mov = 0
    while(status == 1):
        mov += 1
        qval = model(state1)       # Computes Q values from the input state in order to select an action
        qval_ = qval.data.numpy()
        if (random.random() < epsilon):
            action_ = np.random.randint(0,4)
        else:
            action_ = np.argmax(qval_)

        action = action_set[action_]
        game.makeMove(action)
        state2_ = game.board.render_np().reshape(1,64) +
     np.random.rand(1,64)/100.0
        state2 = torch.from_numpy(state2_).float()
        reward = game.reward()
        done = True if reward > 0 else False
        exp =  (state1, action_, reward, state2, done)
        replay.append(exp)
        state1 = state2

        if len(replay) > batch_size:
            minibatch = random.sample(replay, batch_size)
            state1_batch = torch.cat([s1 for (s1,a,r,s2,d) in minibatch])
            action_batch = torch.Tensor([a for (s1,a,r,s2,d) in minibatch])
            reward_batch = torch.Tensor([r for (s1,a,r,s2,d) in minibatch])
            state2_batch = torch.cat([s2 for (s1,a,r,s2,d) in minibatch])
            done_batch = torch.Tensor([d for (s1,a,r,s2,d) in minibatch])

            Q1 = model(state1_batch)
            with torch.no_grad():
                Q2 = model(state2_batch)

            Y = reward_batch + gamma * ((1 - done_batch) *
     torch.max(Q2,dim=1)[0])
            X = \
            Q1.gather(dim=1,index=action_batch.long().unsqueeze(dim=1)).squeeze()
            loss = loss_fn(X, Y.detach())
            optimizer.zero_grad()
            loss.backward()
            losses.append(loss.item())
            optimizer.step()

        if reward != -1 or mov > max_moves:
            status = 0
            mov = 0
losses = np.array(losses)
```

Selects an action using the epsilon-greedy strategy

Creates an experience of state, reward, action, and the next state as a tuple

Adds the experience to the experience replay list

If the replay list is at least as long as the mini-batch size, begins the mini-batch training

Randomly samples a subset of the replay list

Separates out the components of each experience into separate mini-batch tensors

Recomputes Q values for the mini-batch of states to get gradients

Computes Q values for the mini-batch of next states, but doesn't compute gradients

Computes the target Q values we want the DQN to learn

If the game is over, resets status and mov number

In order to store the agent's experiences, we used a data structure called a *deque* in Python's built-in collections library. It's basically a list that you can set a maximum size on, so that if you try to append to the list and it is already full, it will remove the first item in the list and add the new item to the end of the list. This means new experiences replace the oldest experiences. The experiences themselves are tuples of `(state1, reward, action, state2, done)` that we append to the `replay` deque.

The major difference with experience replay training is that we train with mini-batches of data when our replay list is full. We randomly select a subset of experiences from the replay, and we separate out the individual experience components into `state1_batch`, `reward_batch`, `action_batch`, and `state2_batch`. For example, `state1_batch` is of dimensions `batch_size` × 64, or 100 × 64 in this case. And `reward_batch` is just a 100-length vector of integers. We follow the same training formula as we did earlier with fully online training, but now we're dealing with mini-batches. We use the tensor `gather` method to subset the `Q1` tensor (a 100 × 4 tensor) by the action indices so that we only select the Q values associated with actions that were actually chosen, resulting in a 100-length vector.

Notice that the target Q value, `Y = reward_batch + gamma * ((1 - done_batch) * torch.max(Q2,dim=1)[0])`, uses `done_batch` to set the right side to 0 if the game is done. Remember, if the game is over after taking an action, which we call a *terminal state*, there is no next state to take the maximum Q value on, so the target just becomes the reward, r_{t+1}. The `done` variable is a Boolean, but we can do arithmetic on it as if it were a 0 or 1 integer, so we just take `1 - done` so that if `done = True`, `1 - done = 0`, and it sets the right-side term to 0.

We trained for 5,000 epochs this time, since it's a more difficult game, but otherwise the Q-network model is the same as before. When we test the algorithm, it seems to play most of the games correctly. We wrote an additional testing script to see what percentage of games it wins out of 1,000 plays.

Listing 3.6 Testing the performance with experience replay

```
max_games = 1000
wins = 0
for i in range(max_games):
    win = test_model(model, mode='random', display=False)
    if win:
        wins += 1
win_perc = float(wins) / float(max_games)
print("Games played: {0}, # of wins: {1}".format(max_games,wins))
print("Win percentage: {}".format(win_perc))
```

When we run listing 3.6 on our trained model (trained for 5,000 epochs), we get about 90% accuracy. Your accuracy may be slightly better or worse. This certainly suggests it has learned *something* about how to play the game, but it's not exactly what we would expect if the algorithm really knew what it was doing (although you could probably improve the accuracy with a much longer training time). Once you actually know how to play, you should be able to win every single game.

There's a small caveat that some of the initialized games may actually be impossible to win, so the win percentage may never reach 100%; there is no logic preventing the goal from being in the corner, stuck behind a wall and pit, making the game unwinnable. The Gridworld game engine does prevent most of the impossible board configurations, but a small number can still get through. Not only does this mean we can't win every game, but it also means the learning will be mildly corrupted, since it will attempt to follow a strategy that normally would work but fails for an unwinnable game. We wanted to keep the game logic simple to focus on illustrating the concepts so we did not program in the sophisticated logic needed to ensure 100% winnable games.

There's also another reason we're being held back from getting into the 95% + accuracy territory. Let's look at our loss plot, shown in figure 3.14 showing our running average loss (yours may vary significantly).

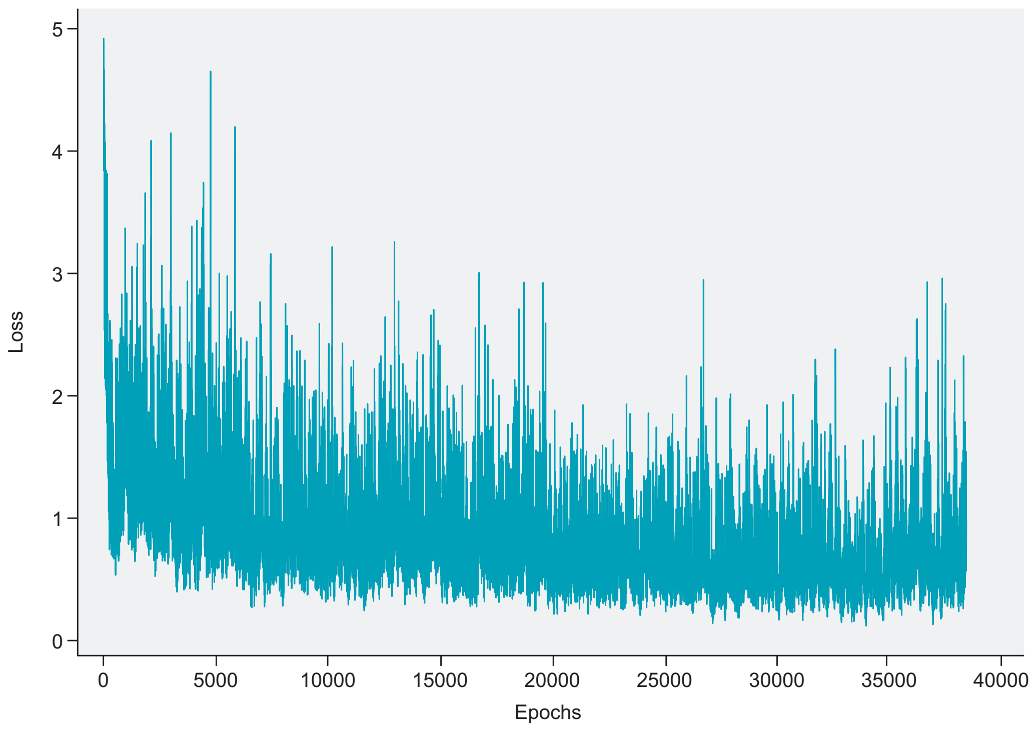

Figure 3.14 The DQN loss plot after implementing experience replay, which shows a clearly down-trending loss, but it's still very noisy.

In the loss in figure 3.14, you can see it's definitely trending downward, but it looks pretty unstable. This is the type of plot you'd be a bit surprised to see in a supervised learning problem, but it's quite common in bare DRL. The experience replay mechanism helps with training stabilization by reducing catastrophic forgetting, but there are other related sources of instability.

3.4 Improving stability with a target network

So far, we've been able to successfully train a deep reinforcement learning algorithm to learn and play Gridworld with both a deterministic static initialization and a slightly harder version where the player is placed randomly on the board each game. Unfortunately, even though the algorithm appears to learn how to play, it is quite possible it is just memorizing all the possible board configurations, since there aren't that many on