Learning to Play Ms. Pac-Man Using Deep Q-Learning

Since we will be using an Atari environment, we must first install OpenAI gym's Atari dependencies. While we're at it, we will also install dependencies for other OpenAI gym environments that you may want to play with. On macOS, assuming you have installed Homebrew, you need to run:

```
$ brew install cmake boost boost-python sdl2 swig wget
```

On Ubuntu, type the following command (replacing `python3` with `python` if you are using Python 2):

```
$ apt-get install -y python3-numpy python3-dev cmake zlib1g-dev libjpeg-dev\
    xvfb libav-tools xorg-dev python3-opengl libboost-all-dev libsdl2-dev swig
```

Then install the extra Python modules:

```
$ pip3 install --upgrade 'gym[all]'
```

If everything went well, you should be able to create a Ms. Pac-Man environment:

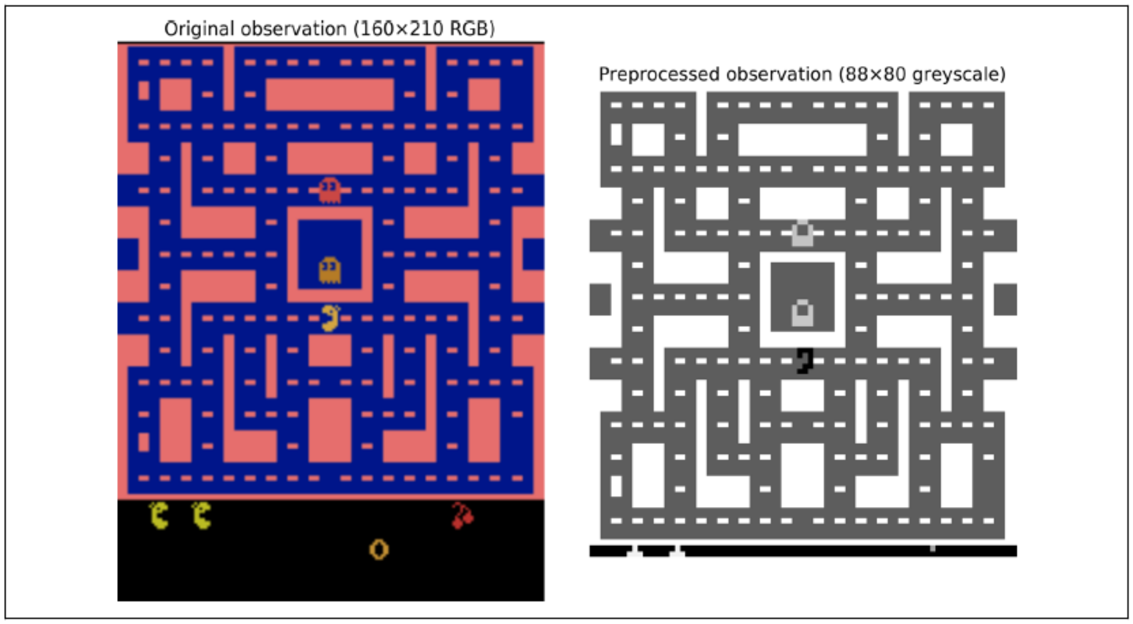
```
>>> env = gym.make("MsPacman-v0")
>>> obs = env.reset()
>>> obs.shape  # [height, width, channels]
(210, 160, 3)
>>> env.action_space
Discrete(9)
```

As you can see, there are nine discrete actions available, which correspond to the nine possible positions of the joystick (left, right, up, down, center, upper left, and so on), and the observations are simply screenshots of the Atari screen (see Figure 16-9, left), represented as 3D NumPy arrays. These images are a bit large, so we will create a small preprocessing function that will crop the image and shrink it down to 88 × 80 pixels, convert it to grayscale, and improve the contrast of Ms. Pac-Man. This will reduce the amount of computations required by the DQN, and speed up training.

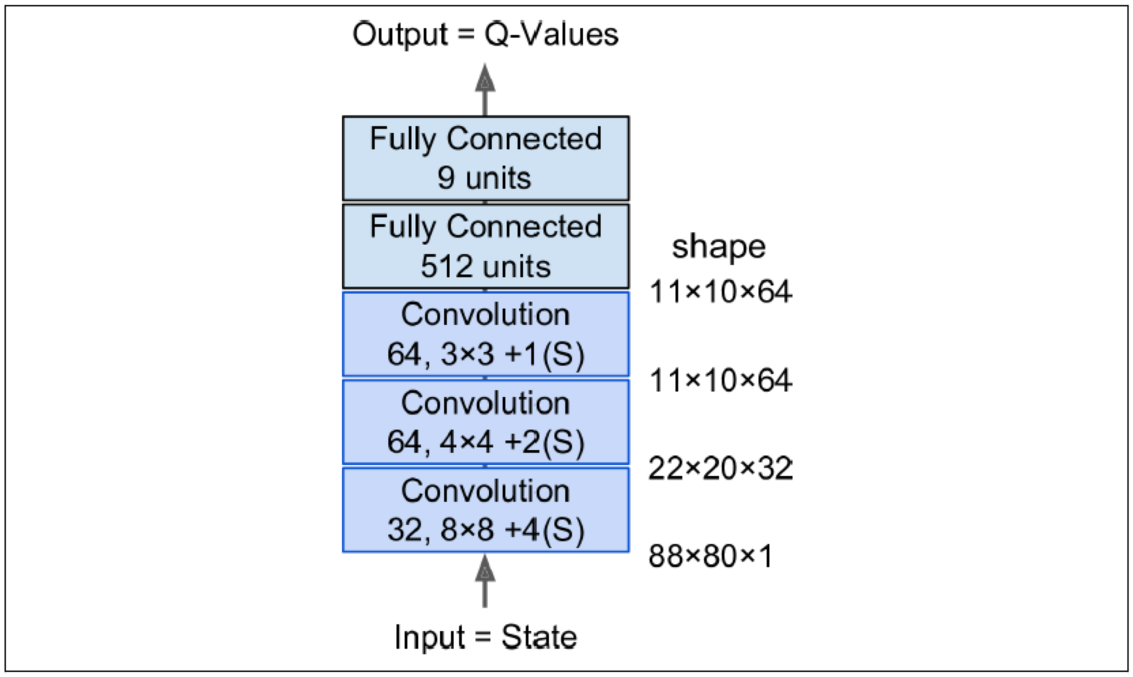
```
mspacman_color = np.array([210, 164, 74]).mean()

def preprocess_observation(obs):
    img = obs[1:176:2, ::2] # crop and downsize
    img = img.mean(axis=2) # to greyscale
    img[img==mspacman_color] = 0 # improve contrast
    img = (img - 128) / 128 - 1 # normalize from -1. to 1.
    return img.reshape(88, 80, 1)
```

The result of preprocessing is shown in Figure 16-9 (right).


Original observation (160×210 RGB)
Preprocessed observation (88×80 greyscale)
Figure 16-9. Ms. Pac-Man observation, original (left) and after preprocessing (right)

Next, let's create the DQN. It could just take a state-action pair (*s*,*a*) as input, and output an estimate of the corresponding Q-Value *Q*(*s*,*a*), but since the actions are discrete it is more convenient to use a neural network that takes only a state *s* as input and outputs one Q-Value estimate per action. The DQN will be composed of three convolutional layers, followed by two fully connected layers, including the output layer (see Figure 16-10).


Output = Q-Values
Fully Connected 9 units
Fully Connected 512 units
Convolution 64, 3×3 +1(S)
Convolution 64, 4×4 +2(S)
Convolution 32, 8×8 +4(S)
Input = State
shape
11×10×64
11×10×64
22×20×32
88×80×1
Figure 16-10. Deep Q-network to play Ms. Pac-Man

As we will see, the training algorithm we will use requires two DQNs with the same architecture (but different parameters): one will be used to drive Ms. Pac-Man during training (the *actor*), and the other will watch the actor and learn from its trials and errors (the *critic*). At regular intervals we will copy the critic to the actor. Since we need two identical DQNs, we will create a `q_network()` function to build them:

```
from tensorflow.contrib.layers import convolution2d, fully_connected

input_height = 88
input_width = 80
input_channels = 1
conv_n_maps = [32, 64, 64]
conv_kernel_sizes = [(8,8), (4,4), (3,3)]
conv_strides = [4, 2, 1]
conv_paddings = ["SAME"]*3
conv_activation = [tf.nn.relu]*3
n_hidden_in = 64 * 11 * 10  # conv3 has 64 maps of 11x10 each
n_hidden = 512
hidden_activation = tf.nn.relu
n_outputs = env.action_space.n  # 9 discrete actions are available
initializer = tf.contrib.layers.variance_scaling_initializer()

def q_network(X_state, scope):
    prev_layer = X_state
    conv_layers = []
    with tf.variable_scope(scope) as scope:
        for n_maps, kernel_size, stride, padding, activation in zip(
                conv_n_maps, conv_kernel_sizes, conv_strides,
                conv_paddings, conv_activation):
            prev_layer = convolution2d(
                prev_layer, num_outputs=n_maps, kernel_size=kernel_size,
                stride=stride, padding=padding, activation_fn=activation,
                weights_initializer=initializer)
            conv_layers.append(prev_layer)
        last_conv_layer_flat = tf.reshape(prev_layer, shape=[-1, n_hidden_in])
        hidden = fully_connected(
            last_conv_layer_flat, n_hidden, activation_fn=hidden_activation,
            weights_initializer=initializer)
        outputs = fully_connected(
            hidden, n_outputs, activation_fn=None,
            weights_initializer=initializer)
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                                       scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var
                              for var in trainable_vars}
    return outputs, trainable_vars_by_name
```

The first part of this code defines the hyperparameters of the DQN architecture. Then the `q_network()` function creates the DQN, taking the environment's state `X_state` as input, and the name of the variable scope. Note that we will just use one observation to represent the environment's state since there's almost no hidden state (except for blinking objects and the ghosts' directions).

The `trainable_vars_by_name` dictionary gathers all the trainable variables of this DQN. It will be useful in a minute when we create operations to copy the critic DQN to the actor DQN. The keys of the dictionary are the names of the variables, stripping the part of the prefix that just corresponds to the scope's name. It looks like this:

```
>>> trainable_vars_by_name
{'/Conv/biases:0': <tensorflow.python.ops.variables.Variable at 0x121cf7b50>,
 '/Conv/weights:0': <tensorflow.python.ops.variables.Variable...>,
 '/Conv_1/biases:0': <tensorflow.python.ops.variables.Variable...>,
 '/Conv_1/weights:0': <tensorflow.python.ops.variables.Variable...>,
 '/Conv_2/biases:0': <tensorflow.python.ops.variables.Variable...>,
 '/Conv_2/weights:0': <tensorflow.python.ops.variables.Variable...>,
 '/fully_connected/biases:0': <tensorflow.python.ops.variables.Variable...>,
 '/fully_connected/weights:0': <tensorflow.python.ops.variables.Variable...>,
 '/fully_connected_1/biases:0': <tensorflow.python.ops.variables.Variable...>,
 '/fully_connected_1/weights:0': <tensorflow.python.ops.variables.Variable...>}
```

Now let's create the input placeholder, the two DQNs, and the operation to copy the critic DQN to the actor DQN:

```
X_state = tf.placeholder(tf.float32, shape=[None, input_height, input_width,
                                            input_channels])
actor_q_values, actor_vars = q_network(X_state, scope="q_networks/actor")
critic_q_values, critic_vars = q_network(X_state, scope="q_networks/critic")

copy_ops = [actor_var.assign(critic_vars[var_name])
            for var_name, actor_var in actor_vars.items()]
copy_critic_to_actor = tf.group(*copy_ops)
```

Let's step back for a second: we now have two DQNs that are both capable of taking an environment state (i.e., a preprocessed observation) as input and outputting an estimated Q-Value for each possible action in that state. Plus we have an operation called `copy_critic_to_actor` to copy all the trainable variables of the critic DQN to the actor DQN. We use TensorFlow's `tf.group()` function to group all the assignment operations into a single convenient operation.

The actor DQN can be used to play Ms. Pac-Man (initially very badly). As discussed earlier, you want it to explore the game thoroughly enough, so you generally want to combine it with an *ε-greedy policy* or another exploration strategy.

But what about the critic DQN? How will it learn to play the game? The short answer is that it will try to make its Q-Value predictions match the Q-Values estimated by the actor through its experience of the game. Specifically, we will let the actor play for a while, storing all its experiences in a *replay memory*. Each memory will be a 5-tuple (state, action, next state, reward, continue), where the "continue" item will be equal to 0.0 when the game is over, or 1.0 otherwise. Next, at regular intervals we will sample a batch of memories from the replay memory, and we will estimate the Q-Values from these memories. Finally, we will train the critic DQN to predict these Q-Values using regular supervised learning techniques. Once every few training iterations, we will copy the critic DQN to the actor DQN. And that's it! Equation 16-7 shows the cost function used to train the critic DQN:

Equation 16-7. Deep Q-Learning cost function

$$J\left(\theta_{\text{critic}}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - Q\left(s^{(i)}, a^{(i)}, \theta_{\text{critic}}\right)\right)^2$$

$$\text{with } y^{(i)} = r^{(i)} + \gamma \cdot \max_{a'} Q\left(s'^{(i)}, a', \theta_{\text{actor}}\right)$$

- $s^{(i)}$, $a^{(i)}$, $r^{(i)}$ and $s'^{(i)}$ are respectively the state, action, reward, and next state of the ith memory sampled from the replay memory.
- m is the size of the memory batch.
- θ_{critic} and θ_{actor} are the critic and the actor's parameters.
- $Q(s^{(i)},a^{(i)},\theta_{\text{critic}})$ is the critic DQN's prediction of the ith memorized state-action's Q-Value.
- $Q(s'^{(i)}, a', \theta_{\text{actor}})$ is the actor DQN's prediction of the Q-Value it can expect from the next state $s'^{(i)}$ if it chooses action a'.
- $y^{(i)}$ is the target Q-Value for the ith memory. Note that it is equal to the reward actually observed by the actor, plus the actor's *prediction* of what future rewards it should expect if it were to play optimally (as far as it knows).
- $J(\theta_{\text{critic}})$ is the cost function used to train the critic DQN. As you can see, it is just the Mean Squared Error between the target Q-Values $y^{(i)}$ as estimated by the actor DQN, and the critic DQN's predictions of these Q-Values.

The replay memory is optional, but highly recommended. Without it, you would train the critic DQN using consecutive experiences that may be very correlated. This would introduce a lot of bias and slow down the training algorithm's convergence. By using a replay memory, we ensure that the memories fed to the training algorithm can be fairly uncorrelated.

Let's add the critic DQN's training operations. First, we need to be able to compute its predicted Q-Values for each state-action in the memory batch. Since the DQN outputs one Q-Value for every possible action, we need to keep only the Q-Value that corresponds to the action that was actually chosen in this memory. For this, we will convert the action to a one-hot vector (recall that this is a vector full of 0s except for a

1 at the i^th^ index), and multiply it by the Q-Values: this will zero out all Q-Values except for the one corresponding to the memorized action. Then just sum over the first axis to obtain only the desired Q-Value prediction for each memory.

```
X_action = tf.placeholder(tf.int32, shape=[None])
q_value = tf.reduce_sum(critic_q_values * tf.one_hot(X_action, n_outputs),
                        axis=1, keep_dims=True)
```

Next let's add the training operations, assuming the target Q-Values will be fed through a placeholder. We also create a nontrainable variable called `global_step`. The optimizer's `minimize()` operation will take care of incrementing it. Plus we create the usual `init` operation and a `Saver`.

```
y = tf.placeholder(tf.float32, shape=[None, 1])
cost = tf.reduce_mean(tf.square(y - q_value))
global_step = tf.Variable(0, trainable=False, name='global_step')
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(cost, global_step=global_step)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

That's it for the construction phase. Before we look at the execution phase, we will need a couple of tools. First, let's start by implementing the replay memory. We will use a `deque` list since it is very efficient at pushing items to the queue and popping them out from the end of the list when the maximum memory size is reached. We will also write a small function to randomly sample a batch of experiences from the replay memory:

```
from collections import deque

replay_memory_size = 10000
replay_memory = deque([], maxlen=replay_memory_size)

def sample_memories(batch_size):
    indices = rnd.permutation(len(replay_memory))[:batch_size]
    cols = [[], [], [], [], []] # state, action, reward, next_state, continue
    for idx in indices:
        memory = replay_memory[idx]
        for col, value in zip(cols, memory):
            col.append(value)
    cols = [np.array(col) for col in cols]
    return (cols[0], cols[1], cols[2].reshape(-1, 1), cols[3],
            cols[4].reshape(-1, 1))
```

Next, we will need the actor to explore the game. We will use the ε-greedy policy, and gradually decrease ε from 1.0 to 0.05, in 50,000 training steps:

```
eps_min = 0.05
eps_max = 1.0
eps_decay_steps = 50000
```

```
def epsilon_greedy(q_values, step):
    epsilon = max(eps_min, eps_max - (eps_max-eps_min) * step/eps_decay_steps)
    if rnd.rand() < epsilon:
        return rnd.randint(n_outputs) # random action
    else:
        return np.argmax(q_values) # optimal action
```

That's it! We have all we need to start training. The execution phase does not contain anything too complex, but it is a bit long, so take a deep breath. Ready? Let's go! First, let's initialize a few variables:

```
n_steps = 100000  # total number of training steps
training_start = 1000  # start training after 1,000 game iterations
training_interval = 3  # run a training step every 3 game iterations
save_steps = 50  # save the model every 50 training steps
copy_steps = 25  # copy the critic to the actor every 25 training steps
discount_rate = 0.95
skip_start = 90  # skip the start of every game (it's just waiting time)
batch_size = 50
iteration = 0  # game iterations
checkpoint_path = "./my_dqn.ckpt"
done = True # env needs to be reset
```

Next, let's open the session and run the main training loop:

```
with tf.Session() as sess:
    if os.path.isfile(checkpoint_path):
        saver.restore(sess, checkpoint_path)
    else:
        init.run()
    while True:
        step = global_step.eval()
        if step >= n_steps:
            break
        iteration += 1
        if done: # game over, start again
            obs = env.reset()
            for skip in range(skip_start): # skip the start of each game
                obs, reward, done, info = env.step(0)
            state = preprocess_observation(obs)

        # Actor evaluates what to do
        q_values = actor_q_values.eval(feed_dict={X_state: [state]})
        action = epsilon_greedy(q_values, step)

        # Actor plays
        obs, reward, done, info = env.step(action)
        next_state = preprocess_observation(obs)

        # Let's memorize what just happened
        replay_memory.append((state, action, reward, next_state, 1.0 - done))
        state = next_state
        if iteration < training_start or iteration % training_interval != 0:
            continue

        # Critic learns
        X_state_val, X_action_val, rewards, X_next_state_val, continues = (
            sample_memories(batch_size))
        next_q_values = actor_q_values.eval(
            feed_dict={X_state: X_next_state_val})
        max_next_q_values = np.max(next_q_values, axis=1, keepdims=True)
        y_val = rewards + continues * discount_rate * max_next_q_values
        training_op.run(feed_dict={X_state: X_state_val,
                                   X_action: X_action_val, y: y_val})

        # Regularly copy critic to actor
        if step % copy_steps == 0:
            copy_critic_to_actor.run()

        # And save regularly
        if step % save_steps == 0:
            saver.save(sess, checkpoint_path)
```

We start by restoring the models if a checkpoint file exists, or else we just initialize the variables normally. Then the main loop starts, where `iteration` counts the total number of game steps we have gone through since the program started, and `step` counts the total number of training steps since training started (if a checkpoint is restored, the global step is restored as well). Then the code resets the game (and skips the first boring game steps, where nothing happens). Next, the actor evaluates what to do, and plays the game, and its experience is memorized in replay memory. Then, at regular intervals (after a warmup period), the critic goes through a training step. It samples a batch of memories and asks the actor to estimate the Q-Values of all actions for the next state, and it applies Equation 16-7 to compute the target Q-Value `y_val`. The only tricky part here is that we must multiply the next state's Q-Values by the `continues` vector to zero out the Q-Values corresponding to memories where the game was over. Next we run a training operation to improve the critic's ability to predict Q-Values. Finally, at regular intervals we copy the critic to the actor, and we save the model.

Unfortunately, training is very slow: if you use your laptop for training, it will take days before Ms. Pac-Man gets any good, and if you look at the learning curve, measuring the average rewards per episode, you will notice that it is extremely noisy. At some points there may be no apparent progress for a very long time until suddenly the agent learns to survive a reasonable amount of time. As mentioned earlier, one solution is to inject as much prior knowledge as possible into the model (e.g., through preprocessing, rewards, and so on), and you can also try to bootstrap the model by first training it to imitate a basic strategy. In any case, RL still requires quite a lot of patience and tweaking, but the end result is very exciting.