DL in Applied Mathematics

Lecture 13: Temporal Difference Learning and Q-Learning

Marat Nurtas
PhD in Mathematical and Computer Modeling
Department of Mathematical and Computer Modeling
International Information Technology University, Almaty, Kazakhstan

The *Temporal Difference Learning* (TD Learning) algorithm is very similar to the Value Iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general we assume that the agent initially knows only the possible states and actions, and nothing more.

Equation 16-4. TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha\left(r + \gamma \,.\, V_k(s')\right)$$

- α is the learning rate (e.g., 0.01).

Similarly, the Q-Learning algorithm is an adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see Equation 16-5).

Equation 16-5. Q-Learning algorithm

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\left(r + \gamma \,.\, \max_{a'} Q_k(s',a')\right)$$

Here is how Q-Learning can be implemented:

```
import numpy.random as rnd

learning_rate0 = 0.05
learning_rate_decay = 0.1
n_iterations = 20000

s = 0 # start in state 0

Q = np.full((3, 3), -np.inf)  # -inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0  # Initial value = 0.0, for all possible actions

for iteration in range(n_iterations):
    a = rnd.choice(possible_actions[s])  # choose an action (randomly)
    sp = rnd.choice(range(3), p=T[s, a]) # pick next state using T[s, a]
    reward = R[s, a, sp]
    learning_rate = learning_rate0 / (1 + iteration * learning_rate_decay)
    Q[s, a] = learning_rate * Q[s, a] + (1 - learning_rate) * (
            reward + discount_rate * np.max(Q[sp])
        )
    s = sp # move to next state
```

Exploration Policies

Alternatively, rather than relying on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before. This can be implemented as a bonus added to the Q-Value estimates, as shown in Equation 16-6.

Equation 16-6. Q-Learning using an exploration function

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \cdot \max_{a'} f\left(Q(s',a'), N(s',a')\right)\right)$$

- $N(s', a')$ counts the number of times the action a' was chosen in state s'.
- $f(q, n)$ is an *exploration function*, such as $f(q, n) = q + K/(1 + n)$, where K is a curiosity hyperparameter that measures how much the agent is attracted to to the unknown.

Approximate Q-Learning

The main problem with Q-Learning is that it does not scale well to large (or even medium) MDPs with many states and actions.

The solution is to find a function that approximates the Q-Values using a manageable number of parameters. This is called *Approximate Q-Learning*.

A DNN used to estimate Q-Values is called a *deep Q-network* (DQN), and using a DQN for Approximate Q-Learning is called *Deep Q-Learning*.

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

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).

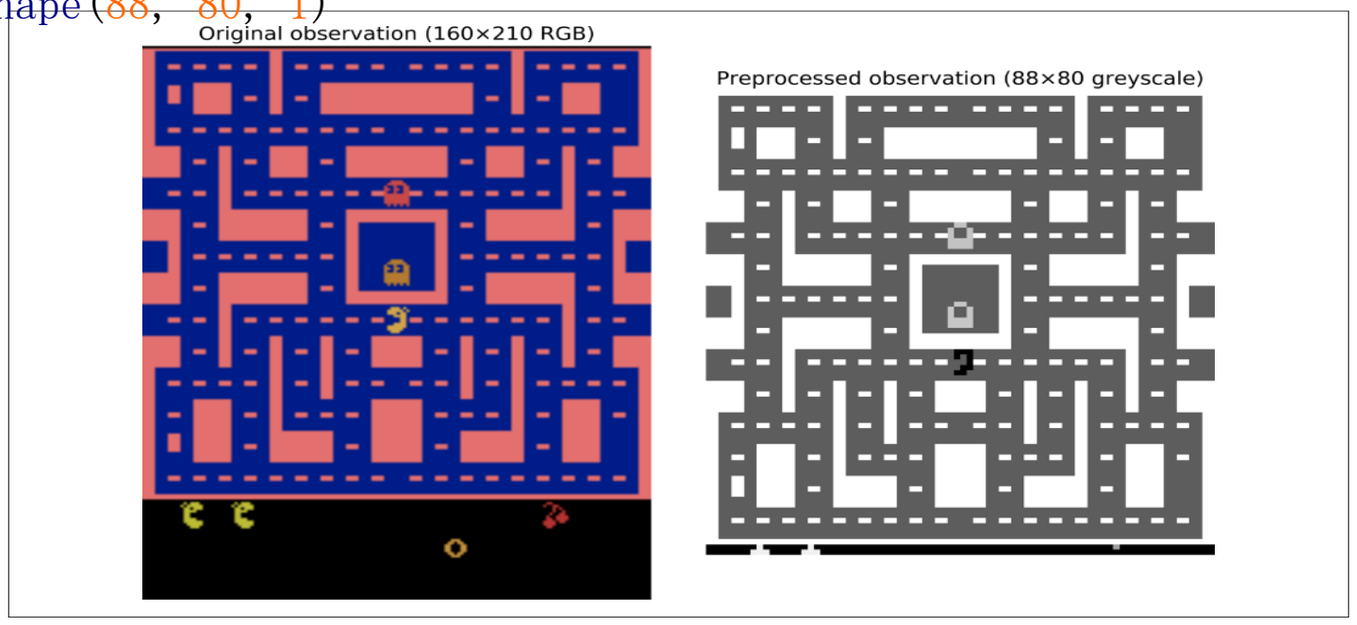

Figure 16-9. Ms. Pac-Man observation, original (left) and after preprocessing (right)

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

The DQN will be composed of three convolutional layers, followed by two fully connected layers, including the output layer (see Figure 16-10).

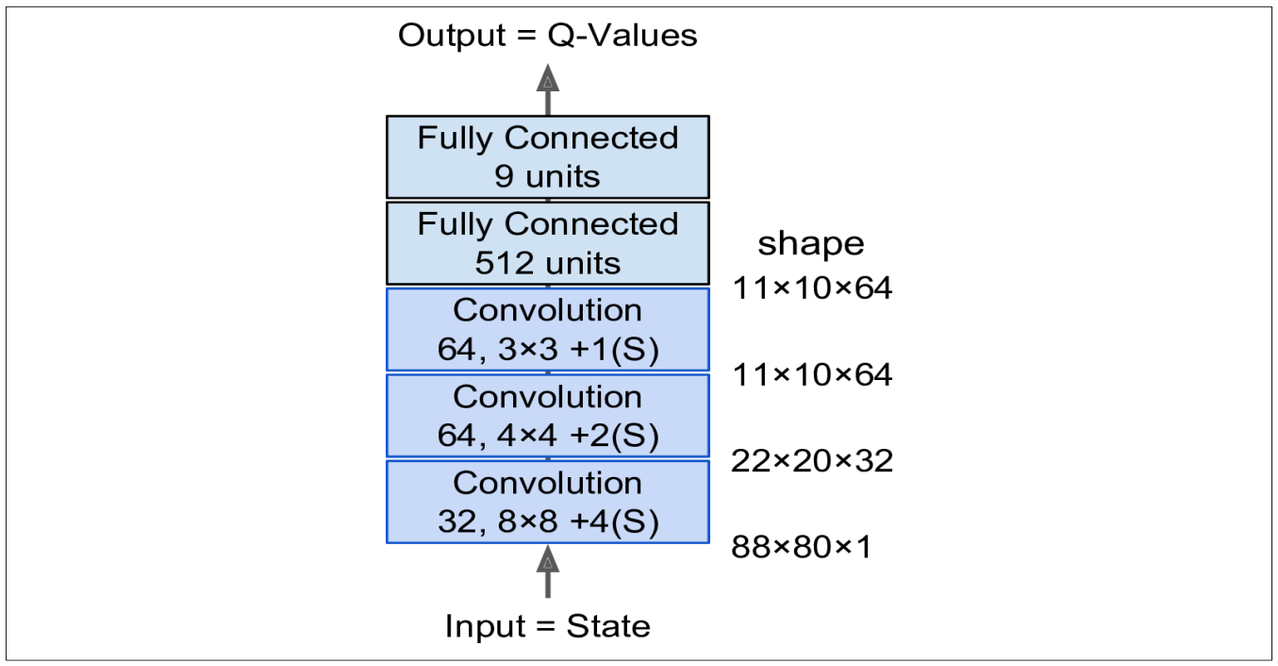

Figure 16-10. Deep Q-network to play Ms. Pac-Man

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

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
```

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

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...>}
```

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

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)
```

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)
```

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

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()
```

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

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))
```

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

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
```

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

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
```

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

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))

        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:
```

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

Algorithm

- 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.

In any case, RL still requires quite a lot of patience and tweaking, but the end result is very exciting.