DEEP Q-LEARNING *

Sinjoy Saha
The Pennsylvania State University

ABSTRACT

In this project, we try to understand the implementation of a reinforcement learning code. Specifically, we try to understand the implementation of Deep Q Learning to play Atari games by Denny Britz. The main code is in the *dqn.py* and can be further explored using the Deep Q-Learning Jupyter notebooks given the the repository. The code contains two main classes: a) StateProcessor b) Estimator, along with a few helper functions. The driver function is called *deep_q_learning()*. This paper describes the functions and working of each piece of code with comments wherever required, in addition to the comments already present in the code.

***K*eywords** reinforcement learning · deep reinforcement learning · Q-learning

1 Introduction

In this project, we try to understand the implementation of a reinforcement learning code. Specifically, we try to understand the implementation of Deep Q Learning to play Atari games by Denny Britz[2]. The main code is in the *dqn.py* and can be further explored using the Deep Q-Learning Jupyter notebooks given the the repository.

This code implements Deep Q-Learning (DQN), training an agent to learn optimal actions in an environment through Q-value estimation. It uses experience replay to store and sample past experiences for stable learning, and an epsilon-greedy policy to balance exploration and exploitation. The agentâĂŹs Q-network estimates action values, while a separate target network stabilizes training by providing fixed targets, thus the name double DQN. For each episode, the agent performs actions, updates the replay memory, and periodically trains on sampled batches. Target network updates occur at intervals, and metrics are logged and saved for analysis, while checkpoints allow training to resume if interrupted.

The code contains two main classes: a) StateProcessor b) Estimator, along with a few helper functions. The driver function is called *deep_q_learning()*. Specifically, we study the Python code which balances exploration and exploitation of the RL agent using an epsilon-greedy strategy, gradually shifting from exploration to exploitation near the end of training. It trains the Q-network to minimize the MSE loss between the predicted Q-values and the target Q-values. It also saves the weights or parameters of the model to give fixed target to stabilize the training of the RL agent. The CNN model used here "watches" the game being played using the OpenAI library Gym[3]. It makes the predictions for the chosen actions using a Q-network. The following sections in the paper describe the functions and working of each piece of code with comments wherever required, in addition to the comments already present in the code. Specifically, the State Processor, Estimator, policy function and the training loop codes are described[1]

*Submission for homework 2 for CSE 584 - Machine Learning: Tools and Algorithms.

[2]https://github.com/dennybritz/reinforcement-learning

[3]https://github.com/openai/gym

2 Code Description

2.1 State Processor

The StateProcessor class has two parameters *input_state* and *output*. The initializations are done in the *__init__()* method. It also has a *process()* method which running the TensorFlow model/session to process a given input state *state* and store the result in the *output* variable.

```
class StateProcessor():
    """
    Processes a raw Atari images. Resizes it and converts it to grayscale.
    """
    def __init__(self):
        # Build the Tensorflow graph
        with tf.variable_scope("state_processor"):
            self.input_state = tf.placeholder(shape=[210, 160, 3], dtype=tf.uint8)
            self.output = tf.image.rgb_to_grayscale(self.input_state)
            self.output = tf.image.crop_to_bounding_box(self.output, 34, 0, 160,
                                                        160)
            self.output = tf.image.resize_images(
                self.output, [84, 84], method=tf.image.ResizeMethod.
                                                        NEAREST_NEIGHBOR)
            self.output = tf.squeeze(self.output)

    def process(self, sess, state):
        """
        Args:
            sess: A Tensorflow session object
            state: A [210, 160, 3] Atari RGB State

        Returns:
            A processed [84, 84] state representing grayscale values.
        """
        return sess.run(self.output, { self.input_state: state })
```

2.2 Estimator

The Estimator class implements a Q-Value neural network using the TensorFlow [4]library. This network is used for both the Q-network and the target network.

The main model is implemented in the *_build_model()* method. It created a three-layer Convolutional Neural Network of 32, 64 and 64 filters, 8x8, 4x4 and 3x3 kernels and 4, 2 and 1 strides respectively. Then, the output of the convolutional layers goes into a fully-connected layer of 512 neurons.

The loss used here is the mean-squared error (MSE) function between the predictions of the chosen actions and the target actions. It is important to apply the Bellman equation to the chosen action only and not all the actions.

The *predict()* method takes in an input state s and applies the model to compute the Q-values (action-value predictions) for a given batch of states.

The *update()* method is used to update the model parameters based on the given states, actions and target Q-values to minimize the training (MSE) loss.

The training loss is minimized using the RMSProp optimizer.

Additionally, there is provision for writing the training parameters and losses into log files to visualize using TensorBoard library.

```
class Estimator():
    """Q-Value Estimator neural network.

    This network is used for both the Q-Network and the Target Network.
    """
```

[4]https://www.tensorflow.org/

```
    def __init__(self, scope="estimator", summaries_dir=None):
        self.scope = scope
        # Writes Tensorboard summaries to disk
        self.summary_writer = None
        with tf.variable_scope(scope):
            # Build the graph
            self._build_model()
            if summaries_dir:
                summary_dir = os.path.join(summaries_dir, "summaries_{}".format(
                                                        scope))
                if not os.path.exists(summary_dir):
                    os.makedirs(summary_dir)
                self.summary_writer = tf.summary.FileWriter(summary_dir)

    def _build_model(self):
        """
        Builds the Tensorflow graph.
        """

        # Placeholders for our input
        # Our input are 4 RGB frames of shape 160, 160 each
        self.X_pl = tf.placeholder(shape=[None, 84, 84, 4], dtype=tf.uint8, name="
                                                X")
        # The TD target value
        self.y_pl = tf.placeholder(shape=[None], dtype=tf.float32, name="y")
        # Integer id of which action was selected
        # This will be used later to get the chosen actions
        # as we should apply the Bellman equation to the chosen actions only
        self.actions_pl = tf.placeholder(shape=[None], dtype=tf.int32, name="
                                                actions")

        X = tf.to_float(self.X_pl) / 255.0
        batch_size = tf.shape(self.X_pl)[0]

        # Three convolutional layers
        conv1 = tf.contrib.layers.conv2d(
            X, 32, 8, 4, activation_fn=tf.nn.relu)
        conv2 = tf.contrib.layers.conv2d(
            conv1, 64, 4, 2, activation_fn=tf.nn.relu)
        conv3 = tf.contrib.layers.conv2d(
            conv2, 64, 3, 1, activation_fn=tf.nn.relu)

        # Fully connected layers
        flattened = tf.contrib.layers.flatten(conv3)
        fc1 = tf.contrib.layers.fully_connected(flattened, 512)
        self.predictions = tf.contrib.layers.fully_connected(fc1, len(
                                                VALID_ACTIONS))

        # Get the predictions for the chosen actions only (as from before)
        gather_indices = tf.range(batch_size) * tf.shape(self.predictions)[1] +
                                                self.actions_pl
        self.action_predictions = tf.gather(tf.reshape(self.predictions, [-1]),
                                                gather_indices)

        # Calculate the training loss
        # This is essentially the mean-squared error between predicted and target
                                                Q-values
        self.losses = tf.squared_difference(self.y_pl, self.action_predictions)
        self.loss = tf.reduce_mean(self.losses)

        # Optimizer Parameters from original paper
        self.optimizer = tf.train.RMSPropOptimizer(0.00025, 0.99, 0.0, 1e-6)
        self.train_op = self.optimizer.minimize(self.loss, global_step=tf.contrib.
                                                framework.get_global_step())
```

```
        # Summaries for Tensorboard
        self.summaries = tf.summary.merge([
            tf.summary.scalar("loss", self.loss),
            tf.summary.histogram("loss_hist", self.losses),
            tf.summary.histogram("q_values_hist", self.predictions),
            tf.summary.scalar("max_q_value", tf.reduce_max(self.predictions))
        ])


    def predict(self, sess, s):
        """
        Predicts action values.

        Args:
          sess: Tensorflow session
          s: State input of shape [batch_size, 4, 160, 160, 3]

        Returns:
          Tensor of shape [batch_size, NUM_VALID_ACTIONS] containing the estimated
          action values.
        """
        return sess.run(self.predictions, { self.X_pl: s })

    def update(self, sess, s, a, y):
        """
        Updates the estimator towards the given targets.

        Args:
          sess: Tensorflow session object
          s: State input of shape [batch_size, 4, 160, 160, 3]
          a: Chosen actions of shape [batch_size]
          y: Targets of shape [batch_size]

        Returns:
          The calculated loss on the batch.
        """
        feed_dict = { self.X_pl: s, self.y_pl: y, self.actions_pl: a }
        summaries, global_step, _, loss = sess.run(
            [self.summaries, tf.contrib.framework.get_global_step(), self.train_op
                                                    , self.loss],
            feed_dict)
        if self.summary_writer:
            self.summary_writer.add_summary(summaries, global_step)
        return loss
```

2.3 Policy Function

This method generates an epsilon-greedy policy function for an RL agent using a given Q-value estimator. This policy is useful in Deep Q-Learning, where an agent balances between exploring new actions (with probability epsilon ϵ) and exploiting the best-known action (with probability $1 - \epsilon$).

```
def make_epsilon_greedy_policy(estimator, nA):
    """
    Creates an epsilon-greedy policy based on a given Q-function approximator and
                                            epsilon.

    Args:
        estimator: An estimator that returns q values for a given state
        nA: Number of actions in the environment.

    Returns:
        A function that takes the (sess, observation, epsilon) as an argument and
                                                returns
```

```
        the probabilities for each action in the form of a numpy array of length
                                                    nA.

    """
    def policy_fn(sess, observation, epsilon):
        A = np.ones(nA, dtype=float) * epsilon / nA
        # Predicts the Q-values using the Q-network
        q_values = estimator.predict(sess, np.expand_dims(observation, 0))[0]
        # The best action is the action id with the highest Q-value (greedy)
        best_action = np.argmax(q_values)
        # Exploitation
        A[best_action] += (1.0 - epsilon)
        return A
    return policy_fn
```

2.4 Training Loop

This code defines a deep_q_learning() method to implement the Deep Q-Learning (DQN) algorithm for training an agent using function approximation, specifically for OpenAI Gym.

A replay memory buffer is used which stores experience tuples (state, action, reward, next_state, done) to help stabilize training of the RL agent. The training gradually shifts from exploration to exploitation, thus the epsilon values gradually decrease from epsilon_start to epsilon_end over epsilon_decay_steps.

Before training, the function populates the replay memory with random transitions, ensuring that the initial training batch has diverse examples. The agent performs random actions in the environment to gather state transitions, which it stores in the replay memory.

Every few steps, the weights from q_estimator are copied to target_estimator, helping to stabilize Q-learning by providing a fixed target.

The policy uses the epsilon-greedy method, balancing between exploring random actions and exploiting the best-known action. The agent takes the selected action, receives a reward, and stores the experience in replay memory. A random batch of experiences is sampled from replay memory, then the Q-values are computed for the next state using the Q-network and target network (Double DQN).

The best actions for the next state are selected using the Q-network, and target Q-values are computed using the target network. The target is calculated by combining immediate rewards and the Q-values for the next states, adjusting for terminal states (by setting the target as only the reward when done). The Q-networkâĂŹs weights are updated to minimize the loss between the predicted Q-values and target Q-values.

```
def deep_q_learning(sess,
                    env,
                    q_estimator,
                    target_estimator,
                    state_processor,
                    num_episodes,
                    experiment_dir,
                    replay_memory_size=500000,
                    replay_memory_init_size=50000,
                    update_target_estimator_every=10000,
                    discount_factor=0.99,
                    epsilon_start=1.0,
                    epsilon_end=0.1,
                    epsilon_decay_steps=500000,
                    batch_size=32,
                    record_video_every=50):
    """
    Q-Learning algorithm for off-policy TD control using Function Approximation.
    Finds the optimal greedy policy while following an epsilon-greedy policy.

    Args:
        sess: Tensorflow Session object
        env: OpenAI environment
```

```
        q_estimator: Estimator object used for the q values
        target_estimator: Estimator object used for the targets
        state_processor: A StateProcessor object
        num_episodes: Number of episodes to run for
        experiment_dir: Directory to save Tensorflow summaries in
        replay_memory_size: Size of the replay memory
        replay_memory_init_size: Number of random experiences to sampel when
                                                    initializing
          the reply memory.
        update_target_estimator_every: Copy parameters from the Q estimator to the
          target estimator every N steps
        discount_factor: Gamma discount factor
        epsilon_start: Chance to sample a random action when taking an action.
          Epsilon is decayed over time and this is the start value
        epsilon_end: The final minimum value of epsilon after decaying is done
        epsilon_decay_steps: Number of steps to decay epsilon over
        batch_size: Size of batches to sample from the replay memory
        record_video_every: Record a video every N episodes

    Returns:
        An EpisodeStats object with two numpy arrays for episode_lengths and
                                                    episode_rewards.
    """

    Transition = namedtuple("Transition", ["state", "action", "reward", "
                                            next_state", "done"])

    # The replay memory
    replay_memory = []

    # Keeps track of useful statistics
    stats = plotting.EpisodeStats(
        episode_lengths=np.zeros(num_episodes),
        episode_rewards=np.zeros(num_episodes))

    # Create directories for checkpoints and summaries
    checkpoint_dir = os.path.join(experiment_dir, "checkpoints")
    checkpoint_path = os.path.join(checkpoint_dir, "model")
    monitor_path = os.path.join(experiment_dir, "monitor")

    if not os.path.exists(checkpoint_dir):
        os.makedirs(checkpoint_dir)
    if not os.path.exists(monitor_path):
        os.makedirs(monitor_path)

    saver = tf.train.Saver()
    # Load a previous checkpoint if we find one
    latest_checkpoint = tf.train.latest_checkpoint(checkpoint_dir)
    if latest_checkpoint:
        print("Loading model checkpoint {}...\n".format(latest_checkpoint))
        saver.restore(sess, latest_checkpoint)

    total_t = sess.run(tf.contrib.framework.get_global_step())

    # The epsilon decay schedule
    # Gradually go from exploration to exploitation
    epsilons = np.linspace(epsilon_start, epsilon_end, epsilon_decay_steps)

    # The policy we're following i.e., epsilon-greedy policy
    policy = make_epsilon_greedy_policy(
        q_estimator,
        len(VALID_ACTIONS))

    # Populate the replay memory with initial experience
    print("Populating replay memory...")
```

```
    state = env.reset()
    state = state_processor.process(sess, state)
    state = np.stack([state] * 4, axis=2)
    for i in range(replay_memory_init_size):
        action_probs = policy(sess, state, epsilons[min(total_t,
                                                        epsilon_decay_steps-1)])
        action = np.random.choice(np.arange(len(action_probs)), p=action_probs)
        next_state, reward, done, _ = env.step(VALID_ACTIONS[action])
        next_state = state_processor.process(sess, next_state)
        next_state = np.append(state[:,:,1:], np.expand_dims(next_state, 2), axis=
                                                        2)
        replay_memory.append(Transition(state, action, reward, next_state, done))
        if done:
            state = env.reset()
            state = state_processor.process(sess, state)
            state = np.stack([state] * 4, axis=2)
        else:
            state = next_state

    # Record videos
    # Use the gym env Monitor wrapper
    env = Monitor(env,
                  directory=monitor_path,
                  resume=True,
                  video_callable=lambda count: count % record_video_every ==0)

    for i_episode in range(num_episodes):

        # Save the current checkpoint
        saver.save(tf.get_default_session(), checkpoint_path)

        # Reset the environment
        state = env.reset()
        state = state_processor.process(sess, state)
        state = np.stack([state] * 4, axis=2)
        loss = None

        # One step in the environment
        for t in itertools.count():

            # Epsilon for this time step
            epsilon = epsilons[min(total_t, epsilon_decay_steps-1)]

            # Add epsilon to Tensorboard
            episode_summary = tf.Summary()
            episode_summary.value.add(simple_value=epsilon, tag="epsilon")
            q_estimator.summary_writer.add_summary(episode_summary, total_t)

            # Maybe update the target estimator
            if total_t % update_target_estimator_every == 0:
                copy_model_parameters(sess, q_estimator, target_estimator)
                print("\nCopied model parameters to target network.")

            # Print out which step we're on, useful for debugging.
            print("\rStep {} ({}) @ Episode {}/{}, loss: {}".format(
                    t, total_t, i_episode + 1, num_episodes, loss), end="")
            sys.stdout.flush()

            # Take a step
            action_probs = policy(sess, state, epsilon)
            # Sample actions with action_probs distribution which is the policy
            action = np.random.choice(np.arange(len(action_probs)), p=action_probs
                                                            )
            # Take the step in the environment after validation of steps
            # If done is True, we will break from this step loop below
```

```
            next_state, reward, done, _ = env.step(VALID_ACTIONS[action])
            next_state = state_processor.process(sess, next_state)
            next_state = np.append(state[:,:,1:], np.expand_dims(next_state, 2),
                                                   axis=2)

            # If our replay memory is full, pop the first element
            if len(replay_memory) == replay_memory_size:
                replay_memory.pop(0)

            # Save transition to replay memory
            replay_memory.append(Transition(state, action, reward, next_state,
                                                   done))

            # Update statistics
            stats.episode_rewards[i_episode] += reward
            stats.episode_lengths[i_episode] = t

            # Sample a minibatch from the replay memory
            samples = random.sample(replay_memory, batch_size)
            states_batch, action_batch, reward_batch, next_states_batch,
                                                   done_batch = map(np.array,
                                                   zip(*samples))

            # Calculate q values and targets (Double DQN)
            q_values_next = q_estimator.predict(sess, next_states_batch)
            best_actions = np.argmax(q_values_next, axis=1)
            q_values_next_target = target_estimator.predict(sess,
                                                   next_states_batch)
            targets_batch = reward_batch + np.invert(done_batch).astype(np.float32
                                                   ) * \
                discount_factor * q_values_next_target[np.arange(batch_size),
                                                       best_actions]

            # Perform gradient descent update
            states_batch = np.array(states_batch)
            loss = q_estimator.update(sess, states_batch, action_batch,
                                                   targets_batch)

            if done:
                break

            state = next_state
            total_t += 1

        # Add summaries to tensorboard
        episode_summary = tf.Summary()
        episode_summary.value.add(simple_value=stats.episode_rewards[i_episode],
                                               node_name="episode_reward", tag="
                                               episode_reward")
        episode_summary.value.add(simple_value=stats.episode_lengths[i_episode],
                                               node_name="episode_length", tag="
                                               episode_length")
        q_estimator.summary_writer.add_summary(episode_summary, total_t)
        q_estimator.summary_writer.flush()

        yield total_t, plotting.EpisodeStats(
            episode_lengths=stats.episode_lengths[:i_episode+1],
            episode_rewards=stats.episode_rewards[:i_episode+1])

    env.monitor.close()
    return stats
```

3 Conclusion

In this work, we study the Deep Q-Learning (DQN) implementation. Specifically, we study the implementation of an epsilon-greedy strategy to balance exploration and exploitation. Moreover, the training implements episodic training, slowly shifting from exploration to exploitation near the end of training. The code saves the episodes and trains the Q-network to minimize the MSE loss between the predicted Q-values and the target Q-values.

References

[1] Denny Britz. Deep q learning github repo. Available at `https://github.com/dennybritz/reinforcement-learning/tree/master/DQN`, 2019.