Name: Sai Chandana Priya Avvaru
ID: 972317206

CSE 584 – Machine Learning
Homework - 2

Abstract:

DQN is the code I chose, and it makes use of reinforcement Learning. Let me explain how DQN, one of the most popular reinforcement learning algorithms that solves the Markov Decision Processes with discrete action spaces, is implemented. This DQN will use experience replay and a target network to make it more stable and improve its performance. It will interact with an environment, collect experiences, and train a neural network to approximate an optimal action-value function. The agent adopts an epsilon-greedy strategy for exploring the environment, which should be decreased gradually. Features like Double Q-Learning and Prioritized Experience Replay are implemented to ensure high efficiency in learning. The learning method coordinates the training loop: action selection, storing experience, and updating the network are handled while logging and performance metrics are managed.

Double Q-learning reduces overestimation bias by having a different network select and evaluate each action. Dueling architecture splits the value and advantage functions while maintaining its ability to learn state value more effectively. Prioritized Experience Replay enables the model to sample more important transitions more often to enhance learning efficiency.

It initializes the environment, starts experiencing the environment by interacting with it, and updates the Q-values using the Bellman equation. A model constantly explores an action space using the epsilon-greedy strategy and hence always balances the rate of exploration and exploitation during training. Key hyperparameters that can be tuned include the learning rate, discounting factor, and exploration strategies, and are hence flexible for a variety of tasks and environments.

This implementation provides a general robust framework for training DQN agents suitable for several reinforcement learning tasks, from playing different kinds of Atari games to robotic control.

Deep Q-Network:

DQN is a significant advancement in reinforcement learning that utilizes deep learning techniques to approximate Q-values, enabling agents to learn in complex environments. Its innovative use of experience replay and target networks enhances the stability and efficiency of the learning process, making it a powerful tool in the field of artificial intelligence.
Link to code:
https://github.com/hill-a/stable-baselines/blob/master/stable_baselines/deepq/dqn.py

While going through the code I found these parts of the code as crucial functions:

1. *setup_model*
2. *learn*

Core Section:

1. **Replay Buffer -**
   - Purpose: The replay buffer stores past experiences (transitions of state, action, reward, next state) that the agent collects while interacting with the environment.
   - Functionality: By storing a diverse set of experiences, the agent can sample these experiences randomly during training. This breaks the correlation between consecutive

samples, which helps stabilize the learning process. Instead of learning from the most recent transition (which could be biased), the agent learns from a variety of past experiences.

if self.prioritized_replay:
self.replay_buffer=PrioritizedReplayBuffer(self.buffer_size,alpha=self.prioritized_replay_alpha)
else:
self.replay_buffer = ReplayBuffer(self.buffer_size)

2. **Double Q-Learning -**
   - **<u>Purpose</u>:** Double Q-Learning aims to reduce overestimation bias in value estimation. Standard Q-learning can overestimate the value of actions when only one value function is used.
   - **<u>Functionality</u>:** Double Q-Learning maintains two separate value functions (or Q-values) and updates one while evaluating the other. This means that when selecting the best action, the Q-value used for that action is obtained from one function, while the value update comes from the other function.

3. **Prioritized Experience Replay -**
   - **<u>Purpose</u>:** Prioritized Experience Replay allows the agent to sample experiences based on their importance, rather than uniformly. This can help the agent learn more efficiently from transitions that are more informative.
   - **<u>Functionality</u>:** The importance of each experience can be measured using the temporal-difference (TD) error. Experiences with higher TD errors indicate that the agent's prediction was less accurate, making them more important for learning. This results in faster convergence as the agent focuses more on learning from critical experiences.

   if self.prioritized_replay:

   self.replay_buffer=PrioritizedReplayBuffer(self.buffer_size,alpha=self.prioritized_replay_alpha)

4. **Exploration Strategy -**
   - **<u>Purpose</u>:** The exploration strategy is crucial for balancing exploration (trying new actions) and exploitation (using known information to maximize rewards).
   - **<u>Functionality</u>:** An epsilon-greedy strategy is a common approach where the agent chooses a random action with a probability of epsilon (exploration) and the best-known action with a probability of (1 - epsilon) (exploitation). Over time, epsilon decreases from its initial value to a minimum value, encouraging the agent to rely more on learned actions as training progresses.

*self.exploration = LinearSchedule(schedule_timesteps=int(self.exploration_fraction * total_timesteps), initial_p=self.exploration_initial_eps, final_p=self.exploration_final_eps)*

<u>Code snippet with comments:</u>

1. *setup_model :*

This function is responsible for defining and configuring the NN Architecture that will be used for approximating the Q-values in the reinforcement learning setting. It checks that the provided action space and policy are appropriate, sets up the TensorFlow graph and session, defines the optimizer, builds the training operations, initializes the model parameters, and prepares summaries for monitoring the training process.

setup_model function:

```
def setup_model(self):
    #set up of DQN Model
    with SetVerbosity(self.verbose):
        # setup up verbosity for logging and ensuring that the action space is discrete, as
required for DQN.
        assert not isinstance(self.action_space, gym.spaces.Box), \
            "Error: DQN cannot output a gym.spaces.Box action space."
        if isinstance(self.policy, partial):
            test_policy = self.policy.func  # Unwrap the partial function to get the actual
policy class.
        else:
            test_policy = self.policy
        assert issubclass(test_policy, DQNPolicy), "Error: the input policy for the DQN
model must be an instance of DQNPolicy."
        # Check if the provided policy is a DQNPolicy .
        self.graph = tf.Graph()  # Create a new computational graph for TensorFlow.
        with self.graph.as_default():  # Set the default graph to the one created.
            self.set_random_seed(self.seed)  # Set the random seed for reproducibility.
            # Create a TensorFlow session for running the graph.
            self.sess = tf_util.make_session(num_cpu=self.n_cpu_tf_sess, graph=self.graph)
            optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate)
 # Adam optimizer for training.
            # Build the training operations and the model's step function.
            self.act, self._train_step, self.update_target, self.step_model = build_train
(
                q_func=partial(self.policy, **self.policy_kwargs),
 # Policy function with its parameters.
                ob_space=self.observation_space,  # Observation space of the environment.
                ac_space=self.action_space,  # Action space of the environment.
                optimizer=optimizer,  # Optimizer to use for training.
                gamma=self.gamma,  # Discount factor for future rewards.
                grad_norm_clipping=10,  # Clip gradients to avoid exploding gradients.
                param_noise=self.param_noise,
 # Whether to apply parameter noise for exploration.
                sess=self.sess,  # TensorFlow session.
                full_tensorboard_log=self.full_tensorboard_log,
  # Log detailed info for TensorBoard.
                double_q=self.double_q  # Enable Double Q-learning.
            )
            self.proba_step = self.step_model.proba_step  # Function to get action
probabilities.
            self.params = tf_util.get_trainable_vars("deepq")  # Get all trainable
variables in the model.
            # Initialize model parameters and copy them to the target network.
            tf_util.initialize(self.sess)  # Initialize all variables in the session.
            self.update_target(sess=self.sess)  # Copy parameters to the target network.
            self.summary = tf.summary.merge_all()  # Merge all summaries for TensorBoard
logging.
```

- *“with SetVerbosity(self.verbose):*
  assert not isinstance(self.action_space, gym.spaces.Box), \ ”

  here the verbosity is set for logging and debugging and checks action space is not box type as DQN often uses continuous action spaces.

- *“assert issubclass(test_policy, DQNPolicy) ”*

  we make sure that we are using the appropriate subclass of DQNPolicy.

- *self.graph = tf.Graph()*

  with self.graph.as_default():

self.set_random_seed(self.seed)

A new TensorFlow graph is created. This graph will contain all operations and variables needed for the model. The set_random_seed function sets the random seed for reproducibility, ensuring that results can be replicated.

- *"optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate)"*

  We are using adam optimizer for training the model.

- *"self.proba_step = self.step_model.proba_step"*

  This is the main method for taking a probabilistic step in the environment allowing for action selection based on the learned Q-values.

- *self.params = tf_util.get_trainable_vars("deepq")*

  This is crucial for encapsulating all trainable parameters of the DQN model under the specified "deepq" scope. This allows for efficient access and manipulation during training, evaluation, and when saving or restoring model states.

2. *Learn:*

The learn function provides an essential loop that balances exploration, data collection, and model updates effectively throughout the training. It first prepares logging, callback functions, replay buffer, and exploration schedule, then at each timestep, the agent acts w.r.t. its current policy, and stores every experience. Occasionally, samples from the replay buffer to do an update of Q-values of the model. Hence, the agent can improve its decision-making time over time. Meanwhile, the logger and callback functions log some performance metrics and support other customized behaviors during training. Finally, this function returns the trained model. As this function does the critical job of training the DQN model, it wraps the reinforcement learning logic to perform the optimization of Q-values based on an agent's gathered experiences.

<u>Learn function:</u>

```
def learn(self, total_timesteps, callback=None, log_interval=100, tb_log_name="DQN",
        reset_num_timesteps=True, replay_wrapper=None):
       # Method to start the learning process for the DQN model.
       new_tb_log = self._init_num_timesteps(reset_num_timesteps)  # Initialize logging.
       callback  =  self._init_callback(callback)    #  Initialize  callback  for  additional
behavior during training.
       with SetVerbosity(self.verbose), TensorboardWriter(self.graph, self.tensorboard_log,
tb_log_name, new_tb_log) \ as writer:
        # Set verbosity and prepare TensorBoard writer.
       self._setup_learn()  # Perform any additional setup before learning.
        # Create the replay buffer based on whether prioritized replay is enabled.
        if self.prioritized_replay:
            self.replay_buffer=PrioritizedReplayBuffer(self.buffer_size,
alpha=self.prioritized_replay_alpha)
             # Determine how many iterations to anneal beta for prioritized replay.
            if self.prioritized_replay_beta_iters is None:
                prioritized_replay_beta_iters = total_timesteps
            else:
                prioritized_replay_beta_iters = self.prioritized_replay_beta_iters
                # Create a linear schedule for beta value in prioritized replay.
                self.beta_schedule = LinearSchedule(prioritized_replay_beta_iters,
                                                initial_p=self.prioritized_replay_beta0,
                                                final_p=1.0)
        else:
            self.replay_buffer = ReplayBuffer(self.buffer_size)  # Standard replay buffer.
            self.beta_schedule = None  # No beta schedule for non-prioritized replay.

        # Optional: Wrap the replay buffer if a wrapper is provided.
```

```
        if replay_wrapper is not None:
            assert not self.prioritized_replay, "Prioritized replay buffer is not supported
by HER"
            self.replay_buffer = replay_wrapper(self.replay_buffer)

        # Create a linear schedule for exploration probabilities.
        self.exploration  = LinearSchedule(schedule_timesteps=int(self.exploration_fraction
* total_timesteps),
                                          initial_p=self.exploration_initial_eps,
                                          final_p=self.exploration_final_eps)

        episode_rewards = [0.0]  # Initialize a list to keep track of episode rewards.
        episode_successes = []  # List to track success rates of episodes.

        callback.on_training_start(locals(), globals())  # Notify that training has started.
        callback.on_rollout_start()  # Notify that a rollout has started.

        reset = True  # Flag to indicate if the environment should be reset.
        obs = self.env.reset()  # Reset the environment and get the initial observation.

        for _ in range(total_timesteps):  # Loop through the total timesteps specified.
            # Determine the exploration strategy and action to take.
            kwargs = {}
            if not self.param_noise:
                update_eps = self.exploration.value(self.num_timesteps)
            # Update epsilon for exploration.
                update_param_noise_threshold = 0.  # No parameter noise.
            else:
                update_eps = 0.  # No exploration if using parameter noise.
                # Compute the threshold for KL divergence for parameter noise.
                update_param_noise_threshold = \
                    -np.log(1. - self.exploration.value(self.num_timesteps) +
                            self.exploration.value(self.num_timesteps)/
float(self.env.action_space.n))
                kwargs['reset'] = reset  # Pass the reset flag.
                kwargs['update_param_noise_threshold'] = update_param_noise_threshold
 # Pass the noise threshold.
                kwargs['update_param_noise_scale'] = True  # Indicate that noise scale should
be updated.

            with  self.sess.as_default():    #  Set  the  default  session  for  TensorFlow
operations.
                action = self.act(np.array(obs)[None], update_eps=update_eps, **kwargs)[0]
# Choose an action.

            env_action = action  # Action to be taken in the environment.
            reset = False  # No longer resetting after the first action.
            new_obs, rew, done, info = self.env.step(env_action)
        # Take action in the environment and receive feedback.

            self.num_timesteps += 1  # Increment the total number of timesteps taken.

            # Update callback for additional logic, and stop if requested.
            callback.update_locals(locals())
            if callback.on_step() is False:
                break
            # Store the transition in the replay buffer.
            self.replay_buffer_add(obs_, action, reward_, new_obs_, done, info)

            obs = new_obs  # Update current observation to the new observation.

            if done:  # Check if the episode has ended.
                obs = self.env.reset()  # Reset the environment for a new episode.
                episode_rewards.append(0.0)  # Start tracking rewards for the new episode.
                reset = True  # Reset flag for the new episode.

            # Check if we can sample from the replay buffer.
            can_sample = self.replay_buffer.can_sample(self.batch_size)
            # Train the model if enough samples are available and training conditions are
met.
```

```
            if can_sample and self.num_timesteps > self.learning_starts and
self.num_timesteps % self.train_freq == 0:
                callback.on_rollout_end()  # Notify that the rollout has ended.
                # Sample a batch from the replay buffer.
                if self.prioritized_replay:
                    assert self.beta_schedule is not None, \
                           "BUG: should be LinearSchedule when self.prioritized_replay True"
                    experience = self.replay_buffer.sample(self.batch_size,

beta=self.beta_schedule.value(self.num_timesteps),
                      env=self._vec_normalize_env)
                (obses_t, actions, rewards, obses_tp1, dones, weights, batch_idxes) =
experience
                else:
       obses_t,actions,rewards,obses_tp1,dones=self.replay_buffer.sample(self.batch_size,
env=self._vec_normalize_env)
                    weights, batch_idxes = np.ones_like(rewards),None
   # Default weights for sampling.
     # Perform a training step to minimize the loss.
                if writer is not None:  # If logging is enabled.
                    # Run loss backpropagation with summary.
                    if (1 + self.num_timesteps) % 100 == 0:
                        run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
# Enable full tracing.
                        run_metadata = tf.RunMetadata()  # Prepare metadata for logging.
                        summary, td_errors = self._train_step(obses_t, actions, rewards,
obses_tp1, obses_tp1,dones, weights, sess=self.sess, options=run_options,
                                       run_metadata=run_metadata)
                        writer.add_run_metadata(run_metadata,'step%d' % self.num_timesteps)
# Log metadata.
                        writer.add_summary(summary, self.num_timesteps)  # Log summary.
                    else:
                        td_errors = self._train_step(obses_t, actions, rewards, obses_tp1,
obses_tp1, dones, weights, sess=self.sess)  # Run training step.
                if self.prioritized_replay:  # Update priorities in prioritized replay.
                    self.replay_buffer.update_priorities(batch_idxes, td_errors)
            if done:  # Update the episode success tracker if the episode has ended.
                episode_successes.append(info.get('is_success', 0))  # Track success based
on info from the environment.
                # Log the reward for the completed episode.
                self.logger.record('rollout/ep_rew', episode_rewards[-1])   #Record the
episode reward.
                self.logger.record('rollout/ep_len', len(episode_rewards) - 1)  # Record the
episode length.
                if len(episode_successes) > 0:
                    self.logger.record('rollout/ep_success', np.mean(episode_successes))  #
Log success rate.
                self.logger.dump(step=self.num_timesteps)  # Dump logs to file.
                callback.on_rollout_start()  # Notify that a new rollout has started.

    return self  # Return the trained model.
```

- Logging and Callback Initialization
  Objective: Log, track, and implement callback interfaces with which external code can communicate with the training loop.
  _init_num_timesteps - initializes the counter for training steps from the argument reset_num_timesteps.
  _init_callback - creates the callback object. It mainly maintains additional custom logic running periodically during train, such as logging performance metrics or early stopping.

- Replay Buffer Creation
  Purpose: The replay buffer stores the agent's past experiences in the tuple form of state, action, reward, next state, and done signal. These experiences enable the model to learn from some past interactions, not entirely reliant on the most recent experience, hence promoting more stable learning.

If prioritized_replay is enabled, then experiences with higher TD (Temporal Difference) errors are sampled more frequently. TD error signifies the difference between the predicted Q-value and the actual Q-value.

A schedule (beta_schedule) is created for beta that balances how much the prioritized experiences, over time, affect how much it learns. If prioritized_replay is off, a standard replay buffer is used.

- <u>Exploration Strategy Epsilon-Greedy Purpose</u>

  The agent tries to explore the environment from time to time by taking random actions instead of always taking the best-known action.

  A LinearSchedule decreases the probability of taking random actions over time, where the probability (epsilon) decreases linearly. An agent starts by exploring more and then it exploits more as it "learns" the environment.

- <u>Main Training Loop</u>

  Objective: To go through each timestep, simulate actions, store experiences, and occasionally update the model.

  Observation and Action: The agent observes the current observation, obs, decides on an action by performing epsilon-greedy, and then takes this action in the environment. This causes a new state of the environment, reward, and whether it was done or not.

  Replay Buffer Update: Add experience tuple (state, action, reward, next state, done) to the replay buffer.

  Environment Reset: If an episode is over (done is True), the environment resets and an episode begin anew. Rewards for an episode are recorded for tracking.

- <u>Model Training Step</u>

  Objective: Update Q-values for the model by sampling from the replay buffer to enable the agent to make decisions to improve its previous experiences.

  Sampling and Training: If the replay buffer has enough samples, can_sample is True, and the model has eclipsed some threshold for learning_starts, it samples a batch of experiences to train on.

  Compute Loss and Backpropagate: The function _train_step computes Q-learning loss and backpropagates it through the model by updating its weights.

  Prioritized Replay Updates: For prioritized replay, experiences with higher TD errors come up with an increased priority via updating the priorities in the replay buffer.

- <u>Logging and Callback Management</u>

  Purpose: to keep track of the training progress and may call the callback object at critical points.

  Log metrics after each episode: Record rewards, episode lengths and - if applicable - episode success rates.

  Dump logs: Dumping collected metrics into a file periodically.

  Callback notifications: Notify the callback at the end of a rollout and this can implement custom behavior it wants to (e.g.: save checkpoints).

All these steps ensure that the DQN is ready to learn from experiences in the environment effectively.