ISPR Project - Reinforcement Learning

Maze solving via Reinforcement Learning Algorithms

Paolo Fasano, p.fasano1@studenti.unipi.it

Problem setting - The environment

Problem statement: The aim of the agent is to navigate a maze and reach the exit before zeroing the score

The environment used in this task was custom made. It was created so that any labyrinth of size nxm (with n < m) could be automatically generated with different parameters to decide the complexity of the maze

What the agent receives as a environment

```
0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0,1,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0
0,1,1,1,1,1,1,0,0,1,0,0,1,1,1,1,0,0,0,0
0,0,0,1,0,1,0,0,0,1,0,0,1,0,0,0,0,0,0,0
0,0,0,1,1,1,1,1,1,1,0,1,1,1,1,1,0,0,0,0
0,0,0,0,0,1,0,0,0,1,1,0,1,0,0,0,0,0,0,0
0,0,0,0,0,1,1,0,0,0,1,1,1,1,1,0,0,0,1,0
0,0,1,1,1,1,0,0,0,0,0,0,1,0,0,0,0,0,1,0
0,0,1,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,1,0
0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,4
```

Visual representation of the environment

Problem setting - The actions and rewards

Action	Reward	Reward on backtrack	Reward on action fail
up	-0.05	-0.25	-0.75
down	-0.05	-0.25	-0.75
left	-0.05	-0.25	-0.75
right	-0.05	-0.25	-0.75

If the agent reaches the finishing state (win position) it will gain +1 point

An action is considered failed if the state after the action is the same

Dynamic Programming - Value Iteration

Problem: find the optimal policy
Solution: using synchronous backups

1. At each iteration k+1
2. For each state s existing in S
3. Update $v_{k+1}(s)$ from $v_k(s')$

```
while delta > theta:
    delta = 0
    for row in range(self.size_x):
        for col in range(self.size_y):
            state = [row, col]
            if state in self.path:

                old_value = state_values[state[0],state[1]]
                q_max = float("-inf")

                a = 0
                for action in self.actions:
                    next_state, reward = utils.action_value_function(state, action, self.finish_coord)
                    if reward > 0: done = True
                    value = reward + self.gamma * state_values[next_state[0],next_state[1]]
                    # Update the maximum Q-value and corresponding action probabilities
                    if value > q_max:
                        q_max = value
                        action_probs = np.zeros(len(self.actions))
                        action_probs[a] = 1

                    a+=1

                # Update the state value with the maximum Q-value
                state_values[state[0],state[1]] = q_max

                state_id = utils.find_index_of_coordinate(self.path, state)
                policy[state_id] = action_probs

                # Update the delta with the maximum difference in state values
                delta = max(delta, abs(old_value - state_values[state[0],state[1]]))
```

Dynamic Programming - Value Iteration

Problem: find the optimal policy
Solution: using synchronous backups

1. At each iteration k+1
2. For each state s existing in S
3. Update $v_{k+1}(s)$ from $v_k(s')$

```
while delta > theta:
    delta = 0
    for row in range(self.size_x):
        for col in range(self.size_y):
            state = [row, col]
            if state in self.path:

                old_value = state_values[state[0],state[1]]
                q_max = float("-inf")

                a = 0
                for action in self.actions:
                    next_state, reward = utils.action_value_function(state, action, self.finish_coord)
                    if reward > 0: done = True
                    value = reward + self.gamma * state_values[next_state[0],next_state[1]]
                    # Update the maximum Q-value and corresponding action probabilities
                    if value > q_max:
                        q_max = value
                        action_probs = np.zeros(len(self.actions))
                        action_probs[a] = 1

                    a+=1

                # Update the state value with the maximum Q-value
                state_values[state[0],state[1]] = q_max

                state_id = utils.find_index_of_coordinate(self.path, state)
                policy[state_id] = action_probs

                # Update the delta with the maximum difference in state values
                delta = max(delta, abs(old_value - state_values[state[0],state[1]]))
```

If the difference in value, between old state values and new one, is smaller than a given theta than we stop the training

Dynamic Programming - Value Iteration

Problem: find the optimal policy
Solution: using synchronous backups

1. At each iteration k+1
2. For each state s existing in S
3. Update $v_{k+1}(s)$ from $v_k(s')$

```
while delta > theta:
    delta = 0
    for row in range(self.size_x):
        for col in range(self.size_y):
            state = [row, col]
            if state in self.path:

                old_value = state_values[state[0],state[1]]
                q_max = float("-inf")

                a = 0
                for action in self.actions:
                    next_state, reward = utils.action_value_function(state, action, self.finish_coord)
                    if reward > 0: done = True
                    value = reward + self.gamma * state_values[next_state[0],next_state[1]]
                    # Update the maximum Q-value and corresponding action probabilities
                    if value > q_max:
                        q_max = value
                        action_probs = np.zeros(len(self.actions))
                        action_probs[a] = 1

                    a+=1

                # Update the state value with the maximum Q-value
                state_values[state[0],state[1]] = q_max

                state_id = utils.find_index_of_coordinate(self.path, state)
                policy[state_id] = action_probs

                # Update the delta with the maximum difference in state values
                delta = max(delta, abs(old_value - state_values[state[0],state[1]]))
```

Dynamic Programming - Value Iteration

Problem: find the optimal policy
Solution: using synchronous backups

1. At each iteration k+1
2. For each state s existing in S
3. Update $v_{k+1}(s)$ from $v_k(s')$

```
while delta > theta:
    delta = 0
    for row in range(self.size_x):
        for col in range(self.size_y):
            state = [row, col]
            if state in self.path:

                old_value = state_values[state[0],state[1]]
                q_max = float("-inf")

                a = 0
                for action in self.actions:
                    next_state, reward = utils.action_value_function(state, action, self.finish_coord)
                    if reward > 0: done = True
                    value = reward + self.gamma * state_values[next_state[0],next_state[1]]
                    # Update the maximum Q-value and corresponding action probabilities
                    if value > q_max:
                        q_max = value
                        action_probs = np.zeros(len(self.actions))
                        action_probs[a] = 1

                    a+=1

                # Update the state value with the maximum Q-value
                state_values[state[0],state[1]] = q_max

                state_id = utils.find_index_of_coordinate(self.path, state)
                policy[state_id] = action_probs

                # Update the delta with the maximum difference in state values
                delta = max(delta, abs(old_value - state_values[state[0],state[1]]))
```

Dynamic Programming - Value Iteration

$$v_{k+1} = \max_{a \in \mathcal{A}} \left(\mathcal{R}^a + \gamma \mathbf{P}^a v_k \right)$$

```
for action in self.actions:
    next_state, reward = utils.action_value_function(state, action, self.finish_coord)
    if reward > 0: done = True
    value = reward + self.gamma * state_values[next_state[0],next_state[1]]
    # Update the maximum Q-value and corresponding action probabilities
    if value > q_max:
        q_max = value
        action_probs = np.zeros(len(self.actions))
        action_probs[a] = 1
```

Dynamic Programming - Value Iteration

$$v_{k+1} = \max_{a \in \mathcal{A}} (\mathcal{R}^a + \gamma \mathcal{P}^a v_k)$$

```
for action in self.actions:
    next_state, reward = utils.action_value_function(state, action, self.finish_coord)
    if reward > 0: done = True
    value = reward + self.gamma * state_values[next_state[0],next_state[1]]
    # Update the maximum Q-value and corresponding action probabilities
    if value > q_max:
        q_max = value
        action_probs = np.zeros(len(self.actions))
        action_probs[a] = 1
```

Dynamic Programming - Value Iteration

$$v_{k+1} = \max_{a \in \mathcal{A}} (\mathcal{R}^a + \gamma \mathcal{P}^a v_k)$$

```
for action in self.actions:
    next_state, reward = utils.action_value_function(state, action, self.finish_coord)
    if reward > 0: done = True
    value = reward + self.gamma * state_values[next_state[0],next_state[1]]
    # Update the maximum Q-value and corresponding action probabilities
    if value > q_max:
        q_max = value
        action_probs = np.zeros(len(self.actions))
        action_probs[a] = 1
```

Dynamic Programming - Value Iteration

$$v_{k+1} = \max_{a \in \mathcal{A}} (\mathcal{R}^a + \gamma \mathbf{P}^a v_k)$$

```
for action in self.actions:
    next_state, reward = utils.action_value_function(state, action, self.finish_coord)
    if reward > 0: done = True
    value = reward + self.gamma * state_values[next_state[0],next_state[1]]
    # Update the maximum Q-value and corresponding action probabilities
    if value > q_max:
        q_max = value
        action_probs = np.zeros(len(self.actions))
        action_probs[a] = 1
```

Dynamic Programming - Value Iteration

$$v_{k+1} = \max_{a \in \mathcal{A}} (\mathcal{R}^a + \gamma \mathbf{P}^a v_k)$$

```
for action in self.actions:
    next_state, reward = utils.action_value_function(state, action, self.finish_coord)
    if reward > 0: done = True
    value = reward + self.gamma * state_values[next_state[0],next_state[1]]
    # Update the maximum Q-value and corresponding action probabilities
    if value > q_max:
        q_max = value
        action_probs = np.zeros(len(self.actions))
        action_probs[a] = 1
```

Temporal Difference - SARSA

Problem: learn optimal policy without model knowledge

Solution: iterative update of the action-value function Q(S,A) based on observed transitions and rewards in the environment

1. Every time-step
   a. policy evaluation - TD
   b. policy improvement - epsilon-greedy improvement

```python
def sarsa(self):
    Q = {}
    for s in range(len(self.path)):
        for a in range(len(self.actions)):
            Q[(s,self.actions[a])] = 0.0

    for i in tqdm(range(self.num_iteration), desc="SARSA learning: "):

        utils = common_functions(self.maze)
        # initialize the state,
        state = self.start_coord
        done = False

        # select the action using epsilon-greedy policy
        action = self.epsilon_greedy_policy(Q, state, self.epsilon)
        while True:
            # then we perform the action and move to the next state, and receive the reward
            next_state, reward = utils.action_value_function(state, action, self.finish_coord)
            if reward > 0:
                done = True

            # again, we select the next action using epsilon greedy policy
            next_action = self.epsilon_greedy_policy(Q, next_state, self.epsilon)

            nxt_s = utils.find_index_of_coordinate(self.path, next_state)
            s = utils.find_index_of_coordinate(self.path, state)
            # we calculate the Q value of previous state using our update rule
            Q[(s,action)] += self.alpha * ((reward + self.gamma * Q[(nxt_s,next_action)])-Q[(s,action)])

            # finally we update our state and action with next action and next state
            action = next_action
            state = next_state

            # we will break the loop, if we are at the terminal state of the episode
            if done:
                break
```

Temporal Difference - SARSA

Problem: learn optimal policy without model knowledge

Solution: iterative update of the action-value function Q(S,A) based on observed transitions and rewards in the environment

1. Every time-step
    a. policy evaluation - TD
    b. policy improvement - epsilon-greedy improvement

```python
def sarsa(self):
    Q = {}
    for s in range(len(self.path)):
        for a in range(len(self.actions)):
            Q[(s,self.actions[a])] = 0.0

    for i in tqdm(range(self.num_iteration), desc="SARSA learning: "):

        utils = common_functions(self.maze)
        # initialize the state,
        state = self.start_coord
        done = False

        # select the action using epsilon-greedy policy
        action = self.epsilon_greedy_policy(Q, state, self.epsilon)
        while True:
            # then we perform the action and move to the next state, and receive the reward
            next_state, reward = utils.action_value_function(state, action, self.finish_coord)
            if reward > 0:
                done = True

            # again, we select the next action using epsilon greedy policy
            next_action = self.epsilon_greedy_policy(Q, next_state, self.epsilon)

            nxt_s = utils.find_index_of_coordinate(self.path, next_state)
            s = utils.find_index_of_coordinate(self.path, state)
            # we calculate the Q value of previous state using our update rule
            Q[(s,action)] += self.alpha * ((reward + self.gamma * Q[(nxt_s,next_action)])-Q[(s,action)])

            # finally we update our state and action with next action and next state
            action = next_action
            state = next_state

            # we will break the loop, if we are at the terminal state of the episode
            if done:
                break
```

Temporal Difference - SARSA

Problem: learn optimal policy without model knowledge

Solution: iterative update of the action-value function Q(S,A) based on observed transitions and rewards in the environment

1. Every time-step
   a. policy evaluation - TD
   b. policy improvement - epsilon-greedy improvement

```python
def sarsa(self):
    Q = {}
    for s in range(len(self.path)):
        for a in range(len(self.actions)):
            Q[(s,self.actions[a])] = 0.0

    for i in tqdm(range(self.num_iteration), desc="SARSA learning: "):

        utils = common_functions(self.maze)
        # initialize the state,
        state = self.start_coord
        done = False

        # select the action using epsilon-greedy policy
        action = self.epsilon_greedy_policy(Q, state, self.epsilon)
        while True:
            # then we perform the action and move to the next state, and receive the reward
            next_state, reward = utils.action_value_function(state, action, self.finish_coord)
            if reward > 0:
                done = True

            # again, we select the next action using epsilon greedy policy
            next_action = self.epsilon_greedy_policy(Q, next_state, self.epsilon)

            nxt_s = utils.find_index_of_coordinate(self.path, next_state)
            s = utils.find_index_of_coordinate(self.path, state)
            # we calculate the Q value of previous state using our update rule
            Q[(s,action)] += self.alpha * ((reward + self.gamma * Q[(nxt_s,next_action)])-Q[(s,action)])

            # finally we update our state and action with next action and next state
            action = next_action
            state = next_state

            # we will break the loop, if we are at the terminal state of the episode
            if done:
                break
```

```python
def epsilon_greedy_policy(self, q, state, epsilon):
    """
        This method returns with epsilon probability either a random action or the best action
    """
    utils = common_functions()
    if random.uniform(0,1) < epsilon:
        return self.actions[random.randrange(0,len(self.actions))]
    else:
        s = utils.find_index_of_coordinate(self.path, state)
        index_of_action = max(list(range(len(self.actions))), key = lambda x: q[(s,self.actions[x])])
        return self.actions[index_of_action]
```

Temporal Difference - SARSA

Problem: learn optimal policy without model knowledge

Solution: iterative update of the action-value function Q(S,A) based on observed transitions and rewards in the environment

1. Every time-step
    a. policy evaluation - TD
    b. policy improvement - epsilon-greedy improvement

```python
def sarsa(self):
    Q = {}
    for s in range(len(self.path)):
        for a in range(len(self.actions)):
            Q[(s,self.actions[a])] = 0.0

    for i in tqdm(range(self.num_iteration), desc="SARSA learning: "):

        utils = common_functions(self.maze)
        # initialize the state,
        state = self.start_coord
        done = False

        # select the action using epsilon-greedy policy
        action = self.epsilon_greedy_policy(Q, state, self.epsilon)
        while True:
            # then we perform the action and move to the next state, and receive the reward
            next_state, reward = utils.action_value_function(state, action, self.finish_coord)
            if reward > 0:
                done = True

            # again, we select the next action using epsilon greedy policy
            next_action = self.epsilon_greedy_policy(Q, next_state, self.epsilon)

            nxt_s = utils.find_index_of_coordinate(self.path, next_state)
            s = utils.find_index_of_coordinate(self.path, state)
            # we calculate the Q value of previous state using our update rule
            Q[(s,action)] += self.alpha * ((reward + self.gamma * Q[(nxt_s,next_action)])-Q[(s,action)])

            # finally we update our state and action with next action and next state
            action = next_action
            state = next_state

            # we will break the loop, if we are at the terminal state of the episode
            if done:
                break
```

Temporal Difference - SARSA

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$$

```
Q[(s,action)] += self.alpha * ((reward + self.gamma * Q[(nxt_s,next_action)])-Q[(s,action)])
```

```
for s in range(len(self.path)):
    for a in range(len(self.actions)):
        Q[(s,self.actions[a])] = 0.0
```

initialized at 0 the value is updated for each interaction

hyperparameters

Reward from action a over state s

Temporal Difference - SARSA (λ)

In SARSA-lambda we add an eligibility trace composed by:

- frequency heuristic
- recency heuristic

$$E_0(s) = 0$$

$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t; s)$$

1. for each state we keep an eligibility trace
2. update value V(s) for each state in proportion to TD-error δ and eligibility trace E

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

$$V(s) = V(s) + \alpha\delta_t E_t(s)$$

```python
def sarsa_lambda(self, lambda_par = 3):
    Q = {}
    for s in range(len(self.path)):
        for a in range(len(self.actions)):
            Q[(s,self.actions[a])] = 0.0

    for i in tqdm(range(self.num_iteration), desc="SARSA-lambda learning: "):

        E = {}
        for s in range(len(self.path)):
            for a in range(len(self.actions)):
                E[(s,self.actions[a])] = 0.0

        state = self.start_coord
        action = self.actions[1]
        done = False

        while True:
            utils = common_functions(self.maze)

            next_state, reward = utils.action_value_function(state, action, self.finish_coord)
            if reward > 0:
                done = True

            # again, we select the next action using epsilon greedy policy
            next_action = self.epsilon_greedy_policy(Q, next_state, self.epsilon)

            nxt_s = utils.find_index_of_coordinate(self.path, next_state)
            s = utils.find_index_of_coordinate(self.path, state)

            delta = reward + (self.gamma * Q[(nxt_s,next_action)])-Q[(s,action)]
            E[(s,action)] += 1

            for s in range(len(self.path)):
                for a in range(len(self.actions)):
                    Q[(s,self.actions[a])] += self.alpha * delta * E[(s,self.actions[a])]
                    E[(s,self.actions[a])] = self.gamma * lambda_par * E[(s,self.actions[a])]


            state = next_state
            action = next_action

            if done: break
```

Temporal Difference - SARSA (λ)

In SARSA-lambda we add an eligibility trace composed by:

- frequency heuristic
- recency heuristic

$$E_0(s) = 0$$

$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t; s)$$

1. for each state we keep an eligibility trace
2. update value V(s) for each state in proportion to TD-error δ and eligibility trace E

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

$$V(s) = V(s) + \alpha\delta_t E_t(s)$$

```
def sarsa_lambda(self, lambda_par = 3):
    Q = {}
    for s in range(len(self.path)):
        for a in range(len(self.actions)):
            Q[(s,self.actions[a])] = 0.0

    for i in tqdm(range(self.num_iteration), desc="SARSA-lambda learning: "):

        E = {}
        for s in range(len(self.path)):
            for a in range(len(self.actions)):
                E[(s,self.actions[a])] = 0.0

        state = self.start_coord
        action = self.actions[1]
        done = False

        while True:
            utils = common_functions(self.maze)

            next_state, reward = utils.action_value_function(state, action, self.finish_coord)
            if reward > 0:
                done = True

            # again, we select the next action using epsilon greedy policy
            next_action = self.epsilon_greedy_policy(Q, next_state, self.epsilon)

            nxt_s = utils.find_index_of_coordinate(self.path, next_state)
            s = utils.find_index_of_coordinate(self.path, state)

            delta = reward + (self.gamma * Q[(nxt_s,next_action)])-Q[(s,action)]
            E[(s,action)] += 1

            for s in range(len(self.path)):
                for a in range(len(self.actions)):
                    Q[(s,self.actions[a])] += self.alpha * delta * E[(s,self.actions[a])]
                    E[(s,self.actions[a])] = self.gamma * lambda_par * E[(s,self.actions[a])]


            state = next_state
            action = next_action

            if done: break
```

Value Function Approximation

Problem: if a environment has too many actions/states it creates problems for time and space:
1. too many states/actions to store in memory 2. slow to learn

Solution: estimate the value function via function approximation

$$\hat{v}(s;\mathbf{w}) \approx v_\pi(s) \qquad \hat{q}(s,a;\mathbf{w}) \approx q_\pi(s,a)$$

Approach:

- Incremental methods (SGD, Linear value approximation)
- Batch methods (DQN)

Deep Q-Network

1. initialize a deep network and target network
2. initialize experience replay
3. action are chosen using ϵ-greedy policy (with epsilon decay)
4. store transition in memory replay D
5. sample mini-batch from memory replay D
6. compute Q-learning targets
7. optimise with MSE between Q-network and Q-learning target
8. update target network weights

Deep Q-Network

1. initialize a deep network and target network
2. initialize experience replay
3. action are chosen using ϵ-greedy policy (with epsilon decay)
4. store transition in memory replay D
5. sample mini-batch from memory replay D
6. compute Q-learning targets
7. optimise with MSE between Q-network and Q-learning target
8. update target network weights

size of the state

size of actions

```
class QNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 128)
        self.fc3 = nn.Linear(128, 256)
        self.fc4 = nn.Linear(256, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        return self.fc4(x)
```

```
self.q_network = QNetwork(self.input_size, output_size).to(self.device)
self.target_network = QNetwork(self.input_size, output_size).to(self.device)
self.target_network.load_state_dict(self.q_network.state_dict())
self.optimizer = optim.Adam(self.q_network.parameters(), lr=lr)
```

Deep Q-Network

1. initialize a deep network and target network
2. initialize experience replay
3. action are chosen using ϵ-greedy policy (with epsilon decay)
4. store transition in memory replay D
5. sample mini-batch from memory replay D
6. compute Q-learning targets
7. optimise with MSE between Q-network and Q-learning target
8. update target network weights

```
class replay_buffer():
    def __init__(self, capacity, buffer):
        self.capacity = capacity
        self.buffer = buffer

    def add_experience(self, state, action, reward, next_state, done):
        # we overwrite the oldest examples with the newest
        if len(self.buffer) >= self.capacity:
            self.buffer.pop()
        experience = (state, action, reward, next_state, done)
        self.buffer.append(experience)

    def update_buffer_at(self, index, new_buffer):
        self.buffer[index] = new_buffer

    def sample_batch(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)

        return states, actions, rewards, next_states, dones

    def __len__(self):
        return len(self.buffer)
```

Deep Q-Network

```python
def select_action(self, state):
    if np.random.rand() < self.epsilon:
        return np.random.choice(self.output_size)
    else:
        with torch.no_grad():
            q_values = self.q_network(torch.tensor(state, dtype=torch.float32).to(self.device))
            return torch.argmax(q_values).item()
```

1. initialize a deep network and target network
2. initialize experience replay
3. action are chosen using ϵ-greedy policy (with epsilon decay)
4. store transition in memory replay D
5. sample mini-batch from memory replay D
6. compute Q-learning targets
7. optimise with MSE between Q-network and Q-learning target
8. update target network weights

```python
def update_epsilon(self):
    self.epsilon = max(self.epsilon * self.epsilon_decay, self.epsilon_min)
```

Deep Q-Network

```
agent.replay_buffer.add_experience(state, action_id, reward, next_state, done)
```

1. initialize a deep network and target network
2. initialize experience replay
3. action are chosen using ϵ-greedy policy (with epsilon decay)
4. store transition in memory replay D
5. sample mini-batch from memory replay D
6. compute Q-learning targets
7. optimise with MSE between Q-network and Q-learning target
8. update target network weights

```
# we sample from our buffer
states, actions, rewards, next_states, dones = self.replay_buffer.sample_batch(self.batch_size)
```

Deep Q-Network

1. initialize a deep network and target network
2. initialize experience replay
3. action are chosen using ϵ-greedy policy (with epsilon decay)
4. store transition in memory replay D
5. sample mini-batch from memory replay D
6. compute Q-learning targets
7. optimise with MSE between Q-network and Q-learning target
8. update target network weights

Q-learning targets

$$\left(r + \gamma \max_{a'} Q(s', a'; w_i^-)\right)$$

```
next_q_values = self.target_network(next_states)
q_values = self.q_network(states).gather(1, actions.unsqueeze(-1)).squeeze()
target = rewards + (1 - dones) * self.gamma * next_q_values.max(1)[0]
```

Deep Q-Network

1. initialize a deep network and target network
2. initialize experience replay
3. action are chosen using ϵ-greedy policy (with epsilon decay)
4. store transition in memory replay D
5. sample mini-batch from memory replay D
6. compute Q-learning targets
7. optimise with MSE between Q-network and Q-learning target
8. update target network weights

Q-learning targets

$$\left(r + \gamma \max_{a'} Q(s', a'; w_i^-)\right)$$

```
next_q_values = self.target_network(next_states)
q_values = self.q_network(states).gather(1, actions.unsqueeze(-1)).squeeze()
target = rewards + (1 - dones) * self.gamma * next_q_values.max(1)[0]
```

Deep Q-Network

1. initialize a deep network and target network
2. initialize experience replay
3. action are chosen using ϵ-greedy policy (with epsilon decay)
4. store transition in memory replay D
5. sample mini-batch from memory replay D
6. compute Q-learning targets
7. optimise with MSE between Q-network and Q-learning target
8. update target network weights

```
loss = self.loss_function(q_values, target)
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

Deep Q-Network

1. initialize a deep network and target network
2. initialize experience replay
3. action are chosen using ϵ-greedy policy (with epsilon decay)
4. store transition in memory replay D
5. sample mini-batch from memory replay D
6. compute Q-learning targets
7. optimise with MSE between Q-network and Q-learning target
8. update target network weights

```python
def update_target_network(self):
    # update target network
    for target_network_param, q_network_param in zip(self.target_network.parameters(), self.q_network.parameters()):
        target_network_param.data.copy_(
            self.gamma * q_network_param.data + (1.0 - self.gamma) * target_network_param.data
        )
```

```python
if episode % agent.target_network_frequency == 0 or done:
    print("updating target network")
    agent.update_target_network()
```

Actor-Critic

1. initialize Actor and Critic Network
2. action are chosen using ϵ-greedy policy (with epsilon decay)
3. store transition in memory replay D
4. sample mini-batch from memory replay D
5. compute td-error and td-target
6. compute actor and critic loss
7. optimise and update the network

Actor-Critic

1. initialize Actor and Critic Network
2. action are chosen using ϵ-greedy policy (with epsilon decay)
3. store transition in memory replay D
4. sample mini-batch from memory replay D
5. compute td-error and td-target
6. compute actor and critic loss
7. optimise and update the network

```python
# Define the actor-critic network
class ActorCritic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(ActorCritic, self).__init__()
        self.fc1 = nn.Linear(state_dim, 64)
        self.fc2 = nn.Linear(64, 128)
        self.fc3 = nn.Linear(128, 256)
        self.fc_pi = nn.Linear(256, action_dim)
        self.fc_v = nn.Linear(256, 1)

    def forward(self, state):
        x = torch.relu(self.fc1(state))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        pi = torch.softmax(self.fc_pi(x), dim=-1)
        v = self.fc_v(x)

        return pi, v
```

Final layer for the actor

Final layer for the critic

Actor-Critic

1. initialize Actor and Critic Network
2. action are chosen using ϵ-greedy policy (with epsilon decay)
3. store transition in memory replay D
4. sample mini-batch from memory replay D
5. compute td-error and td-target
6. compute actor and critic loss
7. optimise and update the network

```
if np.random.rand() < epsilon:
    action_id = np.random.choice(self.action_size)
    probs = np.zeros(self.action_size)
    probs[action_id] = 1/self.action_size
    probs = torch.tensor(probs, dtype=torch.float32)
else:
    probs, _ = self.agent(torch.tensor(state, dtype=torch.float32))
    action_id = np.argmax(probs.detach().numpy())
```

```
epsilon = max((self.epsilon_start * self.epsilon_decay**episode), self.epsilon_min)
```

Actor-Critic

1. initialize Actor and Critic Network
2. action are chosen using ϵ-greedy policy (with epsilon decay)
3. store transition in memory replay D
4. sample mini-batch from memory replay D
5. compute td-error and td-target
6. compute actor and critic loss
7. optimise and update the network

```python
if done:
    self.replay_buffer.add_experience(state, action_id, reward, next_state, 1)
else:
    self.replay_buffer.add_experience(state, action_id, reward, next_state, 0)
```

```python
states, actions, rewards, next_states, dones = self.replay_buffer.sample_batch(self.batch_size)
states = torch.tensor(states, dtype=torch.float32)
actions = torch.tensor(actions, dtype=torch.int64)
rewards = torch.tensor(rewards, dtype=torch.float32)
next_states = torch.tensor(next_states, dtype=torch.float32)
dones = torch.tensor(dones, dtype=torch.float32)
```

Actor-Critic

1. initialize Actor and Critic Network
2. action are chosen using ϵ-greedy policy (with epsilon decay)
3. store transition in memory replay D
4. sample mini-batch from memory replay D
5. compute td-error and td-target
6. compute actor and critic loss
7. optimise and update the network

```
# Calculate the TD error
_, next_values = self.agent(next_states)
_, values = self.agent(states)
target_values = rewards + self.gamma * next_values * (1 - dones)
td_error = target_values - values
```

Actor-Critic

1. initialize Actor and Critic Network
2. action are chosen using ϵ-greedy policy (with epsilon decay)
3. store transition in memory replay D
4. sample mini-batch from memory replay D
5. compute td-error and td-target
6. compute actor and critic loss
7. optimise and update the network

```
# Compute losses
log_probs, _ = self.agent(states)
actor_losses = -log_probs.gather(1, actions.view(-1, 1)).squeeze(1) * td_error.detach()
critic_loss = torch.mean(torch.square(td_error))
loss = torch.mean(actor_losses) + critic_loss
```

```
# Update the network
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

Results

Algorithm	Initial score	Resulting score	Iterations
Baseline	26	mean -208.7	100
Value iteration	26	19.40	100
SARSA	26	19.40	2000
SARSA - (λ)	26	19.40	2000
Deep Q-Network	26	19.40	1500
Actor - Critic	26	-25.6	1500 to 5000