# Deep Q-Network (DQN) Implementation Explanation

This document explains the implementation of the Deep Q-Network (DQN) provided in the code. The DQN is an extension of Q-learning that uses a neural network to approximate the Q-function. It is particularly useful when the state space is large or continuous, making tabular Q-learning impractical.

## 1. Neural Network Definition (DQN class)

The DQN class defines the neural network used to approximate the Q-function. It uses an embedding layer for discrete states, followed by configurable hidden layers and an output layer that predicts Q-values for each possible action.

Code snippet:

class DQN(nn.Module):  
 def \_\_init\_\_(self, state\_size, action\_size):  
 super().\_\_init\_\_()  
 self.embedding = nn.Embedding(state\_size, 16)  
  
 layers = []  
 input\_size = 16  
  
 for \_ in range(DQN\_HIDDEN\_LAYERS):  
 layers.append(nn.Linear(input\_size, DQN\_NODES\_PER\_LAYER))  
 layers.append(nn.ReLU())  
 input\_size = DQN\_NODES\_PER\_LAYER  
  
 layers.append(nn.Linear(input\_size, action\_size))  
 self.net = nn.Sequential(\*layers)  
  
 def forward(self, x):  
 x = self.embedding(x)  
 return self.net(x)

## 2. Training Function (train\_dqn)

The train\_dqn function implements the training process of the DQN. It initializes the policy and target networks, sets up the replay buffer, and runs the training loop over episodes.

Key steps:

1. Initialize policy and target networks

2. Run episodes with ε-greedy action selection

3. Store transitions in replay buffer

4. Sample minibatches and perform gradient descent

5. Update target network periodically

6. Decay epsilon over time

Code snippet:

def train\_dqn(device, env, episodes=1000, batch\_size=64, gamma=0.99, epsilon=1.0,   
 min\_epsilon=0.1, epsilon\_decay=0.995, target\_update\_freq=10, grad\_update\_freq=4):  
 n\_states = env.observation\_space.n  
 n\_actions = env.action\_space.n  
 policy\_net = DQN(n\_states, n\_actions).to(device)  
 target\_net = DQN(n\_states, n\_actions).to(device)  
 target\_net.load\_state\_dict(policy\_net.state\_dict())  
 target\_net.eval()  
  
 optimizer = optim.Adam(policy\_net.parameters(), lr=1e-3)  
 criterion = nn.MSELoss()  
 replay\_buffer = deque(maxlen=10\_000)  
 step\_count = 0  
  
 rewards = []  
 epsilon\_history = []  
 losses = []  
  
 for ep in trange(episodes, desc="🏋️ Training DQN"):  
 state, \_ = env.reset()  
 done = False  
 ep\_reward = 0  
 ep\_losses = []  
  
 while not done:  
 step\_count += 1  
 s\_tensor = torch.tensor([state], device=device)  
  
 if random.random() < epsilon:  
 action = env.action\_space.sample()  
 else:  
 with torch.no\_grad():  
 action = policy\_net(s\_tensor).argmax().item()  
  
 next\_state, reward, terminated, truncated, \_ = env.step(action)  
 done = terminated or truncated  
 ep\_reward += reward  
  
 replay\_buffer.append((state, action, reward, next\_state, done))  
 state = next\_state  
  
 if step\_count % grad\_update\_freq == 0 and len(replay\_buffer) >= batch\_size:  
 batch = random.sample(replay\_buffer, batch\_size)  
 states, actions, rewards\_batch, next\_states, dones = zip(\*batch)  
  
 states = torch.tensor(states, device=device)  
 next\_states = torch.tensor(next\_states, device=device)  
 actions = torch.tensor(actions, device=device)  
 rewards\_batch = torch.tensor(rewards\_batch, dtype=torch.float32, device=device)  
 dones = torch.tensor(dones, dtype=torch.bool, device=device)  
  
 q\_values = policy\_net(states).gather(1, actions.unsqueeze(1)).squeeze()  
 with torch.no\_grad():  
 next\_q = target\_net(next\_states).max(1)[0]  
 targets = rewards\_batch + gamma \* next\_q \* (~dones)  
  
 loss = criterion(q\_values, targets)  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
 ep\_losses.append(loss.item())  
  
 rewards.append(ep\_reward)  
 epsilon\_history.append(epsilon)  
 avg\_loss = np.mean(ep\_losses) if ep\_losses else 0  
 losses.append(avg\_loss)  
  
 epsilon = max(min\_epsilon, epsilon \* epsilon\_decay)  
  
 if ep % target\_update\_freq == 0:  
 target\_net.load\_state\_dict(policy\_net.state\_dict())  
  
 return policy\_net, rewards, epsilon\_history, losses