DQN Trading Algorithm: Step-by-Step Execution Flow

Phase 0: Initialization (Happens Once)

Step 0.1: Environment Creation

```python
cfg = EnvConfig(batch_size=256, T=10)
env = Environment(cfg)
```

What happens:

1. Creates 256 parallel trading environments (like 256 traders in parallel universes)
2. Each environment tracks:
   - `self.price`: [256, 10] - last 10 prices for each environment
   - `self.pos`: [256, 10] - last 10 positions for each environment
3. Generates initial price history by simulating mean-reverting process:
   - Start at S_m = 100
   - For 10 time steps: `S(t+1) = S(t) + σ*ε - κ*(S(t) - S_m)`
   - Clamp between 80-120
4. Creates initial state `self.x` [256, 20] by concatenating normalized prices + positions

Step 0.2: Agent Creation

```python
agent = Agent(gamma=0.99, epsilon=1.0, lr=3e-4, input_dims=20, ...)
```

What happens:

1. Creates neural network `Q_eval`:
   - Input: 20 dimensions (10 prices + 10 positions)
   - Hidden: 256 → 256 neurons with ReLU
   - Output: 3 Q-values (one for each action: sell, hold, buy)
2. Initializes Adam optimizer
3. Allocates replay buffer memory:
   - `state_memory`: [100000, 20] - stores past states

- `action_memory`: [100000] - stores past actions
- `reward_memory`: [100000] - stores past rewards
- `new_state_memory`: [100000, 20] - stores next states
- `terminal_memory`: [100000] - stores done flags

4. Sets `mem_cntr = 0` (no memories yet)
5. Sets `epsilon = 1.0` (100% exploration at start)

Phase 1: Training Loop (Repeats for 500 Episodes)

EPISODE 1, STEP 1

Step 1.1: Reset Environment

```python
state = env.reset()  # Returns [256, 20]
```

- Regenerates fresh price histories for all 256 environments
- All positions set to 0
- Returns normalized state tensor

Step 1.2: Choose Actions (Epsilon-Greedy)

```python
actions = agent.choose_action(state)  # Returns [256] numpy array
```

Detailed process:

1. Convert state to torch tensor and move to device
2. **Forward pass through neural network:**

```
Input [256, 20] → FC1 [256, 256] → ReLU → FC2 [256, 256] → ReLU → FC3 [256, 3]
Output: Q-values [256, 3]
```

3. **Per-row epsilon-greedy:**
   - For each of 256 environments independently:
     - Generate random number r ∈ [0,1]
     - If r < epsilon (currently 1.0):

        - Choose random action from {0, 1, 2}
    - Else:
        - Choose action = argmax(Q-values) for that row
- Initially epsilon=1.0, so ALL 256 environments choose random actions

4. Return numpy array [256] with action indices

Example: `actions = [1, 0, 2, 1, 0, ...]` (256 actions total)

Step 1.3: Environment Step

```python
next_state, rewards, dones, _ = env.step(actions)
```

Detailed process for EACH of 256 environments:

1. **Map action index to actual position:**

```python
a_t = self._idx2val[actions]  # [256] in {-1, 0, +1}
```

- action 0 → position -1 (sell)
- action 1 → position 0 (hold)
- action 2 → position +1 (buy)

2. **Get current state:**

```python
prev_price = self.price[:, -1]      # [256] - last price
prev_pos = self.pos[:, -1]         # [256] - last position
```

3. **Generate next price (mean-reverting process):**

```python
ε ~ N(0,1)  # Random shock for each environment
next_price = prev_price + σ*ε - κ*(prev_price - S_m)
next_price = clamp(round(next_price), 80, 120)
```

- Each environment gets different random shock
- Prices independently evolve

4. **Calculate reward for each environment:**

```python
dS = next_price - prev_price

pnl = a_t * dS                # P&L from position
friction = β * |a_t - prev_pos|  # Cost of changing position
vol_penalty = γ * σ² * a_t²      # Risk penalty

reward = pnl - friction - vol_penalty
```

Example for one environment:

- prev_price = 98, next_price = 100 → dS = +2
- a_t = +1 (buy), prev_pos = 0
- pnl = (+1) * (+2) = +2.0
- friction = 0.1 * |1 - 0| = 0.1
- vol_penalty = 0.01 * 4 * 1 = 0.04
- **reward = 2.0 - 0.1 - 0.04 = 1.86** ✓ Good reward!

5. **Update price and position histories (rolling window):**

```python
self.price = [price[:, 1:], next_price]  # Drop oldest, append newest
self.pos = [pos[:, 1:], a_t]             # Drop oldest, append newest
```

- Each is [256, 10] - sliding window of last 10 values

6. **Update state representation:**

```python
z_price = (self.price - 100) / 2.0  # Normalize
self.x = concat([z_price, self.pos], dim=1)  # [256, 20]
```

7. **Return:**
   - next_state: [256, 20]
   - rewards: [256] floats
   - dones: [256] all False (we don't have terminal states)

Step 1.4: Store Transitions in Replay Buffer

```python
agent.store_transition(state, actions, rewards, next_state, dones)
```

What happens:

1. Convert all torch tensors to numpy arrays
2. Calculate indices for circular buffer:

```python
 indices = [0, 1, 2, ..., 255]  # First 256 slots
```

3. Store in replay buffer:

```python
 state_memory[0:256] = state         # [256, 20]
 action_memory[0:256] = actions       # [256]
 reward_memory[0:256] = rewards       # [256]
 new_state_memory[0:256] = next_state # [256, 20]
 terminal_memory[0:256] = dones       # [256]
```

4. Increment memory counter: `mem_cntr = 256`

Step 1.5: Learn (First Call - SKIPPED!)

```python
agent.learn()
```

What happens:

```python
if self.mem_cntr < self.batch_size:  # 256 < 128? No, 256 >= 128
    return  # Actually will NOT return, will try to learn
```

Wait, we have 256 memories and batch_size=128, so we DO learn!

Learning process (FIRST TIME):

1. **Sample random batch from replay buffer:**

```python
batch = random.choice([0, 1, ..., 255], size=128, replace=False)
# Example: batch = [45, 203, 12, 189, ...]
```

2. **Load batch from memory:**

```python
states = state_memory[batch]        # [128, 20]
actions = action_memory[batch]      # [128]
rewards = reward_memory[batch]      # [128]
next_states = new_state_memory[batch] # [128, 20]
dones = terminal_memory[batch]      # [128]
```

3. **Forward pass to get Q-values for current states:**

```python
q_values = Q_eval(states)  # [128, 3]
# Example row: [0.23, -0.15, 0.08] (random initially)
```

4. **Select Q-values for actions that were actually taken:**

```python
row_idx = [0, 1, 2, ..., 127]
q_eval = q_values[row_idx, actions]  # [128]
# If actions[0]=1, picks q_values[0, 1] = -0.15
```

5. **Compute target Q-values (what we WANT Q to predict):**

```python
with torch.no_grad():  # Don't backprop through target
    q_next_all = Q_eval(next_states)    # [128, 3]
    q_next_max = max(q_next_all, dim=1) # [128] - best action

    # Bellman equation:
    q_target = rewards + γ * (1 - dones) * q_next_max
```

Example for one sample:

- reward = 1.86
- q_next_max = 0.31 (best Q-value for next state)

- q_target = 1.86 + 0.99 * 0.31 = 2.17

6. **Compute loss (Mean Squared Error):**

```python
loss = MSE(q_eval, q_target)
# Example: MSE([-0.15, 0.23, ...], [2.17, 1.45, ...])
```

7. **Backpropagation:**

```python
loss.backward()  # Compute gradients ∂loss/∂weights
optimizer.step() # Update weights: w = w - lr * ∇loss
```

What this does:

- Adjusts all 20*256 + 256*256 + 256*3 ≈ 70,000 parameters
- Makes Q_eval predict values closer to q_target
- Agent learns: "When I see this state and take this action, I should expect ~2.17 reward"

8. **Decay epsilon:**

```python
epsilon = max(0.05, 1.0 - 0.00001)
epsilon = 0.99999  # Barely changed
```

Step 1.6: Update State

```python
state = next_state  # [256, 20]
```

EPISODE 1, STEPS 2-200

Repeat steps 1.2-1.6 for 199 more times:

- Each step, 256 new transitions are stored → memory fills up
- After 200 steps: `mem_cntr = 256 * 200 = 51,200` memories
- Agent has learned 200 times
- Epsilon decayed to: `1.0 - 200*0.00001 = 0.998`

- Q-network weights updated 200 times
- Agent slowly learns patterns: "If price is high and falling, holding gives better Q-value than buying"

EPISODES 2-500

Same process repeats:

Episode 10 (2,000 steps total):

- `mem_cntr = 512,000` but buffer only holds 100,000 → oldest memories overwritten
- epsilon = 0.98 → 98% exploration, 2% exploitation
- Q-values starting to make sense: Q(good_state, good_action) > Q(bad_state, bad_action)

Episode 100 (20,000 steps):

- epsilon = 0.80 → 80% exploration, 20% exploitation
- Agent occasionally picks learned good actions
- Q-network has seen millions of gradient updates

Episode 250 (50,000 steps):

- epsilon = 0.50 → balanced exploration/exploitation
- Agent frequently picks learned actions
- Rewards starting to improve: was -0.10 avg, now -0.05 avg

Episode 400 (80,000 steps):

- epsilon = 0.20 → mostly exploitation
- Agent mostly follows learned policy
- Rewards positive: +0.02 avg

Episode 500 (100,000 steps):

- epsilon = 0.05 → 95% exploitation, 5% exploration
- Agent has converged to learned policy
- Final avg reward: +0.1265 per step

Phase 2: Evaluation (Pure Exploitation)

Step E.1: Disable Exploration

```python
agent.epsilon = 0.0  # GREEDY ONLY
```

Step E.2: Run 20 Evaluation Episodes

For each episode:

1. **Reset environment**

```python
   state = env.reset()
```

2. **Run 100 steps with pure exploitation:**

```python
   for step in range(100):
       actions = agent.choose_action(state)
       # With epsilon=0, ALWAYS picks argmax(Q-values)
       # No randomness!

       next_state, rewards, dones, _ = env.step(actions)
       total_reward += rewards.mean()
       state = next_state
```

3. **Track total reward for episode**
   - Example: Episode 1 = 16.61, Episode 2 = 16.34, ...

Step E.3: Compute Statistics

```python
Average: 17.04 ± 0.54
Min: 16.33
Max: 18.06
```

What the Agent Actually Learned

By the end, the neural network has learned a **value function** Q(s, a):

Input: [normalized prices for last 10 steps, positions for last 10 steps]

Output: [Q(s, sell), Q(s, hold), Q(s, buy)]

Example learned behaviors:

1. **Mean reversion:**
   - If price = 115 (high) and falling → Q(s, sell) = 2.5 (highest)
   - If price = 85 (low) and rising → Q(s, buy) = 2.8 (highest)
2. **Friction avoidance:**
   - If current position = +1 and Q(buy) only slightly better → Q(hold) chosen (avoids cost)
3. **Risk management:**
   - Large positions penalized → agent prefers smaller positions unless strong signal

The agent essentially learned to be a **statistical arbitrage trader** that:

- Buys when prices are low relative to mean
- Sells when prices are high relative to mean
- Avoids excessive trading (friction costs)
- Manages position risk

Key Insight: Bootstrapping

The magic of Q-learning is **bootstrapping** - the agent uses its own estimates to improve its own estimates:

1. **Initially:** Q-values are random → agent explores randomly
2. **Early learning:** Agent discovers "buying at 85 gives +2 reward" → updates Q(s, buy) toward 2
3. **Later:** Agent sees state that leads to state where Q(s, buy)=2 → updates previous state to 2 + γ*2 = 3.98
4. **Eventually:** Chain of Q-values propagates backward, agent learns long-term consequences

It's literally "pulling itself up by its bootstraps" - using estimates to improve estimates!