DeepSeek Highlights:

## Changes from ACER to DQN:

Replay Memory: Changed from EpisodicReplayMemory to a simpler ReplayMemory that stores individual transitions.

Network Architecture: Replaced the actor-critic network with a simple Q-network (DQN).

Loss Calculation: Implemented the DQN loss function, which uses the Bellman equation to compute the loss.

Target Network: Added a target network to stabilize training.

Action Selection: Changed from sampling actions using a policy to selecting actions using an epsilon-greedy strategy (not explicitly shown here, but can be added).

Changes from ACER to TD3:  
Actor Network: The actor network will output a probability distribution over discrete actions (similar to ACER), but we will use a Gumbel-Softmax trick to sample actions during training.

Critic Networks: TD3 uses two Q-networks (critics) and takes the minimum of the two Q-values to reduce overestimation bias.

Target Networks: TD3 uses target networks for both the actor and critics, which are updated using polyak averaging.

Delayed Policy Updates: TD3 updates the policy (actor) less frequently than the Q-functions (critics).

Exploration Noise: Since the action space is discrete, we will use a simple epsilon-greedy strategy for exploration.

## Tricks Used

In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything.

If you only use the very-most recent data, you will overfit to that and things will break;

if you use too much experience, you may slow down your learning. This may take some tuning to get right.

SpinningUp DDPG implementation uses a trick to improve exploration at the start of training.

For a fixed number of steps at the beginning (set with the start\_steps keyword argument),

the agent takes actions which are sampled from a uniform random distribution over valid actions.

After that, it returns to normal DDPG exploration.

## Novelty Features

To facilitate getting higher-quality training data, you may reduce the scale of the noise over the course of training.

(We do not do this in SpinningUp implementation, and keep noise scale fixed throughout.)

# Future Considerations

## DQN implementation

1. Hyperparameter Tuning:

Learning Rate: Try different learning rates, such as 1e-3, 1e-4, or 1e-5.

Batch Size: Experiment with larger or smaller batch sizes.

Epsilon Decay: Adjust epsilon\_start, epsilon\_end, and epsilon\_decay to control exploration.

2. Network Architecture:

Hidden Layers: Experiment with different numbers of hidden layers and units. For example, try adding more layers or increasing the number of units per layer.

Activation Functions: Consider using other activation functions like LeakyReLU.

3. Replay Buffer:

Prioritized Experience Replay: Implement prioritized experience replay to sample more important transitions more frequently.

Buffer Size: Ensure the buffer size is large enough to capture diverse experiences.

4. Target Network Update:

Soft Updates: Instead of hard updates, use soft updates with a small tau value to slowly blend the target network with the policy network.

5. Reward Normalization:

Normalize rewards to stabilize training, especially if rewards have a wide range.

6. Additional Techniques:

Double DQN: Use Double DQN to reduce overestimation bias by decoupling action selection and evaluation.

Dueling DQN: Implement a dueling architecture to separately estimate state value and advantage.

7. Environment-Specific Adjustments:

State Representation: Ensure the state representation captures all necessary information.

Reward Function: Verify that the reward function aligns with the desired behavior.

## The new DDPG Approach

1. Discrete Action Handling
   * DDPG is designed for continuous action spaces. You've used Gumbel-Softmax to generate discrete actions, which is a valid adaptation.
2. Replay Buffer (replay\_memory class)
   * The replay buffer seems non-optimized. Using NumPy for such operations can be slow for larger buffers. Consider using collections like deque or PyTorch tensors for better performance.
   * store method's logic has unnecessary complexity. You could simplify how transitions are added or replaced in the buffer.
3. Exploration vs Exploitation
   * Epsilon-greedy (eps) is used for exploration. While simple, it might not work as effectively for DDPG since the noise-based exploration (e.g., Ornstein-Uhlenbeck noise) is generally more aligned with this algorithm.
   * Consider annealing eps over episodes to encourage more exploitation as training progresses.
4. Performance Metrics
   * Running just one test episode per evaluation step may lead to noisy observations. Average over multiple test episodes for a better estimate.
5. Soft Update
   * The soft update mechanism is correct. However, the frequent updates (once per step) might slow training and increase computational cost. Consider updating less frequently (e.g., once per episode or batch).
   * You're manually iterating over batch elements, which can be replaced with a vectorized approach for efficiency.
6. Code Hygiene

## TD3 Adaptation

Exploration Strategy:

* Current Issue: You're using Gaussian noise (GaussianExploration) to explore, but it may not be enough for stable exploration in a TD3 setup, especially when the environment is complex or has sparse rewards.
* Solution: You could try adjusting the noise parameters (e.g., higher max\_sigma or adjust min\_sigma), or explore other exploration techniques like Ornstein-Uhlenbeck noise, which is commonly used in TD3.

Target Networks Update Frequency:

* Current Issue: You're updating the target networks at every step (via soft\_update), which might be too frequent and cause instability.
* Solution: Update the target networks less frequently (e.g., every 2 to 5 iterations) to allow the Q-values to stabilize before transferring information to the target networks.

Replay Buffer Size:

* Current Issue: Your replay buffer is large (1000000), but it may not be fully utilized depending on how quickly new experiences are added. Sometimes, larger replay buffers may cause slower learning.
* Solution: Consider reducing the replay buffer size, or ensure the buffer has a good mix of exploration and exploitation. A common approach is to sample experiences more efficiently.
  + You can also prioritize experiences based on the TD-error to enhance learning from important transitions, which is implemented in Prioritized Experience Replay.

Gradient Clipping:

* Current Issue: Unstable gradients may lead to large weight updates, which can cause divergence.
* Solution: Apply gradient clipping to stabilize the training process, especially for the policy and value networks. You can clip gradients during the backward pass:

Try code:

torch.nn.utils.clip\_grad\_norm\_(policy\_net.parameters(), max\_norm=0.5)

Learning Rate:

* Current Issue: The learning rate for the actor (lr\_actor = 0.0003) and critic (lr\_critic = 0.001) might not be optimal for your specific problem.
* Solution: Try experimenting with different learning rates, possibly using learning rate schedules (decay over time) to allow the model to learn more effectively. Using a learning rate scheduler like torch.optim.lr\_scheduler can help fine-tune the learning process.

Batch Normalization:

* Current Issue: The model may not be sufficiently regularized or normalized, leading to instability.
* Solution: Adding batch normalization layers or layer normalization can improve the stability of the learning process, especially when training deep networks.

-------------------------------------------------

1. Lowering the Exploration Noise:

* Noise Decay: You are using Ornstein-Uhlenbeck noise, which has a decay rate over time. If the noise decay period is too long, the exploration may remain too high for too long, leading to erratic behavior.
* Impact of High Exploration: If the noise is too large, the agent might not learn meaningful patterns in the environment, as the actions taken are too random. Lowering the standard deviation (noise\_std) or the clipping value (noise\_clip) may encourage more deterministic action selection, enabling faster convergence.

2. Hyperparameters to Check:

* Learning Rates: If the learning rates for the actor (lr\_actor) or critic (lr\_critic) are too high, it can cause instability in training. In many cases, lowering the learning rate helps to reduce the oscillations and improve convergence.
* Replay Buffer and Priority Sampling: Make sure your replay buffer is appropriately sized and the prioritization is working as intended. Too much noise in the prioritized sampling can cause unstable training.
* Target Network Updates: The soft\_tau used for updating the target networks is often a small value (e.g., 1e-2). If this is too high, it could lead to the target networks drifting too far from the main networks, making training unstable.

3. Exploration vs. Exploitation:

* Exploration Decay: Consider introducing a more gradual decay to the exploration noise. You might start with a higher noise for exploration and then gradually decrease it as the agent becomes more confident in its actions.
* Early vs. Late Training: In the beginning, noise should be high enough to encourage exploration, but as the agent progresses, the noise should gradually reduce to allow the learned policy to dominate.

4. Improving Stability:

* Action Normalization: You are using normalized actions; ensure that this normalization process is done correctly for both exploration noise and the network output.
* Critic Network Updates: You are updating both value\_net1 and value\_net2. Make sure both critics are being updated in tandem and there is no significant difference between them, as this could introduce instability.
* Target Network Smoothing: You are doing soft updates to the target networks, which is important for stability. Make sure the soft\_tau value is not too large, as this will cause instability when propagating target values.

5. Considerations:

* If after lowering the exploration noise you still face issues, experiment with different values of gamma (discount factor), batch\_size, and other network architecture parameters like the number of hidden units or layers.
* Make sure your environment is sufficiently challenging but not too complex, as this can affect the learning dynamics.