Policy-Based Deep Reinforcement Learning

Overview

Policy-based Deep Reinforcement Learning is a direct approach to solving RL problems by learning the optimal policy (strategy) without using value functions as intermediaries. Unlike value-based methods like Q-learning, these algorithms directly map states to actions through a probability distribution, making them particularly effective for continuous action spaces and complex environments. The main algorithms include REINFORCE (vanilla policy gradient), PPO (Proximal Policy Optimization), and Actor-Critic methods, each offering different trade-offs between stability and sample efficiency. The key advantage is their ability to learn stochastic policies and handle high-dimensional action spaces, making them ideal for robotics, game playing, and resource management applications, though they typically face challenges with high variance and sample efficiency.

Policy-based methods are a class of algorithms that directly learn the optimal policy without using a value function as an intermediate step. Unlike value-based methods (like Q-learning), these methods directly parameterize the policy and update it through gradient ascent.

Key Concepts

1. Policy Definition

- **Policy ( )**: A function that maps states to actions
- **Deterministic Policy**: (s) → a
- **Stochastic Policy**: (a|s) → [0,1] (probability distribution over actions)

2. Advantages Over Value-Based Methods

- Better suited for continuous action spaces
- Can learn stochastic policies
- More stable learning in many cases
- More effective in high-dimensional action spaces
- Can naturally handle competing objectives

3. Core Algorithms

REINFORCE (Monte Carlo Policy Gradient)

```
# Basic REINFORCE algorithm structure
def REINFORCE(policy_network):
    for episode in episodes:
        states, actions, rewards = collect_trajectory()
        returns = compute_returns(rewards)
        loss = -log_prob(actions) * returns
        policy_network.update(loss)
```

Proximal Policy Optimization (PPO)

- Clips the policy update to prevent too large changes
- More stable training than vanilla policy gradient
- Widely used in modern applications

Actor-Critic Methods

- Combines policy-based and value-based learning
- Actor: Updates policy
- Critic: Estimates value function

- Reduces variance while maintaining bias

Common Challenges and Solutions

1. High Variance

Problems:

- Monte Carlo returns can have high variance
- Slow learning

Solutions:

- Use baseline subtraction
- Implement value function as critic
- Add entropy regularization

2. Sample Efficiency

Improvements:

- Experience replay
- Off-policy learning
- Importance sampling

Applications

1. **Robotics**
   - Continuous control
   - Complex manipulation tasks
2. **Game Playing**
   - Strategy games
   - Real-time decisions
3. **Resource Management**
   - Network routing
   - Power grid optimization

Implementation Tips

1. Network Architecture

```
class PolicyNetwork(nn.Module):
    def __init__(self):
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        action_probs = F.softmax(self.fc2(x), dim=-1)
        return action_probs
```

2. Training Considerations

- Learning rate scheduling
- Gradient clipping
- Proper initialization
- Batch size selection

Recent Developments

1. **Hybrid Approaches**
   - Combining with model-based methods
   - Integration with imitation learning
2. **Multi-Agent Extensions**
   - Cooperative policies
   - Competitive scenarios

Summary

Policy-based methods are powerful tools in deep RL that:

- Directly optimize the policy
- Handle continuous action spaces well
- Work with stochastic policies
- Scale to complex problems

Further Reading

1. "Policy Gradient Methods for Reinforcement Learning with Function Approximation" (Sutton et al.)
2. "Trust Region Policy Optimization" (Schulman et al.)
3. "Proximal Policy Optimization Algorithms" (Schulman et al.)

Question: The neural network that approximates the policy takes the environment state as input. The output layer returns the probability that the agent should select each possible action. Which of the following is a valid activation function for the output layer?

Options:

1. linear (i.e., no activation function)
2. softmax
3. ReLU

Answer: The correct answer is softmax.

Explanation: Softmax is the appropriate activation function for the output layer in policy networks because:

1. It converts raw output scores into probabilities that sum to 1.0
2. Each output node gives a value between 0 and 1, representing the probability of selecting each action
3. It's specifically designed for multi-class probability distributions, which is exactly what we need when selecting between multiple possible actions
4. Unlike linear or ReLU:
   - Linear activation could give invalid probability values (negative or >1)
   - ReLU would only give positive values but wouldn't ensure they sum to 1.0

This matches the requirement of a policy network where we need to output valid probability distributions over the action space.

Gradient Ascent Gradient ascent is similar to gradient descent.

Gradient descent steps in the direction opposite the gradient, since it wants to minimize a function. Gradient ascent is otherwise identical, except we step in the direction of the gradient, to reach the maximum. While we won't cover gradient-based methods in this lesson, you'll explore them later in the course!

Local Minima In the video above, you learned that hill climbing is a relatively simple algorithm that the agent can use to gradually improve the weights in its policy network while interacting with the environment.

Note, however, that it's not guaranteed to always yield the weights of the optimal policy. This is because we can easily get stuck in a local maximum. In this lesson, you'll learn about some policy-based methods that are less prone to this.

Hill Climbing Pseudocode

Here's the exact OCR of the image:

What's the difference between G and J?

You might be wondering: what's the difference between the return that the agent collects in a single episode (G, from the pseudocode above) and the expected return J?

Well … in reinforcement learning, the goal of the agent is to find the value of the policy network weights that maximizes expected return, which we have denoted by J.

In the hill climbing algorithm, the values of are evaluated according to how much return G they collected in a single episode. To see that this might be a little bit strange, note that due to randomness in the environment (and the policy, if it is stochastic), it is highly likely that if we collect a second episode with the same values for , we'll likely get a different value for the return G. Because of this, the (sampled) return G is not a perfect estimate for the expected return J, but it often turns out to be good enough in practice.

Stochastic Policy Search in Reinforcement Learning

Overview

Stochastic policy search is a method in reinforcement learning where the agent learns a probabilistic policy that maps states to a distribution over actions, rather than deterministic action selections.

Key Components

1. Policy Representation

```
class StochasticPolicy(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim),
            nn.Softmax(dim=-1)  # Outputs action probabilities
        )
```

2. Core Concepts

- **Exploration vs Exploitation**: Naturally balances through probability distributions
- **Policy Parameters ( )**: Network weights that define the policy
- **Returns**:
  - G: Single episode return (sampled)
  - J: Expected return (theoretical average)

3. Advantages

1. Better exploration capabilities
2. Handles uncertainty naturally
3. More robust to environmental changes
4. Can represent multiple good solutions

4. Common Algorithms

1. **REINFORCE**
   - Uses policy gradients
   - Monte Carlo sampling
   - High variance but unbiased
2. **Hill Climbing**
   - Simple optimization
   - Uses single episode returns (G)
   - Less stable but easier to implement

Implementation Considerations

1. Sampling Actions

```
def select_action(state):
    probs = policy(state)
    action = torch.multinomial(probs, 1)
    return action
```

2. Training Process

1. Sample trajectories using current policy
2. Compute returns
3. Update policy parameters
4. Repeat

3. Common Challenges

- High variance in returns
- Sample efficiency
- Convergence stability

Best Practices

1. Use baseline subtraction
2. Implement proper exploration schedules
3. Consider using multiple episodes for updates
4. Monitor policy entropy

Applications

- Robotics control
- Game playing
- Resource allocation
- Any domain with inherent uncertainty

Summary

Stochastic policy search is particularly useful when:

- Environment has inherent randomness
- Multiple solutions are viable
- Exploration is crucial
- Continuous action spaces are involved

Remember: While single episode returns (G) aren't perfect estimates of expected returns (J), they're often "good enough" for practical learning.

Summary

Policy-Based Methods

With value-based methods, the agent uses its experience with the environment to maintain an estimate of the optimal action-value function. The optimal policy is then obtained from the optimal action-value function estimate. Policy-based methods directly learn the optimal policy, without having to maintain a separate value function estimate.

Policy Function Approximation

In deep reinforcement learning, it is common to represent the policy with a neural network. This network takes the environment state as input. If the environment has discrete actions, the output layer has a node for each possible action and contains the probability that the agent should select each possible action. The weights in this neural network are initially set to random values. Then, the agent updates the weights as it interacts with (and learns more about) the environment.

More on the Policy

Policy-based methods can learn either stochastic or deterministic policies, and they can be used to solve environments with either finite or continuous action spaces.

Here's the exact OCR of the image:

Hill Climbing

- Hill climbing is an iterative algorithm that can be used to find the weights for an optimal policy.
- At each iteration,
    a. We slightly perturb the values of the current best estimate for the weights _best, to yield a new set of weights.
    b. These new weights are then used to collect an episode. If the new weights _new resulted in higher return than the old weights, then we set _best ← _new.

Beyond Hill Climbing

- Steepest ascent hill climbing is a variation of hill climbing that chooses a small number of neighboring policies at each iteration and chooses the best among them.
- Simulated annealing uses a pre-defined schedule to control how the policy space is explored, and gradually reduces the search radius as we get closer to the optimal solution. Adaptive noise scaling decreases the search radius with each iteration when a new best policy is found, and otherwise increases the search radius.

More Black-Box Optimization

- The cross-entropy method iteratively suggests a small number of neighboring policies, and uses a small percentage of the best performing policies to calculate a new estimate.
- The evolution strategies technique considers the return corresponding to each candidate policy. The policy estimate at the next iteration is a weighted sum of all of the candidate policies, where policies that got higher return are given higher weight.

Why Policy-Based Methods?

There are three reasons why we consider policy-based methods:

- Simplicity: Policy-based methods directly get to the problem at hand (estimating the optimal policy), without having to store a bunch of additional data (i.e., the action values) that may not be useful. Stochastic policies: Unlike value-based methods, policy-based methods can learn true stochastic policies. Continuous action spaces: Policy-based methods are well-suited for continuous action spaces.

Policy Gradient Methods

Policy Gradient Methods

Overview

Policy gradient methods are a subset of policy-based methods that optimize the policy directly using gradient ascent on the expected return. Unlike simpler methods like hill climbing, they calculate exact gradients of the policy performance with respect to the policy parameters.

Key Characteristics

1. **Direct Policy Optimization**
   - Directly compute gradients of policy performance
   - Update policy parameters in direction of steepest ascent
   - More efficient than random perturbation methods
2. **Advantages**
   - More stable learning than hill climbing
   - Better sample efficiency
   - Can handle continuous action spaces naturally
   - Theoretically well-founded with convergence guarantees
3. **Policy Gradient Theorem**

   ```
   J( ) = E[ log  (a|s; ) * Q(s,a)]
   ```

   - J( ): Gradient of expected return
   - (a|s;): Policy function
   - Q(s,a): Action-value function

Common Algorithms

1. **REINFORCE (Basic Policy Gradient)**
   - Monte Carlo policy gradient
   - High variance but unbiased
   - Uses complete episode returns
2. **Actor-Critic Methods**
   - Combines policy gradient with value function estimation
   - Reduces variance through bootstrapping
   - Better sample efficiency

Implementation Tips

1. **Baseline Subtraction**
   - Subtract a baseline from returns to reduce variance
   - Common choice: state value function V(s)

- Doesn't bias gradient estimate

2. **Learning Rate Scheduling**
   - Start with small learning rates
   - Gradually decrease over time
   - Helps with convergence

Common Challenges

1. **High Variance**
   - Returns can vary significantly between episodes
   - Solution: Use baselines and advantage estimates
2. **Sample Efficiency**
   - May require many samples for reliable gradient estimates
   - Solution: Use value function bootstrapping
3. **Step Size Selection**
   - Too large: unstable learning
   - Too small: slow learning
   - Solution: Adaptive learning rates

Comparison with Other Methods

1. **vs Hill Climbing**
   - More efficient parameter updates
   - Better theoretical guarantees
   - More complex implementation
2. **vs Value-Based Methods**
   - Direct policy optimization
   - Better for continuous actions
   - Can learn stochastic policies

Remember: Policy gradient methods provide a more principled approach to policy optimization compared to simpler methods, but require careful implementation and hyperparameter tuning for best results.

Question 1: Which of the following is a valid approach, if we'd like to use a neural network to approximate an agent's stochastic policy (for a discrete action space)? (Select all that apply.)

Options:

1. Use a softmax activation function in the output layer. This will ensure the network outputs probabilities. For each state input, sample an action from the output probability distribution.
2. Use a ReLU activation function in the output layer. This will ensure the network outputs probabilities. For each state input, sample an action from the output probability distribution.

Answer: Option 1 is correct.

Explanation:

- Softmax is the appropriate choice because:
  1. It converts raw outputs into probabilities that sum to 1.0
  2. Each output is between 0 and 1
  3. It's designed for probability distributions over discrete actions
- ReLU is not suitable because:
  1. It doesn't ensure outputs sum to 1.0
  2. It only ensures non-negative values
  3. It's not designed for probability distributions

Question: Which of the following is true about the difference between policy-based and policy gradient methods? (Select all that apply.)

Alt text

Figure 1: Alt text

Options:

1. Policy gradient methods are a subclass of policy-based methods.
2. Not all policy gradient methods are policy-based methods.
3. Not all policy-based methods are policy gradient methods.
4. Both policy-based methods and policy gradient methods directly try to optimize for the optimal policy, without maintaining value function estimates.

Answer: Options 1, 3, and 4 are correct.

Explanation:

1. Policy gradient methods are indeed a subset of policy-based methods
2. This is false - all policy gradient methods are by definition policy-based methods
3. This is true - some policy-based methods (like hill climbing) don't use gradients
4. This is true - both methods directly optimize the policy without requiring value function estimates as intermediaries (though some implementations might use value functions to reduce variance, it's not a requirement)

Question: Which of the following is true about how the policy gradient method will work? (Select all that apply.)

Options:

1. For each episode, if the agent won the game, we'll amend the policy network weights to make each (state, action) pair that appeared in the episode to make them more likely to repeat in future episodes.
2. For each episode, we randomly nudge the policy weights a bit by adding Gaussian noise, and if the agent did better, we keep the new weights; otherwise, we revert to the old weights.
3. For each episode, if the agent lost the game, we'll change the policy network weights to make it less likely to repeat the corresponding (state, action) pairs in future episodes.
4. After each time step, if we arrive at a terminal state, we add noise to the policy network weights.

Answer: Options 1 and 3 are correct.

Explanation:

- Option 1 is correct: Policy gradient methods do increase the probability of (state, action) pairs that led to good outcomes (wins)
- Option 2 describes hill climbing, not policy gradient methods. Policy gradients use exact gradient calculations rather than random perturbations
- Option 3 is correct: Policy gradient methods decrease the probability of (state, action) pairs that led to poor outcomes (losses)
- Option 4 is incorrect: Policy gradient methods don't randomly add noise to weights at terminal states; they use calculated gradients based on the entire episode's return

The key distinction is that policy gradient methods use calculated gradients to systematically adjust the policy, rather than random perturbations or noise.

Connections to Supervised Learning

Important Note

Before moving on, make sure it's clear to you that the equation discussed in the video (and shown below) calculates an expectation.

U( ) = P(;)R( )

To see how it corresponds to the expected return, note that we've expressed the return R( ) as a function of the trajectory . Then, we calculate the weighted average (where the weights are given by P(;)) of all possible values that the return R( ) can take.

Why Trajectories?

You may be wondering: why are we using trajectories instead of episodes? The answer is that maximizing expected return over trajectories (instead of episodes) lets us search for optimal policies for both episodic and continuing tasks!

That said, for many episodic tasks, it often makes sense to just use the full episode. In particular, for the case of the video game example described in the lessons, reward is only delivered at the end of the episode. In this case, in order to estimate the expected return, the trajectory should correspond to the full episode; otherwise, we don't have enough reward information to meaningfully estimate the expected return.

REINFORCE

You've learned that our goal is to find the values of the weights in the neural network that maximize the expected return U

U( ) = P(;)R( )

where is an arbitrary trajectory. One way to determine the value of that maximizes this function is through gradient ascent. This algorithm is closely related to gradient descent, where the differences are that:

- gradient descent is designed to find the minimum of a function, whereas gradient ascent will find the maximum, and
- gradient descent steps in the direction of the negative gradient, whereas gradient ascent steps in the direction of the gradient.

Our update step for gradient ascent appears as follows:

← + U( )

where is the step size that is generally allowed to decay over time. Once we know how to calculate or estimate this gradient, we can repeatedly apply this update step, in the hopes that converges to the value that maximizes U( ).

Pseudocode

The algorithm described in the video is known as REINFORCE. The pseudocode is summarized below.

1. Use the policy to collect m trajectories { (1), (2), …, (m)} with horizon H. We refer to the i-th trajectory as

   (i) = (s0(i), a0(i), …, sH(i), aH(i), sH+1(i))

2. Use the trajectories to estimate the gradient U( ):

   U( ) ĝ := (1/m) log (ai(i)|si(i))R((i))

3. Update the weights of the policy:

   ← + ĝ

4. Loop over steps 1-3.

This is the REINFORCE algorithm, also known as Monte Carlo Policy Gradient, which:

Alt text

Figure 2: Alt text

Alt text

Figure 3: Alt text

1. Collects trajectory samples
2. Estimates policy gradient
3. Updates policy parameters
4. Repeats the process

The key insight is that it uses sampled trajectories to estimate the true gradient of the expected return, then uses this estimate to update the policy parameters in the direction that maximizes expected return.

Derivation

If you'd like to learn how to derive the equation that we use to approximate the gradient, please read the text below. Specifically, you'll learn how to derive

U( ) ĝ = (1/m) log (ai(i)|si(i))R((i))

This derivation is optional and can be safely skipped.

Likelihood Ratio Policy Gradient

We'll begin by exploring how to calculate the gradient U( ). The calculation proceeds as follows:

U( ) = P(;)R( ) (1) = P(;)R( ) (2) = P(;)/P(;) P(;)R( ) (3) = P(;)(P(;)/P(;))R( ) (4) = P(;) log P(;)R( ) (5)

First, we note line (1) follows directly from U( ) = P(;)R( ), where we've only taken the gradient of both sides.

Then, we can get line (2) by just noticing that we can rewrite the gradient of the sum as the sum of the gradients.

In line (3), we only multiply every term in the sum by P(;)/P(;), which is perfectly allowed because this fraction is equal to one!

Next, line (4) is just a simple rearrangement of the terms from the previous line. That is, P(;)/P(;) P(;) = P(;) P(;)/P(;).

Finally, line (5) follows from the chain rule, and the fact that the gradient of the log of a function is always equal to the gradient of the function, divided by the function. (In case it helps to see this with simpler notation, recall that x log f(x) = xf(x)/f(x).) Thus, log P(;) = P(;)/P(;).

The final "trick" that yields line (5) (i.e., log P(;) = P(;)/P(;)) is referred to as the likelihood ratio trick or REINFORCE trick.

Likewise, it is common to refer to the gradient as the likelihood ratio policy gradient:

U( ) = P(;) log P(;)R( )

Once we've written the gradient as an expected value in this way, it becomes much easier to estimate.

Alt text

Figure 4: Alt text

Sample-Based Estimate

In the video on the previous page, you learned that we can approximate the likelihood ratio policy gradient with a sample-based average, as shown below:

U( ) (1/m) log P((i);)R((i))

where each (i) is a sampled trajectory.

Finishing the Calculation

Before calculating the expression above, we will need to further simplify log P((i);). The derivation proceeds as follows:

log P((i);) = log [t=0^H P(s(i)t+1|s(i)t, a(i)t) (a(i)t|s(i)t)] (1)

= [t=0^H log P(s(i)t+1|s(i)t, a(i)t) + t=0^H log (a(i)t|s(i)t)] (2)

= t=0^H log P(s(i)t+1|s(i)t, a(i)t) + t=0^H log (a(i)t|s(i)t) (3)

= t=0^H log (a(i)t|s(i)t) (4)

= t=0^H log (a(i)t|s(i)t) (5)

First, line (1) shows how to calculate the probability of an arbitrary trajectory (i). Namely, P((i);) = t=0^H P(s(i)t+1|s(i)t, a(i)t) (a(i)t|s(i)t), where we have to take into account the action-selection probabilities from the policy and the state transition dynamics of the MDP.

Then, line (2) follows from the fact that the log of a product is equal to the sum of the logs.

Then, line (3) follows because the gradient of the sum can be written as the sum of gradients.

Next, line (4) holds, because t=0^H log P(s(i)t+1|s(i)t, a(i)t) has no dependence on , so t=0^H log P(s(i)t+1|s(i) t, a(i)t) = 0.

Finally, line (5) holds, because we can rewrite the gradient of the sum as the sum of gradients.

Plugging in the calculation above yields the equation for estimating the gradient:

U( ) ĝ = (1/m) log (ai(i)|si(i))R((i))

Summary

What are Policy Gradient Methods? Policy-based methods are a class of algorithms that search directly for the optimal policy, without simultaneously maintaining value function estimates. Policy gradient methods are a subclass of policy-based methods that estimate the weights of an optimal policy through gradient ascent. In this lesson, we represent the policy with a neural network, where our goal is to find the weights of the network that maximize expected return. The Big Picture The policy gradient method will iteratively amend the policy network weights to: make (state, action) pairs that resulted in positive return more likely, and make (state, action) pairs that resulted in negative return less likely.

Problem Setup

- A trajectory is a state-action sequence s , a , . . . , sH, aH, sH+1.
- In this lesson, we will use the notation R( ) to refer to the return corresponding to trajectory .
- Our goal is to find the weights of the policy network to maximize the expected return U( ) := P(;)R( ).

REINFORCE

The pseudocode for REINFORCE is as follows:

1. Use the policy to collect m trajectories { (1), (2), . . . , (m)} with horizon H. We refer to the i-th trajectory as (i) = (s (i), a (i), . . . , sH(i), aH(i), sH+1(i)).
2. Use the trajectories to estimate the gradient U( ): U( ) ĝ := (1/m) log (at(i)|st(i))R((i))
3. Update the weights of the policy: ← + ĝ
4. Loop over steps 1-3.

Derivation

- We derived the likelihood ratio policy gradient: U( ) = P(;) log P(;)R( ).
- We can approximate the gradient above with a sample-weighted average: U( ) (1/m) log P((i);)R((i)).
- We calculated the following: log P((i);) = t=0^H log (at(i)|st(i)).

Proximal Policy Optimization

Proximal Policy Optimization (PPO)

Overview

PPO is a policy gradient method that aims to improve training stability by limiting the size of policy updates. It addresses the challenge of determining the optimal step size when updating policies.

Key Concepts

1. Clipped Objective Function

```
L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsi
```

where:

- `r_t(\theta)` is the probability ratio: `\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}`
- `\hat{A}_t` is the estimated advantage
- `\epsilon` is the clipping parameter (typically 0.1 or 0.2)

2. Advantages Over Standard Policy Gradient

1. Better sample efficiency
2. More stable training
3. Easier hyperparameter tuning
4. Better performance across different environments

Implementation Details

1. Main Algorithm Components

```
def PPO_update():
    # Collect trajectories under current policy
    trajectories = collect_trajectories(policy_current)

    # Compute advantages and returns
```

```
    advantages = compute_advantages(trajectories)

    # Multiple epochs of optimization on collected data
    for epoch in range(K_epochs):
        # Update policy using clipped objective
        policy_loss = compute_clipped_surrogate_loss()
        optimize_policy(policy_loss)
```

2. Key Hyperparameters

- Clipping parameter ( )
- Number of epochs (K)
- Minibatch size
- Learning rate
- Value function coefficient
- Entropy coefficient

Practical Considerations

1. Value Function Updates

```
L^{VF}(\theta) = \mathbb{E}_t[(V_\theta(s_t) - R_t)^2]
```

2. Combined Loss Function

```
L^{TOTAL}(\theta) = L^{CLIP}(\theta) - c_1L^{VF}(\theta) + c_2S[\pi_\theta]
```

where:

- `c_1`, `c_2` are coefficients
- `S` is the entropy bonus

Best Practices

1. **Advantage Normalization**
   - Normalize advantages before updates
   - Helps with training stability
2. **Learning Rate Scheduling**
   - Start with larger learning rates
   - Gradually decrease during training
3. **Batch Size Selection**
   - Larger batches generally more stable
   - Trade-off with computation time

Common Challenges and Solutions

1. **Premature Convergence**
   - Solution: Adjust entropy coefficient
   - Monitor policy entropy during training
2. **Value Function Instability**
   - Solution: Use separate value network
   - Adjust value loss coefficient
3. **Sample Efficiency**
   - Solution: Proper advantage estimation
   - Multiple epochs per batch of data

Performance Metrics

1. **Training Stability**
   - Monitor policy ratio clipping frequency
   - Track value function loss
2. **Learning Progress**
   - Average episode returns
   - Policy entropy
   - KL divergence between updates

Remember: PPO's success comes from its simplicity and robustness, making it a great choice for many RL applications while being easier to implement than more complex algorithms like TRPO.

REINFORCE Algorithm - Lecture Notes

Introduction

REINFORCE is a fundamental policy gradient algorithm in reinforcement learning that directly optimizes policy parameters using gradient ascent.

Core Concepts

1. Policy Gradient Formulation

```
\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a|s)R(\tau)]
```

where:

- `J( )` is the expected return
- `_` is the policy parameterized by
- `R( )` is the return of trajectory

2. Algorithm Steps

1. **Sample Collection**

```
def collect_trajectories(policy, num_trajectories):
    trajectories = []
    for _ in range(num_trajectories):
        states, actions, rewards = [], [], []
        state = env.reset()
        done = False
        while not done:
            action = policy.sample_action(state)
            next_state, reward, done = env.step(action)
            states.append(state)
            actions.append(action)
            rewards.append(reward)
            state = next_state
        trajectories.append((states, actions, rewards))
    return trajectories
```

2. **Gradient Estimation**

```
\hat{g} = \frac{1}{m}\sum_{i=1}^m \sum_{t=0}^H \nabla_\theta \log \pi_\theta(a_t^{(i)}|s_t^{(i)})R(\tau
```

3. **Policy Update**

```
\theta \leftarrow \theta + \alpha \hat{g}
```

Key Properties

1. Advantages

- Simple and intuitive
- Unbiased gradient estimates
- Direct policy optimization
- Works with continuous action spaces

2. Limitations

1. **High Variance**
   - Monte Carlo returns can vary significantly
   - Leads to unstable training
2. **Sample Inefficiency**
   - Requires many samples for reliable gradient estimates
   - Can be slow to converge
3. **Credit Assignment**
   - Difficulty in attributing rewards to specific actions
   - All actions in trajectory receive same reward signal

Implementation Considerations

1. Baseline Subtraction

To reduce variance:

```
\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a|s)(R(\tau) -
```

where `b(s)` is the baseline (typically value function)

2. Learning Rate Selection

- Start with small learning rates
- Use adaptive methods (Adam, RMSprop)
- Consider learning rate scheduling

Common Issues and Solutions

1. **Reward Scaling**
   - Normalize returns
   - Use advantage normalization
   - Apply reward clipping
2. **Exploration**
   - Add entropy bonus
   - Use epsilon-greedy exploration
   - Implement proper initialization

Transition to PPO

REINFORCE's limitations led to development of more advanced algorithms like PPO, which:

1. Controls policy updates
2. Uses value function estimation
3. Implements clipping for stability
4. Allows multiple epochs per batch

Code Example (Basic Implementation)

```
class REINFORCE:
    def __init__(self, policy_network, optimizer, gamma=0.99):
        self.policy = policy_network
        self.optimizer = optimizer
        self.gamma = gamma

    def compute_returns(self, rewards):
        returns = []
        R = 0
        for r in reversed(rewards):
            R = r + self.gamma * R
            returns.insert(0, R)
        return torch.tensor(returns)

    def update(self, trajectories):
        for states, actions, rewards in trajectories:
            returns = self.compute_returns(rewards)
            log_probs = self.policy.get_log_probs(states, actions)
            loss = -(log_probs * returns).mean()

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
```

Remember: REINFORCE serves as the foundation for understanding more advanced policy gradient methods, making it crucial to grasp its concepts and limitations.

Beyond REINFORCE

Here, we briefly review key ingredients of the REINFORCE algorithm.

REINFORCE works as follows: First, we initialize a random policy `latex \pi_\theta(a; s)`, and using the policy we collect a trajectory – or a list of (state, actions, rewards) at each time step:

`s_1, a_1, r_1, s_2, a_2, r_2, ...`

Second, we compute the total reward of the trajectory `latex R = r_1 + r_2 + r_3 + ...`, and compute an estimate the gradient of the expected reward, g:

`g = R\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t)`

Third, we update our policy using gradient ascent with learning rate :

`\theta \leftarrow \theta + \alpha g`

The process then repeats.

What are the main problems of REINFORCE? There are three issues:

1. The update process is very **inefficient**! We run the policy once, update once, and then throw away the trajectory.
2. The gradient estimate g is very **noisy**. By chance the collected trajectory may not be representative of the policy.
3. There is no clear **credit assignment**. A trajectory may contain many good/bad actions and whether these actions are reinforced depends only on the final total output.

In the following concepts, we will go over ways to improve the REINFORCE algorithm and resolve all 3 issues. All of the improvements will be utilized and implemented in the PPO algorithm.

Noise Reduction

The way we optimize the policy is by maximizing the average rewards U( ). To do that we use stochastic gradient ascent. Mathematically, the gradient is given by an average over all the possible trajectories,

[average over all trajectories]

```
\nabla_\theta U(\theta) = \sum_\tau P(\tau;\theta)\left(R_\tau \sum_t \nabla_\theta \log \pi_\theta(a_t
```

[only one is sampled]

There could easily be well over millions of trajectories for simple problems, and infinite for continuous problems.

For practical purposes, we simply take one trajectory to compute the gradient, and update our policy. So a lot of times, the result of a sampled trajectory comes down to chance, and doesn't contain that much information about our policy. How does learning happen then? The hope is that after training for a long time, the tiny signal accumulates.

The easiest option to reduce the noise in the gradient is to simply sample more trajectories! Using distributed computing, we can collect multiple trajectories in parallel, so that it won't take too much time. Then we can estimate the policy gradient by averaging across all the different trajectories:

```
\begin{bmatrix}
    s_t^{(1)}, a_t^{(1)}, r_t^{(1)} \\
    s_t^{(2)}, a_t^{(2)}, r_t^{(2)} \\
    s_t^{(3)}, a_t^{(3)}, r_t^{(3)} \\
    \vdots
\end{bmatrix} \rightarrow g = \frac{1}{N}\sum_{i=1}^N R_i \sum_t \nabla_\theta \log \pi_\theta(a_t^{(i)}
```

Rewards Normalization

There is another bonus for running multiple trajectories: we can collect all the total rewards and get a sense of how they are distributed.

In many cases, the distribution of rewards shifts as learning happens. Reward = 1 might be really good in the beginning, but really bad after 1000 training episode.

Learning can be improved if we normalize the rewards, where is the mean, and the standard deviation.

```
R_i \leftarrow \frac{R_i - \mu}{\sigma}, \quad \mu = \frac{1}{N}\sum_i R_i, \quad \sigma = \sqrt{\frac{
```

(when all the Ri are the same, = 0, we can set all the normalized rewards to 0 to avoid numerical problems)

This batch-normalization technique is also used in many other problems in AI (e.g. image classification), where normalizing the input can improve learning.

Intuitively, normalizing the rewards roughly corresponds to picking half the actions to encourage/discourage, while also making sure the steps for gradient ascents are not too large/small.

Credit Assignment

Going back to the gradient estimate, we can take a closer look at the total reward R, which is just a sum of reward at each step R = r + r + … + rt-1 + rt + …

```
g = \sum_t(... + r_{t-1} + r_t + ...)\nabla_\theta \log \pi_\theta(a_t|s_t)
```

Let's think about what happens at time-step t. Even before an action is decided, the agent has already received all the rewards up until step t - 1. So we can think of that part of the total reward as the reward from the past. The rest is denoted as the future reward.

```
(\underbrace{... + r_{t-1}}_{R_t^{\text{past}}} + \underbrace{r_t + ...}_{R_t^{\text{future}}})
```

Because we have a Markov process, the action at time-step t can only affect the future reward, so the past reward shouldn't be contributing to the policy gradient. So to properly assign credit to the action at, we should ignore the past reward. So a better policy gradient would simply have the future reward as the coefficient.

```
g = \sum_t R_t^{\text{future}}\nabla_\theta \log \pi_\theta(a_t|s_t)
```

Notes on Gradient Modification

You might wonder, why is it okay to just change our gradient? Wouldn't that change our original goal of maximizing the expected reward?

It turns out that mathematically, ignoring past rewards might change the gradient for each specific trajectory, but it doesn't change the **averaged** gradient. So even though the gradient is different during training, on average we are still maximizing the average reward. In fact, the resultant gradient is less noisy, so training using future reward should speed things up!

Suppose we are training an agent to play a computer game. There are only two possible action:

- 0: Do nothing
- 1: Move

There are three time-steps in each game, and our policy is completely determined by one parameter , such that the probability of "moving" is , and the probability of doing nothing is 1 -

Initially = 0.5. Three games are played, the results are:

Game	Actions	Rewards
1	(1,0,1)	(1,0,1)
2	(1,0,0)	(0,0,1)
3	(0,1,0)	(1,0,1)

Let's solve these questions one by one using the information from the table and previous discussion about future rewards.

Question 1: What are the future rewards for the first game?

- Game 1 data: actions (1,0,1), rewards (1,0,1)
- Answer: (2,1,1)
- Explanation:
  - For each time step, future reward = sum of current and all future rewards
  - At t=1: 1 + 0 + 1 = 2
  - At t=2: 0 + 1 = 1
  - At t=3: 1 = 1
  - Therefore future rewards are (2,1,1)

Question 2: What is the policy gradient computed from the second game, using future rewards?

- Game 2 data: actions (1,0,0), rewards (0,0,1)
- Answer: -2
- Explanation:
  - Future rewards for game 2 would be (1,1,1) since:

    - t=1: 0 + 0 + 1 = 1
    - t=2: 0 + 1 = 1
    - t=3: 1 = 1
  - For = 0.5, log gradients are:
    - For action 1: 1/0.5 = 2
    - For action 0: -1/(1-0.5) = -2
  - Sum of gradients × future rewards:
    - (2×1) + (-2×1) + (-2×1) = -2

Question 3: Which statements are true regarding the 3rd game?

- Game 3 data: actions (0,1,0), rewards (1,0,1)
- Future rewards would be (2,1,1)

For Game 3: actions: (0,1,0) rewards: (1,0,1)

Let's analyze each statement:

1. "We can add a baseline -1 point to the rewards, the computed gradient wouldn't change"
   - This is FALSE
   - Adding a baseline keeps the average gradient the same, but individual computed gradients would differ
2. "The contribution to the gradient from the second and third steps cancel each other"
   - This is TRUE
   - With actions (0,1,0), the second and third steps have opposite effects on the gradient
3. "The computed policy gradient from this game is 0"
   - This is FALSE
   - With future rewards (2,1,1), and computed gradient is -2/0.5 = -4
4. "The computed policy gradient from this game is negative"
   - This is TRUE
   - The final computed gradient is negative (-4)
5. "Using the total reward vs future reward give the same policy gradient in this game"
   - This is TRUE
   - Total reward = 2
   - The second and third time-steps cancel each other anyway
   - So both methods yield the same result

The key insight is that while some methods might seem different, in this specific game they produce equivalent results due to the cancellation of later steps and the structure of the rewards.

Importance Sampling

1. Policy Update in REINFORCE

Let's go back to the REINFORCE algorithm. We start with a policy, , then using that policy, we generate a trajectory (or multiple ones to reduce noise) (st, at, rt). Afterward, we compute a policy gradient, g, and update ' ← + g.

At this point, the trajectories we've just generated are simply thrown away. If we want to update our policy again, we would need to generate new trajectories once more, using the updated policy.

You might ask, why is all this necessary? It's because we need to compute the gradient for the current policy, and to do that the trajectories need to be representative of the current policy.

But this sounds a little wasteful. What if we could somehow recycle the old trajectories, by modifying them so that they are representative of the new policy? So that instead of just throwing them away, we recycle them!

Then we could just reuse the recycled trajectories to compute gradients, and to update our policy, again, and again. This would make updating the policy a lot more efficient. So, how exactly would that work?

2. Importance Sampling

This is where importance sampling comes in. Let's look at the trajectories we generated using the policy . It had a probability P(;), to be sampled.

Now Just by chance, the same trajectory can be sampled under the new policy, with a different probability P(; ')

Imagine we want to compute the average of some quantity, say f( ). We could simply generate trajectories from the new policy, compute f( ) and average them.

Mathematically, this is equivalent to adding up all the f( ), weighted by a probability of sampling each trajectory under the new policy.

```
\sum_\tau P(\tau;\theta')f(\tau)
```

Now we could modify this equation, by multiplying and dividing by the same number, P(;) and rearrange the terms.

```
\sum_\tau \underbrace{P(\tau;\theta)}_{\text{sampling under old policy }\pi_\theta} \underbrace{\frac{P
```

It doesn't look we've done much. But written in this way, we can reinterpret the first part as the coefficient for sampling under the old policy, with an extra re-weighting factor, in addition to just averaging.

Intuitively, this tells us we can use old trajectories for computing averages for new policy, as long as we add this extra re-weighting factor, that takes into account how under or over-represented each trajectory is under the new policy compared to the old one.

The same tricks are used frequently across statistics, where the re-weighting factor is included to un-bias surveys and voting predictions.

3. The re-weighting factor

Now Let's a closer look at the re-weighting factor.

```
\frac{P(\tau;\theta')}{P(\tau;\theta)} = \frac{\pi_{\theta'}(a_1|s_1) \pi_{\theta'}(a_2|s_2) \pi_{\theta
```

Because each trajectory contains many steps, the probability contains a chain of products of each policy at different time-step.

This formula is a bit complicated. But there is a bigger problem. When some of policy gets close to zero, the re-weighting factor can become close to zero, or worse, close to 1 over 0 which diverges to infinity.

When this happens, the re-weighting trick becomes unreliable. So, in practice, we want to make sure the re-weighting factor is not too far from 1 when we utilize importance sampling.

PPO part 1- The Surrogate Function

Re-weighting the Policy Gradient

Suppose we are trying to update our current policy, '. To do that, we need to estimate a gradient, g. But we only have trajectories generated by an older policy . How do we compute the gradient then?

Mathematically, we could utilize importance sampling. The answer just what a normal policy gradient would be, times a re-weighting factor P(; ')/P(;):

```
g = \frac{P(\tau;\theta')}{P(\tau;\theta)} \sum_t \frac{\nabla_{\theta'}\pi_{\theta'}(a_t|s_t)}{\pi_{\t
```

We can rearrange these equations, and the re-weighting factor is just the product of all the policy across each step – I've picked out the terms at time-step t here. We can cancel some terms, but we're still left with a product of the policies at different times, denoted by "…".

```
g = \sum_t \frac{...\pi_{\theta'}(a_t|s_t)...}{...\pi_{\theta}(a_t|s_t)...} \frac{\nabla_{\theta'}\pi_{\
```

Can we simplify this expression further? This is where proximal policy comes in. If the old and current policy is close enough to each other, all the factors inside the "…" would be pretty close to 1, and then we can ignore them.

Then the equation simplifies

```
g = \sum_t \frac{\nabla_{\theta'}\pi_{\theta'}(a_t|s_t)}{\pi_{\theta}(a_t|s_t)} R_t^{\text{future}}
```

It looks very similar to the old policy gradient. In fact, if the current policy and the old policy is the same, we would have exactly the vanilla policy gradient. But remember, this expression is different because we are comparing two *different* policies.

The Surrogate Function

Now that we have the approximate form of the gradient, we can think of it as the gradient of a new object, called the surrogate function:

```
g = \nabla_{\theta'} L_{\text{sur}}(\theta', \theta)
```

```
L_{\text{sur}}(\theta', \theta) = \sum_t \frac{\pi_{\theta'}(a_t|s_t)}{\pi_{\theta}(a_t|s_t)} R_t^{\tex
```

So using this new gradient, we can perform gradient ascent to update our policy – which can be thought of as directly maximizing the surrogate function.

But there is still one important issue we haven't addressed yet. If we keep reusing old trajectories and updating our policy, at some point the new policy might become different enough from the old one, so that all the approximations we made could become invalid.

PPO part 2- Clipping Policy Updates

PPO's clipping mechanism is like having a safety limit on how much an AI can change its behavior at once - imagine teaching someone to throw a ball, where instead of completely changing their throwing style (which could make things worse), you only allow small adjustments each time (like tweaking their arm angle by 10-20%). This prevents the AI from making dramatic changes that could break what it already learned, similar to how you'd teach someone new skills through gradual improvements rather than radical changes, ensuring stable and consistent learning over time.

I don't see the actual content for "PPO part 2- Clipping Policy Updates" in your message. However, I can explain the key concepts of PPO's clipping mechanism:

PPO Part 2 - Clipping Policy Updates

The key innovation in PPO is the introduction of a clipped objective function that prevents too large policy updates. Here's how it works:

1. **Probability Ratio**

```
r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_\text{old}}(a_t|s_t)}
```

2. **Clipped Objective**

```
L^{\text{CLIP}}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\eps
```

Where:

Alt text

Figure 5: Alt text

Alt text

Figure 6: Alt text

- is a small hyperparameter (typically 0.1 or 0.2)
- At is the advantage estimate
- The clip function restricts the ratio to [1- , 1+]

Key Benefits:

1. Prevents destructively large policy updates
2. Maintains trust region without complex second-order optimization
3. Improves training stability
4. Simple to implement

The clipping mechanism ensures that the ratio between new and old policies doesn't deviate too much from 1, effectively creating a trust region that bounds policy updates.

The Policy/Reward Cliff

What is the problem with updating our policy and ignoring the fact that the approximations are not valid anymore? One problem is it could lead to a really bad policy that is very hard to recover from. Let's see how:

Say we have some policy parameterized by ' (shown on the left plot in black), and with an average reward function (shown on the right plot in black).

The current policy is labelled by the red text, and the goal is to update the current policy to the optimal one (green star). To update the policy we can compute a surrogate function Lsur (dotted-red curve on right plot). So Lsur approximates the reward pretty well around the current policy. But far away from the current policy, it diverges from the actual reward.

If we continually update the policy by performing gradient ascent, we might get something like the red-dots. The big problem is that at some point we hit a cliff, where the policy changes by a large amount. From the perspective of the surrogate function, the average reward is really great. But the actually average reward is really bad!

What's worse, the policy is now stuck in a deep and flat bottom, so that future updates won't be able to bring the policy back up! we are now stuck with a really bad policy.

How do we fix this? Wouldn't it be great if we can somehow stop the gradient ascent so that our policy doesn't fall off the cliff?

Clipped Surrogate Function

Policy Update with Flattened Surrogate Function

Here's an idea: what if we just flatten the surrogate function (blue curve)? What would policy update look like then?

So starting with the current policy (blue dot), we apply gradient ascent. The updates remain the same, until we hit the flat plateau. Now because the reward function is flat, the gradient is zero, and the policy update will stop!

Alt text

Figure 7: Alt text

Now, keep in mind that we are only showing a 2D figure with one ' direction. In most cases, there are thousands of parameters in a policy, and there may be hundreds/thousands of high-dimensional cliffs in many different directions. We need to apply this clipping mathematically so that it will automatically take care of all the cliffs.

Clipped Surrogate Function

Here's the formula that will automatically flatten our surrogate function to avoid all the cliffs:

```
L_sur^{clip}(\theta', \theta) = \sum_t \min\left\{\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}R_t
```

Now let's dissect the formula by looking at one specific term in the sum, and set the future reward to 1 to make things easier.

We start with the original surrogate function (red), which involves the ratio '(at|st)/ (at|st). The black dot shows the location where the current policy is the same as the old policy (' =)

We want to make sure the two policy is similar, or that the ratio is close to 1. So we choose a small (typically 0.1 or 0.2), and apply the clip function to force the ratio to be within the interval [1 - , 1 +] (shown in purple).

Now the ratio is clipped in two places. But we only want to clip the top part and not the bottom part. To do that, we compare this clipped ratio to the original one and take the minimum (show in blue). This then ensures the clipped surrogate function is always less than the original surrogate function `textmate L_sur^{clip} L_sur`, so the clipped surrogate function gives a more conservative "reward".

(the blue and purple lines are shifted slightly for easier viewing)

So that's it! We can finally summarize the PPO algorithm:

1. First, collect some trajectories based on some policy , and initialize theta prime:

   ```
   ' =
   ```

2. Next, compute the gradient of the clipped surrogate function using the trajectories

3. Update ' using gradient ascent:

   ```
   ' ← ' +  'L_sur^{clip}( ', )
   ```

4. Then we repeat step 2-3 without generating new trajectories. Typically, step 2-3 are only repeated a few times

5. Set = ', go back to step 1, repeat.

The details of PPO was originally published by the team at OpenAI, and you can read their paper through this link

Note: The link mentioned in the text isn't visible in the image, but the original PPO paper is "Proximal Policy Optimization Algorithms" by Schulman et al., published by OpenAI.

Deep RL Actor-Critic Methods - Notes

Simple Overview

Actor-Critic methods are like having two specialists working together: an "actor" that learns to make decisions (like a chess player making moves), and a "critic" that evaluates how good those decisions are (like a chess coach providing feedback). This combination helps the AI learn more efficiently and stably than using either approach alone.

Detailed Explanation

1. Core Components

- **Actor (Policy Network)**

```
\pi_\theta(a|s) = P(a|s;\theta)
```

Where:

- is the policy function
- represents the actor's neural network parameters
- a is the action
- s is the state

- **Critic (Value Network)**

```
V_\phi(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t|s_0=s]
```

Where:

- V is the value function
- represents the critic's neural network parameters
- is the discount factor
- r is the reward

2. Advantage Function

The critic estimates the advantage:

```
A(s,a) = Q(s,a) - V(s)
```

This tells us how much better an action is compared to the average action in that state.

3. Policy Update

The actor updates using the critic's feedback:

```
\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \log \pi_\theta(a|s)A(s,a)]
```

4. Learning Process

1. Actor collects experience by interacting with environment
2. Critic evaluates the state-action values
3. Calculate advantages using critic's estimates
4. Update both networks:
   - Actor moves toward better actions
   - Critic improves its value estimates

5. Key Benefits

1. Reduced variance compared to pure policy gradient methods
2. Better sample efficiency
3. Works well with continuous action spaces
4. More stable learning dynamics

6. Common Variants

1. A2C/A3C: Synchronous/Asynchronous advantage actor-critic
2. PPO: Combines actor-critic with trust region optimization
3. SAC: Adds entropy maximization for exploration
4. TD3: Uses twin critics to reduce overestimation

7. Implementation Considerations

- Learning rate balancing between actor and critic
- Network architecture design
- Proper advantage estimation
- Exploration strategy
- Batch size and update frequency

This architecture forms the backbone of many modern RL algorithms, combining the best aspects of both policy-based and value-based methods.

Bias and Variance

Think of bias and variance like aiming at a target: bias is how far your average shots are from the bullseye (systematic error), while variance is how spread out your shots are (inconsistency). In reinforcement learning, we need to balance these two factors - too much bias means consistently wrong estimates, while too high variance means unstable learning.

Detailed Explanation

1. Mathematical Foundation

The mean squared error (MSE) can be decomposed as:

```
MSE = Bias^2 + Variance + Irreducible Error
```

2. Bias in RL

- **Definition**: Systematic deviation from true value
- **Sources**:

  ```
  Bias[V(s)] = E[V_est(s)] - V_true(s)
  ```

  1. Function approximation limitations
  2. Bootstrapping from incomplete information
  3. Limited state/action space exploration

3. Variance in RL

- **Definition**: Variability in estimates
- **Sources**:

  ```
  Variance[V(s)] = E[(V_est(s) - E[V_est(s)])^2]
  ```

  1. Stochastic environments

2. Random sampling
3. Policy exploration

4. Trade-offs

1. **Monte Carlo Methods**
   - Zero bias (uses actual returns)
   - High variance (full trajectory needed)

   `G_t = R_t + R_{t+1} + ^2R_{t+2} + ...`
2. **TD Learning**
   - Some bias (bootstrapping)
   - Lower variance (single step)

   `V(s_t) ← V(s_t) + [R_t + V(s_{t+1}) - V(s_t)]`

5. Common Solutions

1. **Baseline Subtraction**
   - Reduces variance without adding bias

   `A(s,a) = Q(s,a) - V(s)`
2. **N-step Returns**
   - Balances bias-variance trade-off

   `G_t^{(n)} = R_t + R_{t+1} + ... + ^{n-1}R_{t+n-1} + ^nV(s_{t+n})`
3. **GAE (Generalized Advantage Estimation)**
   - Combines multiple n-step estimates

   `A^{GAE}(s,a) = \sum_{k=0}^{\infty} ( )^k _{t+k}`

6. Practical Implications

1. Choose algorithms based on environment characteristics
2. Monitor training stability
3. Adjust hyperparameters to balance trade-off
4. Use appropriate baseline functions
5. Consider ensemble methods

Understanding this trade-off is crucial for:

- Algorithm selection
- Hyperparameter tuning
- Debugging learning issues
- Performance optimization

Monte Carlo vs TD Learning - Bias-Variance Trade-off

Simple Overview

- **Monte Carlo (MC)**: High variance, NO bias
  - Like calculating your average test score using ALL your test results
  - More accurate (no bias) but needs lots of samples (high variance)
- **Temporal Difference (TD)**: Low variance, SOME bias
  - Like estimating your final grade mid-semester using partial results
  - More efficient (low variance) but less accurate (biased)

Detailed Explanation

Monte Carlo

```
G_t = r_t +  r_{t+1} +  ^2r_{t+2} + ... +  ^{T-t}r_T
```

1. **Why No Bias?**
   - Uses actual, complete returns
   - Doesn't make assumptions about future rewards
   - Waits until episode ends to calculate true return
2. **Why High Variance?**
   - Needs full episodes
   - Subject to randomness in environment
   - Each trajectory can be very different
   - Requires many samples to get stable estimates

Temporal Difference

```
V(s_t) ← V(s_t) +  [r_t +  V(s_{t+1}) - V(s_t)]
```

1. **Why Some Bias?**
   - Uses bootstrapping (estimates to predict estimates)
   - Makes assumptions about future values
   - Relies on current value function approximation
2. **Why Low Variance?**
   - Uses single-step updates
   - Doesn't wait for episode end
   - More frequent updates
   - More stable learning

Key Takeaway

The trade-off is between:

- MC: Perfect accuracy (no bias) but unstable learning (high variance)
- TD: Approximate accuracy (some bias) but stable learning (low variance)

This is why many modern algorithms use combinations of both approaches (like n-step returns or GAE) to get the best of both worlds.

The argument you often hear as to why to call a neural network trained with Monte-Carlo estimates a "Critic" is because function approximators, such as a neural network, are biased as a byproduct that they are not perfect. That's a fair point, though, I prefer the distinction based on whether we pick a Monte-Carlo or a TD estimate to train our function approximator. Now, definitely we should not be calling Actor-Critic methods every method that uses 2 neural networks. You'll be surprised!

The important takeaway for you, though, is that there are inconsistencies out there. You often see methods named "Actor-Critic" when they are not. I just want to bring the issue to your attention.

Policy-based, Value-Based, and Actor-Critic

Comparison of RL Methods

Aspect	Policy-Based	Value-Based	Actor-Critic
Main Idea	Directly learn policy (a|s)	Learn value function Q(s,a) or V(s)	Combine policy and value learning
Output	Action probabilities	Value of state/action	Both actions and values

Aspect	Policy-Based	Value-Based	Actor-Critic
Action Space	Continuous & Discrete	Mainly Discrete	Continuous & Discrete
Convergence	Guaranteed to local optimum	Can oscillate	Better convergence
Sample Efficiency	Low	High	High
Variance	High	Low	Medium
Bias	Low/No bias	Can be high	Moderate
Examples	REINFORCE, PPO	DQN, Q-Learning	A2C, SAC, PPO
Memory Usage	Low	High (replay buffer)	Moderate
Stability	Less stable	More stable	Most stable
Exploration	Natural exploration	Needs explicit exploration	Both explicit & natural
Hyperparameters	Fewer	Many	Many

Legend: - Advantage - Disadvantage - Depends/Moderate

A Basic Actor-Critic Agent

One important thing to note here is that I use `textmate V(s; _v)` or `textmate A(s, a)`, but sometimes `textmate V_ (s; _v)` or `textmate A_ (s, a)` (see the there? See the _v? What's going on?)

There are 2 thing actually going on in there.

1. A very common thing you'll see in reinforcement learning is the oversimplification of notation. However, both styles, whether you see `textmate A(s, a)`, or `textmate A_ (s, a)` (value functions with or without a ,) it means you are evaluating a value function of policy . In the case of A, the advantage function. A different case would be when you see a superscript *. For example, `textmate A*(s, a)` means the optimal advantage function. Q-learning learns the optimal action-value function, `textmate Q*(s, a)`, for example.
2. The other thing is the use of _v in some value functions and not in others. This only means that such value function is using a neural network. For example, `textmate V(s; _v)` is using a neural network as a function approximator, but `textmate A(s, a)` is not. We are calculating the advantage function `textmate A(s, a)` using the state-value function `textmate V(s; _v)`, but `textmate A(s, a)` is not using function approximation directly.

A3C: Asynchronous Advantage Actor-Critic, N-step

A3C is like having multiple students learning the same task independently and sharing their experiences with a central teacher. Each student (agent) learns in parallel, using both an actor (decision maker) and critic (evaluator), while periodically updating a shared master model. This parallel learning makes training more efficient and stable.

Detailed Explanation

1. Core Components

1. **Actor Network (Policy)**

```
(a|s; ) = P(a|s; )
```

2. **Critic Network (Value)**

```
V(s; _v) = E[R_t|s_t = s]
```

3. **N-step Returns**

```
R_t^{(n)} = r_t +  r_{t+1} + ... +  ^{n-1}r_{t+n-1} +  ^nV(s_{t+n})
```

2. Advantage Estimation

```
A(s_t, a_t) = R_t^{(n)} - V(s_t)
```

3. Loss Functions

1. **Policy Loss**

```
L_  = -log  (a_t|s_t; )A(s_t, a_t) +   H( (·|s_t))
```

Where H is entropy bonus

2. **Value Loss**

```
L_v = (R_t^{(n)} - V(s_t))^2
```

4. Key Features

1. **Asynchronous Updates**
   - Multiple agents train in parallel
   - Each agent has local copy of networks
   - Periodic updates to global network
   - No need for experience replay
2. **N-step Returns**
   - Balance between MC and TD
   - Better credit assignment
   - Reduced variance compared to MC
   - Less bias than 1-step TD
3. **Advantage Actor-Critic**
   - Combines policy and value learning
   - Reduced variance through baseline
   - Better policy gradients

5. Implementation Tips

1. **Parallel Processing**
   - Use multiple CPU cores
   - Independent environment copies
   - Asynchronous gradient updates
   - Lock-free implementation
2. **Hyperparameters**
   - Learning rates (actor & critic)
   - N-step length (typically 5-20)
   - Entropy coefficient
   - Update frequency
3. **Network Architecture**
   - Shared layers between actor/critic
   - Separate output heads
   - Appropriate activation functions

6. Advantages

1. More stable than A2C
2. Better exploration through parallelization

3. No replay buffer needed
4. Good sample efficiency
5. Works well on both discrete and continuous actions

7. Limitations

1. Implementation complexity
2. Hardware requirements
3. Sensitive to hyperparameters
4. Potential communication overhead

A3C remains influential despite newer algorithms, demonstrating the power of asynchronous learning in RL.

Off-policy vs On-policy in A3C - Notes

Simple Overview

Think of on-policy as learning from your own experiences (like learning to cook by following your own recipes), while off-policy is learning from others' experiences (like learning from cooking shows). A3C is primarily on-policy, meaning each agent learns from its own actions, but understanding both approaches helps grasp why this choice matters.

Detailed Explanation

1. Basic Definitions

On-Policy

```
_behavior =  _target
```

- Same policy generates actions and learns from them
- More stable but less sample efficient

Off-Policy

```
_behavior   _target
```

- Different policies for action selection and learning
- Requires importance sampling correction:

```
_t = \frac{ _target(a_t|s_t)}{ _behavior(a_t|s_t)}
```

2. A3C's On-Policy Nature

1. **Policy Updates**

```
_  J( ) = E_ [ _  log  _ (a_t|s_t)A(s_t,a_t)]
```

- Uses current policy for both sampling and updates
- No importance sampling needed
- Direct gradient estimation

2. **Value Function Updates**

```
V_new(s_t) ← V(s_t) +  [R_t^{(n)} - V(s_t)]
```

- Based on actions from current policy
- N-step returns from same trajectory

3. Comparison

Aspect	On-Policy (A3C)	Off-Policy
Sample Efficiency	Lower	Higher
Stability	Higher	Lower
Implementation	Simpler	Complex
Memory Usage	Lower	Higher
Experience Replay	Not needed	Required

4. Trade-offs

1. **Advantages of A3C's On-Policy Approach**
   - Simpler implementation
   - No replay buffer needed
   - More stable learning
   - Natural exploration
2. **Limitations**
   - Can't reuse old experiences
   - Lower sample efficiency
   - Requires parallel environments
   - More compute-intensive

5. Implementation Considerations

1. **Synchronization**

```
_global ← _global +  _ J( _local)
```

- Local policies must stay close to global
- Frequent updates needed

2. **Exploration**

- Relies on policy entropy
- Each agent explores independently

```
L = L_ + L_V + H( )
```

6. Best Practices

1. **Update Frequency**
   - Balance communication overhead
   - Keep policies synchronized
   - Maintain stability
2. **Parallelization**
   - Multiple independent environments
   - Asynchronous updates
   - Load balancing
3. **Hyperparameter Tuning**
   - Learning rates
   - Entropy bonus
   - N-step length
   - Number of parallel agents

7. Modern Context

1. **Evolution**
   - PPO: On-policy with better sample efficiency
   - SAC: Off-policy with stability
   - IMPALA: Distributed architecture
2. **Current Usage**
   - Still relevant for specific scenarios
   - Foundation for newer algorithms
   - Benchmark for comparison

Understanding these distinctions helps in: - Algorithm selection - Implementation choices - Performance optimization - Debugging issues

A2C (Advantage Actor-Critic) - Notes

Simple Overview

A2C is like having a player (actor) and a coach (critic) working together: the player learns to make better decisions while the coach evaluates how good those decisions are. It's the synchronous version of A3C, meaning all agents wait for each other before updating, making it simpler but potentially slower than A3C.

Detailed Explanation

1. Core Components

1. **Actor (Policy Network)**

```
_(a|s) = P(a|s; )
```

2. **Critic (Value Network)**

```
V_(s) = E[ ^t r_t|s]
```

3. **Advantage Estimation**

```
A(s_t, a_t) = R_t - V_(s_t)
```

2. Loss Functions

1. **Policy Loss**

```
L_  = -log  _(a_t|s_t)A(s_t,a_t) +  H( _ )
```

Where H is entropy bonus for exploration

2. **Value Loss**

```
L_v = (R_t - V_(s_t))^2
```

3. **Total Loss**

```
L_total = L_  + c_v L_v - c_e H( _ )
```

3. Key Features

1. **Synchronous Updates**
   - All agents collect experiences
   - Wait for all to finish
   - Update networks simultaneously
   - More stable, but slower than A3C

2. **Advantage Function Benefits**
    - Reduces variance
    - Better credit assignment
    - More stable training
3. **Shared Network Architecture**
    - Common feature extraction layers
    - Separate policy and value heads
    - Parameter sharing efficiency

4. Implementation Tips

1. **Batching**
    - Collect experiences in parallel
    - Synchronous updates
    - Efficient GPU utilization
    - Vectorized environments
2. **Hyperparameters**
    - Learning rates (actor & critic)
    - Value loss coefficient (c_v)
    - Entropy coefficient (c_e)
    - Discount factor ( )
3. **Common Issues & Solutions**
    - Value function scaling
    - Advantage normalization
    - Gradient clipping
    - Learning rate scheduling

5. Advantages

1. Simpler implementation than A3C
2. Better GPU utilization
3. More stable training
4. Deterministic behavior
5. Easier to debug

6. Limitations

1. Slower than A3C
2. Synchronization overhead
3. Less exploration than A3C
4. Resource underutilization

7. Best Practices

1. Use vectorized environments
2. Normalize observations
3. Monitor value function accuracy
4. Balance exploration vs exploitation
5. Regular gradient norm checks

A2C remains popular due to its simplicity and stability, making it a good baseline for new RL implementations.

Alt text

Figure 8: Alt text

GAE (Generalized Advantage Estimation) - Notes

Simple Overview

GAE is like having adjustable binoculars for evaluating actions: you can tune how far ahead you look ( ) and how much you trust future predictions ( ). It combines multiple n-step advantage estimates to get the best of both worlds - the accuracy of long-term rewards and the stability of short-term estimates.

Detailed Explanation

1. Core Formula

```
A^{GAE( , )} =  ( )^k  _k
```

Where temporal difference (TD) error is:

```
 _t = r_t +  V(s_{t+1}) - V(s_t)
```

2. Components Breakdown

1. **Hyperparameters**
   - (gamma): discount factor for rewards
   - (lambda): controls trade-off between bias/variance
2. **Value Function**

```
V(s_t) = E[ ^k r_{t+k}]
```

3. **N-step Advantage Estimates**

```
A^{(n)}_t = -V(s_t) + r_t +  r_{t+1} + ... +  ^{n-1}r_{t+n-1} +  ^nV(s_{t+n})
```

3. Trade-off Control

1. **= 0**
   - Uses only one-step TD error
   - Low variance, high bias

   ```
   A^{GAE}_t =  _t
   ```
2. **= 1**
   - Uses full Monte Carlo return
   - High variance, no bias

   ```
   A^{GAE}_t =  ^k  _{t+k}
   ```

4. Implementation Tips

1. **Practical Considerations**
   - Normalize advantages
   - Use proper value function initialization
   - Handle terminal states correctly
   - Implement efficient computation
2. **Common Values**
   - : 0.99 (typical)
   - : 0.95 (typical)
   - Advantage normalization
   - Value function clipping

Alt text

Figure 9: Alt text

5. Benefits

1. **Flexibility**
   - Adjustable bias-variance trade-off
   - Works with any policy optimization
   - Compatible with value-based methods
2. **Performance**
   - More stable learning
   - Better credit assignment
   - Reduced variance in policy gradients

6. Code Structure

```
def compute_gae(rewards, values, gamma, lambda_):
    advantages = []
    gae = 0
    for t in reversed(range(len(rewards))):
        delta = rewards[t] + gamma * values[t+1] - values[t]
        gae = delta + gamma * lambda_ * gae
        advantages.insert(0, gae)
    return advantages
```

7. Common Issues & Solutions

1. **Numerical Stability**
   - Use log-space computations
   - Clip extreme values
   - Normalize advantages
2. **Hyperparameter Sensitivity**
   - Start with recommended values
   - Tune based on environment
   - Monitor value estimates

8. Best Practices

1. **Implementation**
   - Vectorize computations
   - Handle episode boundaries
   - Proper advantage scaling
   - Regular sanity checks
2. **Integration**
   - Works well with PPO
   - Compatible with A2C/A3C
   - Can improve TRPO
   - Enhances policy gradients

GAE is a crucial component in modern RL algorithms, particularly in PPO, offering a flexible and effective way to estimate advantages.

DDPG (Deep Deterministic Policy Gradient) - Notes

Simple Overview

DDPG is like teaching a robot to perform precise movements: instead of choosing from a set of discrete actions (like "move left" or "right"), it can output exact values (like "rotate joint by 27.3 degrees"). It combines the best ideas from DQN (experience replay, target networks) with actor-critic methods for continuous action spaces.

Detailed Explanation

1. Core Components

1. **Actor Network ( )**

```
(s| ^ ): S → A
```

Deterministic policy that maps states to specific actions

2. **Critic Network (Q)**

```
Q(s,a| ^Q): S × A → R
```

Evaluates state-action pairs

3. **Target Networks**

```
 '( ^{ '})   ( ^ )
Q'( ^{Q'})   Q( ^Q)
```

2. Update Rules

1. **Critic Update**

```
y_i = r_i +  Q'(s_{i+1},  '(s_{i+1}| ^{ '})| ^{Q'})
L = 1/N  (y_i - Q(s_i,a_i| ^Q))^2
```

2. **Actor Update**

```
 _{ ^ }J   1/N  _aQ(s,a| ^Q)|_{s=s_i,a= (s_i)} _{ ^ } (s| ^ )|_{s=s_i}
```

3. Key Features

1. **Exploration**
   - Ornstein-Uhlenbeck process
   - Gaussian noise
   - Parameter space noise
2. **Experience Replay**
   - Store (s, a, r, s') transitions
   - Random sampling
   - Batch updates
3. **Soft Updates**

```
 ' ←   + (1- ) '
```

Where 1 (typically 0.001)

4. Implementation Tips

1. **Network Architecture**
   - Separate actor and critic
   - Batch normalization

- Action input to critic's second layer
- ReLU activations
- tanh output for actor

2. **Hyperparameters**
   - Learning rates (actor & critic)
   - Soft update rate ( )
   - Batch size
   - Buffer size
   - Noise parameters
3. **Stability Tricks**
   - Gradient clipping
   - Action scaling
   - State normalization
   - Layer normalization

5. Advantages

1. Continuous action spaces
2. Sample efficient
3. Off-policy learning
4. End-to-end learning
5. No policy discretization

6. Limitations

1. Sensitive to hyperparameters
2. Requires careful tuning
3. Can be unstable
4. Exploration challenges
5. Biased gradient estimates

7. Best Practices

1. **Training**
   - Normalize observations
   - Clip critic gradients
   - Use proper initialization
   - Monitor Q-values
2. **Debugging**
   - Start with simple environments
   - Validate critic predictions
   - Check action bounds
   - Monitor target networks

8. Modern Variants

1. TD3 (Twin Delayed DDPG)
   - Double critics
   - Delayed policy updates
   - Target policy smoothing
2. SAC (Soft Actor-Critic)
   - Stochastic policies
   - Entropy regularization
   - Better exploration

DDPG remains influential in continuous control tasks, forming the foundation for modern algorithms like TD3 and SAC.

DDPG Soft Updates

Soft updates are like gradually mixing paint colors instead of switching them instantly. Instead of directly copying weights from the main networks to target networks (hard update), DDPG slowly blends them together. This creates more stable learning by preventing sudden changes in the target values.

Detailed Explanation

1. Core Formula

_target ← _main + (1-) _target

Where: - (tau) is a small number (typically 0.001) - _target: target network parameters - _main: main network parameters

2. Comparison

1. **Hard Updates (DQN style)**

_target = _main

- Complete replacement
- Every N steps
- Can cause instability

2. **Soft Updates (DDPG style)**

_target = 0.001 × _main + 0.999 × _target

- Gradual changes
- Every step
- More stable learning

3. Implementation Example

```
def soft_update(target_network, main_network, tau=0.001):
    for target_param, main_param in zip(target_network.parameters(), main_network.parameters()):
        target_param.data.copy_(
            tau * main_param.data + (1.0 - tau) * target_param.data
        )
```

4. Benefits

1. Smoother training
2. More stable Q-values
3. Better convergence
4. Reduced oscillations
5. Continuous learning signal

5. Key Considerations

1. Choice of
   - Too high: unstable
   - Too low: slow learning
   - Typical range: 0.001 - 0.01

2. Update frequency
    - Every step vs batch
    - Computational overhead
    - Memory requirements

Remember: Soft updates are a key innovation in DDPG that helps maintain stability during training by preventing sudden changes in target values.

Deep RL for Finance

Deep Reinforcement Learning in finance is like having an AI trader that learns from market patterns and its own trading decisions. Instead of following fixed rules, it adapts to changing market conditions by learning from experience, optimizing trading strategies, portfolio management, and risk assessment in real-time.

In practical applications, DRL agents can handle complex financial tasks like portfolio optimization (deciding what mix of assets to hold), market making (providing buy/sell quotes), and algorithmic trading (executing large orders efficiently). The agent learns to balance multiple objectives like maximizing returns while managing risks, transaction costs, and market impact. For example, in portfolio management, the state might include market prices, volumes, and economic indicators; actions would be portfolio weights; and rewards would be risk-adjusted returns.

However, financial applications face unique challenges: highly stochastic environments (markets are noisy and unpredictable), non-stationary data (market patterns change over time), and sparse rewards (feedback only comes with price changes). Modern approaches tackle these using techniques like adversarial training to handle market uncertainty, multi-agent systems to model market interactions, and hierarchical RL to break down complex trading strategies into manageable sub-tasks. Common algorithms include A2C for high-frequency trading, DDPG for portfolio management, and PPO for order execution.

Advantages of RL for Trading and Investment - Notes

Simple Overview

RL offers unique advantages in trading by learning to adapt to market conditions without explicit programming. For short-term trading, it can identify and exploit micro-patterns in market data, while for long-term investment, it can learn complex relationships between economic factors and asset performance.

Short-Term Trading Advantages

1. **Real-Time Decision Making**
    - Microsecond reaction times
    - Pattern recognition in tick data
    - Dynamic order book analysis
    - Adaptive to market microstructure
2. **Risk Management**

```
Risk_adjusted_return = Return -   × Risk
```

    - Dynamic position sizing
    - Stop-loss optimization
    - Volatility adaptation
    - Transaction cost minimization
3. **Market Making**
    - Optimal bid-ask spreads
    - Inventory management
    - Queue position optimization

- Adverse selection avoidance

Long-Term Investment Advantages

1. **Portfolio Optimization**

```
Portfolio_objective = E[Return] -   × Variance -   × Transaction_costs
```

- Asset allocation
- Risk factor balancing
- Rebalancing optimization
- Tax-loss harvesting

2. **Macro Analysis**
   - Economic indicator integration
   - Cross-asset correlations
   - Regime change detection
   - Long-term trend identification

Common Benefits

1. **Adaptability**
   - Market regime changes
   - New asset classes
   - Changing regulations
   - Economic cycles
2. **Objectivity**
   - Emotion-free decisions
   - Consistent strategy execution
   - Backtesting capabilities
   - Performance attribution
3. **Scalability**
   - Multi-market analysis
   - 24/7 operation
   - Large data processing
   - Strategy combination

Remember: While RL offers powerful advantages, it requires careful implementation, robust risk management, and continuous monitoring to be effective in real-world financial applications.

Optimal Liquidation Problem and Solutions - Notes

Simple Overview

The optimal liquidation problem is like selling a large house without crashing local property prices: you need to balance speed of sale against market impact. In trading, it means breaking up large orders into smaller ones to minimize price impact while managing timing risk. The goal is to find the best trade-off between urgency and market impact costs.

Mathematical Formulation

1. **Basic Objective Function**

```
min E[ (Implementation_Shortfall + Risk_Penalty)]
```

2. **Implementation Shortfall**

```
IS =  (P_t - P_0)v_t +  (v_t)
```

Where: - P_t: execution price - P_0: initial price - v_t: volume at time t - : market impact function

Common Solutions

1. Static Strategies

1. **VWAP (Volume-Weighted Average Price)**

```
v_t = V × (Volume_t/Total_Volume)
```

2. **TWAP (Time-Weighted Average Price)**

```
v_t = V/T
```

3. **Almgren-Chriss Model**
   - Optimal trading trajectory
   - Balance between impact and risk
   - Assumes linear permanent impact

2. Dynamic Strategies (RL-Based)

1. **State Space**
   - Remaining shares
   - Time left
   - Market conditions
   - Order book state
2. **Action Space**
   - Trading rate
   - Order size
   - Order type selection
   - Venue selection
3. **Reward Function**

```
R = -(Price_Impact + Timing_Risk + Transaction_Costs)
```

Implementation Considerations

1. **Market Impact**
   - Temporary impact
   - Permanent impact
   - Non-linear effects
   - Cross-asset impact
2. **Risk Management**
   - Price volatility
   - Liquidity risk
   - Execution uncertainty
   - Information leakage
3. **Practical Constraints**
   - Trading hours
   - Minimum tick size
   - Lot sizes
   - Trading halts

Modern Approaches

1. **Deep RL Solutions**
   - Adaptive to market conditions
   - Learn from historical data

- Real-time optimization
- Multi-asset consideration

2. **Multi-Agent Systems**
   - Game-theoretic approach
   - Strategic interaction
   - Market maker interaction
   - Competitive behavior
3. **Hybrid Methods**
   - Combine classical models with RL
   - Incorporate market microstructure
   - Adaptive parameters
   - Risk-aware execution

Remember: The optimal solution depends heavily on market conditions, asset characteristics, and specific trading constraints. Regular recalibration and monitoring are essential.

Almgren-Chriss Model - Notes

The Almgren-Chriss model is like planning a careful exit from a crowded room: it provides a mathematical framework for optimal trading execution that balances the trade-off between market impact (disturbing prices) and timing risk (price uncertainty). It's a foundational model in algorithmic trading that gives a closed-form solution for optimal trading trajectories.

Mathematical Framework

1. Core Components

1. **Trading Trajectory**

```
x(t): shares remaining at time t
v(t) = -dx/dt: trading rate
```

2. **Price Impact Model**

```
S(t) = S  +  B(t) +  (x(t)-x ) +  v(t)
```

Where: - S : initial price - : volatility - : permanent impact - : temporary impact

2. Optimization Problem

1. **Objective Function**

```
min E[C] +  Var[C]
```

Where: - C: total trading cost - : risk aversion parameter

2. **Trading Schedule**

```
x(t) = x  cosh( (T-t))/cosh( T)
```

Where: - = √(²/) - T: liquidation horizon

Key Features

1. **Assumptions**
   - Linear price impact
   - Normal price distribution
   - Continuous trading
   - Risk-variance trade-off

Alt text

Figure 10: Alt text

2. **Parameters**
    - Market impact (,)
    - Volatility ( )
    - Risk aversion ( )
    - Time horizon (T)

Practical Implementation

1. **Trading Strategy**

```
def optimal_trading_rate(shares_left, time_left, params):
     = np.sqrt(params.risk_aversion * params.volatility**2 / params.temp_impact)
   return shares_left *   * np.tanh(  * time_left)
```

2. **Risk Management**
    - Position monitoring
    - Parameter estimation
    - Impact measurement
    - Cost analysis

Extensions and Modifications

1. **Modern Adaptations**
    - Non-linear impact
    - Stochastic volatility
    - Adaptive parameters
    - Multiple assets
2. **Integration with ML**
    - Parameter learning
    - Market regime detection
    - Dynamic adjustment
    - Real-time optimization

Remember: While the model provides elegant mathematical solutions, real-world implementation requires careful parameter estimation and regular recalibration based on market conditions.
