0.1 Understanding PPO Calculations and the Value Network

Let's break down the calculations involved in a PPO training step and clarify how the value network is structured within AutoModelForCausalLMWithValueHead.
How PPO Works: The Calculation Steps

PPO is an iterative algorithm. In each training iteration (or "step" in the context of the PPOTrainer), the goal is to update the model's policy and value function based on a batch of collected data. Here are the key calculations involved [1]:

1. **Data Collection (Rollout)**:
    - The current policy (the AutoModelForCausalLM part) is used to generate a sequence of actions (tokens) given an initial state (the query). This generates a trajectory of (state, action) pairs.
    - For each action taken, a reward is received from the environment (in this case, the sentiment score).
    - This process is repeated for a batch of initial queries to collect a set of trajectories and corresponding rewards.
2. **Calculating the Advantage**:
    - The advantage of taking a specific action in a specific state is an estimate of how much better that action is than the average expected outcome from that state [1].
    - The AutoModelForCausalLMWithValueHead's value network provides an estimate of the expected cumulative reward from a given state, denoted as $V(s)$.
    - The actual cumulative reward obtained from a state can be estimated using the collected rewards from the trajectory. A common method is using Generalized Advantage Estimation (GAE), which balances bias and variance in the advantage estimate. The formula for GAE involves a discount factor γ and a smoothing parameter,λ, which is

$$\hat{A}_t = \sum_{i=t}^{T-1} (\gamma\lambda)^{i-t}\delta_i$$

, where $\delta_i = r_i + \gamma V_\theta(s_{i+1}) - V_\theta(s_i)$ is the temporal difference error. if λ is set to 1, it can be simplified to:

$$A_t = \sum_{k=0}^{n-t} (\gamma)^k r_{t+k} + (\gamma)^{n-t+1} V(s_{n+1}) - V(s_t)$$

where A_t is the advantage at time step t, n is $T-1$, r_{t+k} is the reward at time step $t+k$, $V(s)$ is the value estimate for state s, γ is the discount factor for future rewards, and λ is the GAE parameter. The sum $\sum_{k=0}^{n-t}(\gamma\lambda)^k r_{t+k}$ accumulates the discounted rewards from time step t up to time step n.

The term $(\gamma\lambda)^{n-t+1}V(s_{n+1})$ is a bootstrapped estimate of the value of the state following the last state for which you have a direct reward in the sum. Here, s_{n+1} is the state reached after taking action a_n in state s_n.

Where does $V(s_{n+1})$ come from?
$V(s_{n+1})$ is the value predicted by the value network for state s_{n+1}. During the rollout phase, when the agent reaches state $s_n$and takes action a_n, it transitions to state s_{n+1}. Even though you might stop collecting rewards for this specific segment at s_n, you typically still compute the value prediction $\hat{V}(s_{n+1})$using the old value network (the one with parameters ϕ_{old} from before the PPO update). This $\hat{V}(s_{n+1})$ is then used as a "terminal value" or "bootstrap value" in the GAE calculation to estimate the value of the trajectory segment. It essentially serves as a stand-in for the expected future discounted rewards that would be obtained from state s_{n+1} onwards, according to the current value network's understanding.

Why include $V(s_{n+1})$?
Including the value estimate $V(s_{n+1})$ helps to reduce the variance of the advantage estimate [1]. If you only summed the collected rewards up to s_n and didn't include $V(s_{n+1})$, the advantage estimate would be purely based on the limited rewards collected in that specific segment. This can be noisy, especially if the segment is short. By including the value network's prediction for the subsequent state, you are incorporating a learned estimate of the value of the rest of the potential trajectory. This helps to smooth out the advantage signal and provides a more stable target for updating the policy.
In essence, GAE uses a combination of actual collected rewards within a segment and a learned value function estimate for the end of that segment to get a better, lower-variance estimate of the advantage. $V(s_{n+1})$ is that learned estimate for the state immediately following the last state with a directly summed reward in the GAE calculation.

- The PPOTrainer handles these calculations internally.

3. **Constructing the Clipped PPO Objectiv**e:

   - The core of PPO is its clipped objective function. For each (state, action) pair in the collected data, the objective aims to maximize the following: $L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$ where:
     - θ are the parameters of the policy network.
     - $\hat{E}_t$ denotes the empirical expectation over the samples in the batch.
     - $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the ratio of the probability of taking action a_t in state s_t under the new policy π_θ compared to the old policy $\pi_{\theta_{\text{old}}}$ (the policy before the update).
     - $\hat{A}_t$ is the advantage estimate calculated in step 2.

- $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$ clips the probability ratio to be within the interval $[1-\epsilon, 1+\epsilon]$, where ϵ is a small hyperparameter (e.g., 0.1 or 0.2).

- The objective function encourages increasing the probability of actions with positive advantage ($\hat{A}_t > 0$) and decreasing the probability of actions with negative advantage ($\hat{A}_t < 0$).
- The clipping mechanism is crucial:
  - If the advantage is positive, the objective is simply $r_t(\theta)\hat{A}_t$, encouraging the new policy to be more likely to take that action. However, the clipping ensures that if $r_t(\theta)$ goes above $1+\epsilon$, the objective is capped at $(1+\epsilon)\hat{A}_t$, preventing excessively large policy updates.
  - If the advantage is negative, the objective is $r_t(\theta)\hat{A}_t$. The clipping ensures that if $r_t(\theta)$ goes below $1-\epsilon$, the objective is floored at $(1-\epsilon)\hat{A}_t$, preventing the policy from becoming drastically less likely to take that action.
  - The min operation effectively chooses the lower of the two terms, enforcing the clipping constraint.

4. **Adding an Entropy Bonus (Optional but common)**:

   - An entropy bonus can be added to the objective function to encourage exploration. This term is proportional to the entropy of the policy distribution, which measures the randomness of the policy. Maximizing entropy encourages the policy to assign non-zero probabilities to a wider range of actions.

   $$S[\pi_\theta](s_t) = -\sum_{a_t} \pi_\theta(a_t|s_t) \log \pi_\theta(a_t|s_t)$$

   This formula defines the entropy bonus, which encourages the policy to explore different actions.$\pi_\theta(a_t|s_t)$ is the probability of taking action a_t in state s_t under the policy π_θ

5. **Total PPO objective**:

   - The modified objective becomes:

   $$L^{CLIP+V+S}(\theta) = L^{CLIP}(\theta) - c_1 L^{VF}(\phi) + c_2 S[\pi_\theta](s_t)$$

   where $L^{VF}(\phi)$ is a value function error term (see step 5), S is the entropy bonus, and c_1 and c_2 are coefficients.

   - It is typically presented as an objective to be maximized.
   - A higher value for $L^{CLIP}(\theta)$ indicates that the new policy is improving performance (based on positive advantages) or avoiding significant degradation (when advantages are negative) without making drastic changes. So, we add this term to the objective we want to maximize.
   - $L^{VF}(\theta)$ is a loss function, specifically a squared error. Its purpose is to measure the discrepancy between the predicted value and the target value. We want to minimize this loss. If we are formulating

the overall objective as a maximization problem, we want to reduce the negative impact of this loss on the total objective. Minimizing a loss L is equivalent to maximizing $-L$. Therefore, to incorporate the minimization of the value loss into a maximization objective, we subtract the value loss term: $-c_1 L^{VF}(\theta)$. As $L^{VF}(\theta)$ decreases (gets closer to zero, which is minimal for a squared error), $-c_1 L^{VF}(\theta)$ increases (gets closer to zero), contributing positively to the overall maximization objective.

- The entropy $S(\pi_\theta)$ is a measure of the randomness of the policy. Higher entropy means more exploration, which is generally desirable. We want to maximize the entropy. Therefore, to incorporate the maximization of entropy into a maximization objective, we add the entropy term: $+c_2 S(\pi_\theta)$.

6. **Updating the Value Function**:

   - The value network parameters ϕ are updated to minimize the difference between the predicted value $V_\phi(s_t)$ and the estimated return from state s_t (usually the GAE estimate plus the predicted value at the end of the truncated trajectory, or a simple Monte Carlo return). This predicted value is from a reference value network (or a previous version of the value network)
   - A common loss function for the value network is the squared error: $L^{VF}(\phi) = \hat{E}_t[(V_\phi(s_t) - \hat{R}_t)^2]$, where $\hat{R}_t$ is the estimated return.

7. **Optimization Step**:

   - The overall objective (combining the clipped policy objective, the value function loss, and potentially the entropy bonus) is optimized using gradient descent (e.g., using an Adam optimizer) to update the parameters of both the policy network and the value network. This is typically done for multiple "epochs" over the collected batch of data [1].

These steps are repeated iteratively, with new data collected using the updated policy in each iteration, allowing the agent to learn and refine its behavior.

0.2 Delving Deeper: $\hat{E}_t$ and the Optimization Step in PPO

Let's clarify the meaning of $\hat{E}_t$ and then explore the optimization step in more detail, explaining how the two networks (policy and value) are used. What is $\hat{E}_t$?

$\hat{E}_t$ represents the empirical expectation over a batch of data collected at timestep t.

In the context of PPO, "timestep t" refers to a specific step in the reinforcement learning episode (a single sequence of states, actions, and rewards from the beginning to the end of a task, or until a certain condition is met). When we collect a "batch of data," we are running the current policy to generate several trajectories (sequences of states and actions).

So, $\hat{E}_t$ is not calculated using the two networks themselves, but rather it is an operator that indicates how to compute the average of a quantity over the data samples collected in a single PPO training iteration.

Here's a breakdown:

- Empirical Expectation ($\hat{E}$): In probability and statistics, the expected value $E[X]$ of a random variable X is the average value it would take over many trials. The empirical expectation $\hat{E}[X]$ is an approximation of the true expected value, calculated by averaging the observed values of X from a finite sample of data.
- Subscript t: In the PPO objective $\hat{E}_t[\dots]$, the subscript t signifies that the expectation is taken over the samples collected at (or rather, from) timestep t within the collected trajectories. More practically, it refers to averaging over the (state, action, advantage) pairs collected in the current batch of rollouts.

How it's used:

The PPO objective function [1] is defined as: $L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$

To compute this objective for a given batch of data, you perform the following:

1. For each (state, action) pair (s_i, a_i) in your collected batch:
   - Calculate the probability ratio $r_i(\theta) = \frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{\text{old}}}(a_i|s_i)}$ using the current policy network and the policy network from the previous iteration.
   - Calculate the advantage estimate $\hat{A}_i$ for this pair using the rewards and the value network's predictions.
   - Compute the clipped term: $\text{clip}(r_i(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_i$.
   - Take the minimum of the two terms: $\min(r_i(\theta)\hat{A}_i, \text{clip}(r_i(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_i)$.
2. Average the results from step 1 over all the (state, action) pairs in your batch. This average is the value of the empirical expectation $\hat{E}_t[\dots]$.

So, the two networks (AutoModelForCausalLM for the policy and the WithValueHead for the value function) are used within the terms being averaged, specifically to calculate the probability ratios $r_t(\theta)$ and the advantage estimates $\hat{A}_t$. The $\hat{E}_t$ operation itself is simply the mathematical process of averaging over the batch.

The Optimization Step in Detail

The optimization step in PPO is where the model's parameters are updated to improve the policy and value function [1]. This is typically done using a gradient-based optimizer like Adam.

Here's a more detailed look:

1. Define the Full Loss Function: The actual loss function that is minimized during the optimization step usually combines the clipped policy objective, the value function loss, and potentially an entropy bonus: $L(\theta, \phi) = \hat{E}_t[L^{CLIP}(\theta) - c_1 L^{VF}(\phi) + c_2 S[redactedlink]]$ where:
   - $L^{CLIP}(\theta)$ is the clipped policy objective we discussed, which depends on the policy parameters θ.

- $L^{VF}(\phi) = (V_\phi(s_t) - \hat{R}_t)^2$ is the value function loss, depending on the value function parameters ϕ. The target return $\hat{R}_t$ is usually derived from the collected rewards and potentially the value function itself (as in GAE).
- $S[redactedlink]$ is the entropy of the policy at state s_t, depending on the policy parameters θ.
- c_1 and c_2 are hyperparameters that control the weighting of the value loss and entropy bonus, respectively.

2. Compute Gradients: Using the collected batch of data, the gradients of the loss function $L(\theta, \phi)$ are computed with respect to the model's parameters (θ for the policy and ϕ for the value function). This is typically done using backpropagation through the neural network architecture.
    - The policy gradient is derived from the L^{CLIP} and S terms.
    - The value gradient is derived from the L^{VF} term.
3. Optimization Algorithm: An optimization algorithm (like Adam, used in the PPOTrainer) takes these gradients and updates the model's parameters. The update rule for a parameter ω (which can be θ or ϕ) using a generic gradient descent approach is: $\omega \leftarrow \omega - \alpha \nabla_\omega L(\theta, \phi)$ where α is the learning rate. Adam is a more sophisticated optimization algorithm that uses adaptive learning rates and momentum, but the core idea is still to move the parameters in the direction that reduces the loss.
4. Multiple Epochs over the Same Batch: Unlike some other algorithms that update after seeing each data point, PPO typically performs multiple optimization epochs over the same batch of collected data [1]. This means the gradients are computed and parameters are updated multiple times using the same set of trajectories collected in the initial rollout phase of that PPO iteration. This allows for more efficient use of the data but requires careful tuning of the number of epochs to avoid overfitting to the current batch.
5. Separate vs. Joint Optimization: In AutoModelForCausalLMWithValueHead, the policy and value networks share the initial layers (the AutoModelForCausalLM part). The gradients for both the policy loss and the value loss flow back through these shared layers, leading to a joint optimization of the shared parameters. The value head's specific layers only receive gradients from the value loss. This shared representation can be beneficial for learning, as features useful for predicting the next token might also be useful for estimating the value of a state.

In summary, the optimization step in PPO involves calculating the combined loss function over a batch of data, computing the gradients of this loss with respect to the policy and value network parameters, and then using an optimizer to update these parameters to reduce the loss. The unique aspect of PPO's optimization is the clipped objective, which helps to keep the policy updates stable and prevent large divergences from the previous policy. The AutoModelForCausalLMWithValueHead facilitates this by providing the necessary outputs (policy probabilities and value estimates) for computing the loss and gradients.

0.3 How the target of the value network loss function is created

You're asking about the target for the value network loss function. If the predicted value $\hat{V}(s_t)$ comes from the value network, where does the estimated return from a state at t come from, which is used as the target?

This is a core concept in training value functions in reinforcement learning, particularly in actor-critic methods like PPO. The estimated return (the target) is typically a combination of actual rewards collected during the interaction with the environment and value estimates from the current value network, but applied to future states.

Here are the common ways the target for the value network loss (often denoted as $V_{\text{target}}(s_t)$ is constructed:

- **Monte Carlo Return (High Variance, Low Bias)**:
  $V_{\text{target}}(s_t) = G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$ Here, G_t is the actual discounted sum of rewards from state s_t until the end of the episode (or truncated trajectory at timestep T). This is the "true" return observed in the collected data. The value network is trained to minimize the difference between $\hat{V}(s_t)$ and G_t. This target is unbiased but has high variance because it depends on the entire future trajectory.

- **One-Step TD Target (Low Variance, High Bias)**:
  $V_{\text{target}}(s_t) = r_t + \gamma \hat{V}(s_{t+1})$ This target uses the immediate reward r_t and the value estimate of the next state $\hat{V}(s_{t+1})$, as predicted by the current value network. The value network is trained to minimize the difference between $\hat{V}(s_t)$ and this TD target. This target has lower variance because it only depends on the immediate reward and the next state's value estimate, but it is biased by any inaccuracies in the value function itself at s_{t+1}.

- **N-Step TD Target (Bias-Variance Trade-off)**:
  $V_{\text{target}}(s_t) = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \hat{V}(s_{t+n})$ This target uses the sum of actual rewards for the next n steps and the value estimate of the state n steps into the future, $\hat{V}(s_{t+n})$, predicted by the current value network. This provides a balance between the Monte Carlo return and the one-step TD target.

- **GAE Return (Used in PPO)**:
  In PPO, the advantage is typically calculated using GAE. The target for the value network is often derived from the GAE advantage and the current value estimate: $V_{\text{target}}(s_t) = \hat{V}(s_t) + \hat{A}t^{\text{GAE}(\gamma,\lambda)}$ This target is essentially an estimate of the return based on the current value prediction plus the estimated advantage of the action taken from that state. Alternatively, the target can be the bootstrapped GAE return estimate: $V\text{target}(s_t) = \sum_{k=0}^{n-t-1} (\gamma\lambda)^k r_{t+k} + (\gamma\lambda)^{n-t} \hat{V}(s_n)$ (using the formula we discussed earlier)

Where do $\hat{V}(s_{t+1})$, $\hat{V}(s_{t+n})$ etc., come from?

Crucially, these future value estimates are produced by the same current value network that produced $\hat{V}(s_t)$.

During the data collection phase, for each state s visited in the trajectory (including s_t, s_{t+1}, ..., s_n), the current value network is run to produce its value estimate $\hat{V}(s)$. These stored value estimates are then used in the offline calculation of the target values for the value loss function.

In the context of the notebook's PPO implementation:

The PPO trainer likely uses a form of GAE. The target for the value network loss will be based on the rewards collected during the generation step and the value estimates for future states produced by the current value network. The value loss will then be calculated as the difference between the value network's prediction for a state $\hat{V}(s_t)$ and the calculated target value for that state.

The key idea is that the value network is being trained to make its predictions $\hat{V}(s)$ consistent with the observed rewards and the network's own estimates of future values. This bootstrapping process allows the network to learn to predict returns without having to wait until the very end of an episode (if there is one).

In the GAE return estimate formula you mentioned, $V_{\text{target}}(s_t) = \sum_{k=0}^{n-t-1} (\gamma\lambda)^k r_{t+k} + (\gamma\lambda)^{n-t}\hat{V}(s_n)$, the variable 'n' represents the end timestep of the finite trajectory or segment of a trajectory that is being used for calculating the GAE.

Here's a breakdown:

Trajectory Segment: In PPO, especially when dealing with continuous environments or very long episodes (like generating text), we often collect data in batches, which are essentially finite segments or "rollouts" of interaction with the environment. The trajectory starts at time t and ends at time n. The states and rewards collected are $s_t, r_t, s_{t+1}, r_{t+1}, ..., s_n, r_n$. The state after the last reward is s_{n+1}.

Sum from k=0 to n-t-1: The sum $\sum_{k=0}^{n-t-1} (\gamma\lambda)^k r_{t+k}$ is a weighted sum of the rewards collected from timestep t up to timestep $n-1$. The index k represents the time step relative to the starting time t. So, $k = 0$ corresponds to reward r_t, $k = 1$ corresponds to reward r_{t+1}, and $k = n-t-1$ corresponds to reward $r_{n-t-1+t} = r_{n-1}$. The rewards are discounted by $(\gamma\lambda)^k$. Why n-t-1? The sum goes up to $n-t-1$ because the reward r_n is typically associated with the transition from state s_n to s_{n+1}. In this formulation, the terminal value term $(\gamma\lambda)^{n-t}\hat{V}(s_n)$ is used to "bootstrap" from the value estimate of the state s_n, which is the last state where an action led to a reward within the collected segment.

$(\gamma\lambda)^{n-t}\hat{V}(s_n)$: This is the terminal value term. It represents the discounted value estimate of the state s_n, which is the last state in the collected trajectory segment where a reward r_n was received leading to s_{n+1}. The discount factor $(\gamma\lambda)^{n-t}$ reflects the time steps from the starting state s_t to the state s_n.

In the context of the notebook:

When the ppo_trainer.step function is called, it operates on a batch of data that represents a collection of short trajectories or segments of text generation.

t would be the starting timestep of a particular segment (often 0 within that segment). n would be the last timestep in that segment where a reward was obtained (corresponding to the last generated token that contributed to the sentiment score).

The PPO trainer collects a batch of queries and generated responses. For each query/response pair, this forms a short trajectory segment. The rewards are calculated based on the sentiment of the full generated response. The GAE calculation then uses these rewards and the value estimates of the states within that segment to compute the advantage for each timestep. The 'n' in the formula would be the length of the generated response minus one (since rewards are often associated with the transition to the next state).

In summary:

'n' in the GAE target formula represents the end timestep of the finite trajectory segment used for the calculation. The sum goes up to $n - t - 1$ to include the rewards up to the second-to-last state of the segment, and the terminal value term uses the value estimate of the last state in the segment to account for the future expected rewards beyond the collected data. This truncated sum combined with the terminal value estimate forms the GAE target for training the value network.

0.4 Summary

When the causal language model generates a sequence, it does so one token at a time:

Start with initial query: s_0 (e.g., "The movie was")

Step 1:

1. Input to model: s_0 ("The movie was")
2. Causal LM processes s_0.
3. Value Head processes the hidden state(s) corresponding to s_0 and predicts $\hat{V}(s_0)$.
4. Policy Head predicts the next token based on s_0. Let's say it generates token a_0 ("really").
5. New state is $s_1 = s_0 + a_0$ ("The movie was really").

Step 2:

1. Input to model: s_1 ("The movie was really")
2. Causal LM processes s_1.
3. Value Head processes the hidden state(s) corresponding to s_1 and predicts $\hat{V}(s_1)$.
4. Policy Head predicts the next token based on s_1. Let's say it generates token a_1 ("good").
5. New state is $s_2 = s_1 + a_1$("The movie was really good").

This continues until the end of the generated sequence (s_n).

What the value network uses is the internal representation of the last token in the sequence at that particular step as the input for its prediction.

So, for predicting $\hat{V}(s_t)$, the value network takes the hidden state output of the causal language model after it has processed the sequence s_t. This hidden state is the representation of the "final state" at that specific time t.

When the model moves to the next step and the state becomes s_{t+1}, the entire new sequence s_{t+1} is fed into the model. The causal language model processes this new, longer sequence, produces a new set of hidden states, and the hidden state corresponding to the last token in s_{t+1} is then fed to the value head to predict $\hat{V}(s_{t+1})$.

Therefore, the value network is using the representation of the "final state" of the sequence, but it's doing so sequentially for each state $(s_t, s_{t+1}, \ldots, s_n)$ that occurs during the generation process. It doesn't just use the final state of the entire generated response to predict a single value for the whole response; it predicts a value for each prefix of the response as it's being built. These per-step value predictions are crucial for algorithms like PPO that use temporal difference methods (like GAE) to calculate advantages, which require comparing value estimates across consecutive states.

In a standard PPO implementation for text generation (like the one likely used in the trl library for this notebook), the optimization step does not begin until a batch of entire sequences (or truncated sequences) has been generated during the rollout phase.

Here's why this is necessary:

1. Calculating Rewards: The reward function (the sentiment analysis pipeline in this case) typically evaluates the entire generated response or a significant portion of it to assign a reward. You need the full context of the response to determine its sentiment. You can't calculate the final reward after generating just one token.

2. Calculating Advantages: Advantage estimation methods like GAE require knowing the sequence of rewards and value predictions over a segment of a trajectory. While you get value predictions $\hat{V}(s_t)$ at each step during the rollout, you need the rewards that occur after those states to compute the advantages [1]. Since the sentiment reward is typically at the end of the response, you need the full response to get that reward signal.

3. Batch Processing for Efficiency: Neural network training is significantly more efficient when performed in batches. Collecting a batch of full trajectories allows for vectorized operations and parallel processing on GPUs. Optimizing after each token generation would be computationally very expensive and inefficient.

4. Policy Updates: The PPO algorithm updates the policy parameters based on the clipped objective, which involves the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$. To calculate this ratio accurately, you need the probabilities of the actions taken during the rollout under both the old policy (which generated the data) and the current policy (which you are updating). These probabilities are readily available from the data collected during the rollout.

The workflow for one PPO iteration looks like this:

1. Start of Iteration: You have the current parameters of your policy and value networks ($\theta_{\text{old}}, \phi_{\text{old}}$.
2. Rollout (Data Collection):
   (a) For each query in the batch:
      i. Use the policy based on θ_{old} to generate tokens sequentially, forming states $s_t, s_{t+1}, \ldots, s_n$ and actions $a_t, a_{t+1}, \ldots, a_n$.
      ii. At each step t, use the value network based on ϕ_{old} to predict $\hat{V}(s_t)$. Store $(s_t, a_t, \hat{V}(s_t))$.
      iii. Once the sequence is complete (or truncated), calculate the reward(s) for the trajectory. Store the rewards.
3. Optimization Phase (After Rollout):
   (a) Using the collected data (states, actions, rewards, old value predictions $\hat{V}(s_t)$):
   (b) Calculate advantages $\hat{A}_t$ for each time step.
   (c) Compute the policy loss (using the current policy parameters θ, the old probabilities from the rollout data, and the calculated advantages).
   (d) Compute the value loss (using the current value network parameters ϕ, the states s_t, and the target returns $\hat{R}_t$ derived from rewards and old value predictions).
   (e) Combine the losses.
   (f) Compute gradients of the combined loss with respect to θ and ϕ.
   (g) Use the optimizer (e.g., Adam) to update θ and ϕ.
   (h) Perform multiple optimization epochs on this same batch of data if configured to do so.
4. End of Iteration: The model has updated parameters. You are ready for the next PPO iteration, starting with a new rollout using the now-updated policy and value function.

So, the sequential prediction of $\hat{V}(s_t), \hat{V}(s_{t+1}), \ldots$ happens during the data collection phase to gather the necessary information for the subsequent batch optimization phase.

0.5 Summary of AutoModelForCausalLMWithValueHead

The causal language model, including the AutoModelForCausalLM part of AutoModelForCausalLMWithValu generates the response one token at a time.

This is the fundamental way that autoregressive language models work. They are trained to predict the probability distribution over the next token given the sequence of tokens generated so far.

Here's the typical process for generating a sequence:

1. Input the initial prompt/query: The model receives the initial sequence of tokens.

2. Predict the first token: The model processes the initial sequence and predicts the probability distribution for the token that should come next.

3. Sample/Choose the first token: Based on the predicted distribution (using techniques like sampling, greedy decoding, beam search, etc., which are controlled by parameters like do_sample, top_k, top_p in your generation_kwargs), a token is selected. This is the first token of the response.

4. Append the chosen token to the sequence: The newly chosen token is added to the end of the input sequence.

5. Predict the second token: The new, longer sequence (initial prompt + first generated token) is fed back into the model. The model processes this updated sequence and predicts the probability distribution for the next token.

6. Sample/Choose the second token: A token is selected from this new distribution. This is the second token of the response.

7. Repeat: Steps 4-6 are repeated until a stop condition is met, such as generating a special end-of-sequence token, reaching a maximum length (max_new_tokens), or some other criteria.

So, even though we talk about generating an "entire sequence" in the PPO rollout, this generation happens sequentially, token by token. At each step of this sequential generation process, the current state is the sequence of tokens generated so far, and the model makes a decision about the next token.

This is why the value network needs to be able to predict the value of these incrementally growing sequences $(s_t, s_{t+1}, s_{t+2}, \dots)$ during the rollout phase – because these are the states the agent is in at each step of its sequential action-taking process.

0.5.1 Using top_k or top_p

Using parameters like top_k or top_p introduces stochasticity (randomness) into the token generation process, but in a typical PPO rollout for collecting a batch of data, you still generate one response trajectory per initial query in the batch.

Here's how it works:

- Stochastic Sampling: When do_sample=True is used along with top_k, top_p, or a temperature parameter, the model doesn't just pick the single token with the highest probability predicted by the policy network. Instead, it samples from a modified probability distribution over the vocabulary.
  - top_k: The model only considers the k most likely next tokens and redistributes the probability mass among them before sampling.
  - top_p (nucleus sampling): The model considers the smallest set of most likely tokens whose cumulative probability exceeds p, and redistributes the probability mass among them before sampling.
- Single Trajectory per Query in Rollout: Even with stochastic sampling, for a given initial query in your batch, you still perform a single "rollout" or "episode." This rollout results in one specific sequence of generated tokens (one response).

  - At each step t, the policy network outputs a probability distribution over the vocabulary.
  - Based on this distribution and the sampling parameters (top_k, top_p, etc.), the model samples one token a_t.
  - This sampled token a_t becomes the action taken at state s_t.
  - The environment transitions to state s_{t+1} (the sequence s_t plus a_t). This sampled action a_t and the state s_t (along with its value prediction $\hat{V}(s_t)$ from the old value network) are recorded for the trajectory.

- Value Network and Optimization: The value network makes its predictions $\hat{V}(s_t)$ based on the actual sequence s_t that was generated during the rollout. The optimization process then uses the specific trajectory (the sequence of generated states s_t, the sampled actions a_t, the received rewards, and the old value predictions $\hat{V}(s_t)$) to calculate the policy and value losses and update the model parameters.

Why use top_k or top_p in PPO generation?

Using these stochastic sampling methods during the rollout phase is common and beneficial in PPO for text generation because it encourages exploration.

- If the model always picked the single most likely token (greedy decoding), it would follow a very narrow path. This might lead to getting stuck in local optima and producing repetitive or less creative text.
- Stochastic sampling allows the policy to explore a wider range of possible responses, even those that might have slightly lower probabilities initially but could potentially lead to higher rewards in the long run.
- By exploring different sequences, the agent gathers more diverse data for the optimization process, which can lead to a more robust and better-performing policy.

In summary:

Using top_k or top_p makes the process of generating each token within a single response probabilistic. However, for each initial query in your batch, the PPO rollout still produces just one complete sequence trajectory. This single trajectory, with its sequence of states, sampled actions, corresponding old value predictions, and eventual rewards, is then used as a data point for the batch optimization step. You don't generate multiple full responses for each query and then somehow average or select among them for the PPO update; you generate one response trajectory stochastically and learn from that specific sequence of events.