DPO process

The model doesn't inherently "know" it's predicting a preferred or rejected response in the same way a human does. Instead, during Direct Preference Optimization (DPO) training, the model learns to assign higher probabilities to the sequences of tokens that constitute the "chosen" responses compared to the sequences of tokens that constitute the "rejected" responses, given the same prompt.

Here's how it works based on the DPO loss function we discussed:
====> **Data Input**: The training data provides pairs of responses for each prompt: a "chosen" response (which is preferred by a human or a reward model) and a "rejected" response (which is not preferred).
====> **Model Probability**: For a given prompt, both the chosen response (y_w) and the rejected response (y_l) are fed through the model (with parameters θ). The model calculates the probability of generating the sequence of tokens for each response, $p_\theta(y_w|x)$ and $p_\theta(y_l|x)$.
====> **Reference Model Probability**: Similarly, the probabilities of the chosen and rejected responses are calculated using a reference model ($p_{\text{ref}}(y_w|x)$ and $p_{\text{ref}}(y_l|x)$). The reference model is typically a frozen version of the initial language model before DPO training. This helps in regularizing the training and preventing the model from drifting too far from its original capabilities.
====> **Log-Probability Ratios**: The core of the DPO loss involves the difference between the log-probability ratios of the chosen and rejected responses for the fine-tuned model relative to the reference model: $\log \frac{p_\theta(y_w|x)}{p_{\text{ref}}(y_w|x)} - \log \frac{p_\theta(y_l|x)}{p_{\text{ref}}(y_l|x)}$ This can be rewritten as: $\log \frac{p_\theta(y_w|x)}{p_\theta(y_l|x)} - \log \frac{p_{\text{ref}}(y_w|x)}{p_{\text{ref}}(y_l|x)}$ This term essentially measures how much the fine-tuned model's preference for the chosen response over the rejected response has increased compared to the reference model's preference.
====> **DPO Loss Calculation**: This difference in log-probability ratios is then scaled by the β= parameter and passed through a sigmoid function. The negative log of this result is the DPO loss. The loss is designed such that it is lower when the fine-tuned model assigns a significantly higher probability to the chosen response compared to the rejected response (relative to the reference model).
====> **Gradient Descent**: The training process uses gradient descent to update the

model's parameters (θ) to minimize this DPO loss. By minimizing the loss, the model is effectively trained to:
=========>Increase the probability of generating the "chosen" responses.
=========> Decrease the probability of generating the "rejected" responses.

Therefore, the model learns through the optimization process to associate the patterns and characteristics of the "chosen" responses with higher likelihoods of generation, and the patterns of "rejected" responses with lower likelihoods. When you then use the trained DPO model to generate a response for a new prompt, it will tend to generate sequences that are similar in style and content to the "chosen" responses it was trained on because those sequences now have higher predicted probabilities.

In essence, the model doesn't have a conscious understanding of "preferred" or "rejected"; it simply learns to reproduce the probabilistic distributions of the provided data, where the chosen responses are associated with higher rewards (lower loss) during training.

The effect of beta summarized:
=====>**High** β: Makes the loss function more sensitive to the difference in log-probabilities. Even a small preference for the rejected response will result in a very high loss, pushing the model strongly to maximize the difference between the chosen and rejected log-probabilities. This can lead to more aggressive optimization and potentially more polarized responses (strongly favoring chosen). However, too high a beta can make training unstable or lead to overfitting.
=====>**Low** β: Makes the loss function less sensitive. The model is less heavily penalized for assigning similar probabilities to chosen and rejected responses. This leads to more gentle optimization and potentially less deviation from the reference model. It can result in responses that are closer to the original model's style but might not strongly reflect the human preferences.

Analogy (Temperature):
The term "temperature" is used because it's similar to the temperature parameter in softmax or other probability distributions. A high temperature makes the distribution

flatter (more uniform probabilities), while a low temperature makes it sharper (probabilities concentrated on a few options). In the DPO loss, a high beta "sharpens" the penalty for getting the preference wrong, while a low beta "smoothes" it out.

In the context of the code with beta=0.1, it's a relatively low value, suggesting a more gentle optimization process that might not aggressively push the model towards extreme preference for the chosen responses. This can be beneficial for stability and preventing overfitting, especially with limited data.

Choosing the right beta value is often a hyperparameter tuning task, depending on the dataset and the desired strength of the preference alignment.

Process in Detail

Let's refine the process:

====> **Prompt Input**: The prompt x is fed into the model.

====> **Calculating the Probability of the Chosen Response (y_w)**:

=========>The model generates a probability distribution over the next token based on the prompt x. Let's call the first token of the chosen response $y_{w,1}$. The model calculates the probability $P(y_{w,1}|x)$.

=========>Then, the prompt x and the first token $y_{w,1}$ are fed back into the model to generate a probability distribution over the next token. The model calculates the probability of the second token of the chosen response, $y_{w,2}$, given the prompt and the first token: $P(y_{w,2}|x, y_{w,1})$.

=========>This process continues for every token in the chosen response sequence. For the i-th token, the model calculates $P(y_{w,i}|x, y_{w,1}, \ldots, y_{w,i-1})$.

=========>The probability of the entire chosen response sequence, $p_\theta(y_w|x)$, is the product of the probabilities of each token given the preceding tokens and the prompt: $p_\theta(y_w|x) = P(y_{w,1}|x) \times P(y_{w,2}|x, y_{w,1}) \times \cdots \times P(y_{w,n}|x, y_{w,1}, \ldots, y_{w,n-1})$ where n is the length of the chosen response sequence. In practice, it's often calculated in the log-space to avoid numerical underflow: $\log p_\theta(y_w|x) = \sum_{i=1}^{n} \log P(y_{w,i}|x, y_{w,1}, \ldots, y_{w,i-1})$

....

====> **Calculating the Probability of the Rejected Response (y_l)**:

A completely separate process is performed for the rejected response y_l. The model calculates the probability of the entire rejected response sequence given the same prompt x, in the same token-by-token manner: $\log p_\theta(y_l|x) = \sum_{j=1}^{m} \log P(y_{l,j}|x, y_{l,1}, \ldots, y_{l,j-1})$ where m is the length of the rejected response sequence.

So, it's not comparing a single probability distribution to the chosen/rejected responses. Instead, it's calculating the joint probability of the entire sequence of the chosen response and the joint probability of the entire sequence of the rejected response, both conditioned on the prompt. These sequence probabilities are then used in the DPO loss calculation.

The term "probability distribution of the sequence" can be a bit misleading here. The model predicts a probability distribution over the next possible tokens at each step. The probability of the entire sequence is derived by multiplying the probabilities of each token in the sequence, conditioned on the prompt and the preceding tokens. This is a standard way to calculate the probability of a sequence in autoregressive language models.

Model Process
let's delve into how a language model, like the GPT-2 model used in the code, calculates the probability of the first token of a response, $P(y_{w,1}|x)$, given only the prompt x.

This is a fundamental operation in autoregressive language models. Here's the general idea:
====> **Tokenization and Embedding**:
=========>The input prompt x is first tokenized into a sequence of numerical IDs representing the tokens in the prompt.
=========>These token IDs are then converted into numerical vectors called embeddings. These embeddings capture some initial semantic meaning of the tokens.
====> **Model Forward Pass (with Prompt)**:
=========>The sequence of prompt embeddings is fed through the model's layers (e.g., transformer blocks in GPT-2).
=========>These layers process the embeddings, applying complex mathematical operations (attention mechanisms, feed-forward networks, etc.). The goal is to integrate

the information from the entire prompt and build a rich representation of the context provided by the prompt.
====> **Output Layer and Logits**:
=========>After passing through the main layers, the model produces an output representation for the end of the prompt sequence. This output representation is then passed through a final layer (often a linear layer).
=========>This final layer outputs a vector of raw scores, called logits. The length of this vector is equal to the size of the model's vocabulary. Each logit corresponds to a specific token in the vocabulary. Higher logits suggest that the corresponding token is more likely to appear next.
====> **Softmax Activation**:
=========>The vector of logits is then passed through a softmax function.
=========>The softmax function converts the logits into a probability distribution over the entire vocabulary. The output of the softmax is a vector where each element is a probability between 0 and 1, and the sum of all elements in the vector is 1. Each element represents the probability of the corresponding vocabulary token being the very next token after the prompt.
====> **Probability of the First Response Token**:
=========>The first token of the chosen response, $y_{w,1}$, is a specific token from the model's vocabulary.
=========>The value in the softmax output vector that corresponds to the token $y_{w,1}$ is precisely the probability $P(y_{w,1}|x)$.

In summary:

The model processes the prompt to understand the context. Based on this context, its final output layer, followed by a softmax function, produces a probability distribution over all possible next tokens. The probability of the specific token that happens to be the first token of the chosen response ($y_{w,1}$) is taken directly from this probability distribution.

This is the foundation for how autoregressive models predict the next token, and it's used iteratively to calculate the probability of subsequent tokens in the sequence, conditioned on the tokens that came before.