

CS 7650 Midterm Review (Modules 1-6)

Oct 15, 2026 · @Brian

What the exam emphasizes

The practice exam asks one or two questions per module, and most of them want an explanation of *why*, not just a definition. Expect a few select-all items, one hand computation (IDF), and several "explain the mechanism" short answers on LSTMs, Word2Vec, attention, and RLHF.

Module	Topic	Practice exam asks you to...
1	Intro to NLP	Pick out what makes language hard (ambiguity, non-literal meaning, new words)
2	Foundations	Know what Generalized Bayes computes and its independence assumption
3	Classification	Explain log-probabilities, softmax normalization and argmax shortcuts, and bag-of-words limits
4	Language models	Explain LSTM gating vs. vanishing gradients; define perplexity
5	Semantics	Contrast CBOW and Skip-Gram and their losses; compute IDF by hand
6	Modern neural architecture	Explain positional embeddings, cross-attention vs. masked self-attention, and RLHF reward models

Study order: read each module section, then cover the answers and try the practice exam cold.

Module 1: Intro to NLP

The exam point: language is hard because one surface form maps to many meanings and the vocabulary never stops growing.

Why NLP is hard

- **Lexical ambiguity / polysemy:** "bank" (river vs. money); domain-specific senses ("kernel" in OS vs. ML).
- **Syntactic ambiguity:** "I saw the man with the telescope": who has the telescope? (PP attachment)
- **Non-literal language:** metaphor, idiom, sarcasm ("kick the bucket").
- **Productivity:** new words, portmanteaus ("brunch", "doomscrolling"), compounding.
- **Context and pragmatics:** meaning depends on speaker, world knowledge, prior discourse; coreference ("it", "they").

Levels of analysis: morphology (word parts) → syntax (structure) → semantics (meaning) → pragmatics/discourse (meaning in context).

Pipeline vocabulary: tokenization, normalization (lowercasing, stemming vs. lemmatization), stop words, vocabulary V , out-of-vocabulary (OOV) words, sub-word tokens (BPE) as the modern fix for OOV.

Traps: absolute statements ("computers can't learn new words", "grammar can't be modeled statistically") are false. Statistical and neural NLP exists precisely because rule-based grammar alone fails.

Module 2: Foundations

The exam point: know how to flip a conditional with Bayes, and the conditional-independence assumption that makes Generalized Bayes tractable.

Probability rules to know cold

$$P(A, B) = P(A \mid B) \cdot P(B) = P(B \mid A) \cdot P(A)$$

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}, \quad \text{quad } P(B) = \sum_a P(B \mid a) \cdot P(a)$$

- **Prior** $P(A)$: belief before evidence. **Likelihood** $P(B|A)$: how well A explains the evidence. **Posterior** $P(A|B)$. **Evidence/normalizer** $P(B)$: same for every A , so it can be dropped when you only need the argmax.
- **Chain rule:** $P(w_1, \dots, w_n) = \prod P(w_i \mid w_1, \dots, w_{i-1})$. This is the basis of language modeling in Module 4.
- **Independence:** $P(A, B) = P(A)P(B)$. **Conditional independence:** $P(B_1, B_2 \mid A) = P(B_1|A)P(B_2|A)$.

Generalized Bayes: one cause, many observed effects. Assume the effects are conditionally independent given the cause, so the likelihood factors:

$$P(A \mid B_1, \dots, B_n) = \frac{P(A) \prod_i P(B_i \mid A)}{P(A) \prod_i P(B_i \mid A)}$$

That is exactly Naive Bayes with words as the effects and the class as the cause.

Neural-net basics that later modules assume

- Linear layer $z = Wx + b$. Nonlinearities: sigmoid, tanh, ReLU.
- Softmax turns scores into a distribution. Cross-entropy loss = $-\log P(\text{correct class})$.
- Training = gradient descent on the loss via backpropagation (the chain rule on derivatives).
- Dot product and cosine similarity measure vector similarity; you'll use them again in Modules 5 and 6.

Traps: "independent" is not "mutually exclusive". Mutually exclusive events are actually *dependent*, since one happening rules out the other. Nothing in Bayes needs normal distributions.

Module 3: Classification

The exam point: Naive Bayes vs. logistic regression on bag-of-words features, why we work in log space, and when normalization can be skipped.

Bag-of-words (BoW). A document becomes a $|V|$ -length vector of word counts (or 0/1 flags). It is simple and works surprisingly well for topic and sentiment. Downsides:

- No word order ("dog bites man" = "man bites dog").
- High-dimensional and sparse.
- No grammar.
- Unseen words are ignored.
- Synonyms sit in unrelated dimensions.

Naive Bayes classifier. Generalized Bayes with words as the effects:

$$\hat{y} = \arg\max_y \left[\log P(y) + \sum_{w \in d} \text{count}(w, d) \log P(w \mid y) \right]$$

- Estimate by counting: $P(w|y) = \text{count}(w, y) / \text{total words in class } y$.

- **Laplace (add-1) smoothing:** $(\text{count} + 1) / (\text{total} + |V|)$. Without it, one unseen word gives a probability of 0, and $\log 0 = -\infty$ wipes out the whole class.
- **Log space:** prevents underflow, turns products into sums, and preserves argmax because log is monotonic. $P(d)$ is dropped because it is the same for every class.

Logistic regression. A discriminative model that learns weights directly instead of counting.

- Binary: $P(y=1|x) = \sigma(w \cdot x + b)$, with sigmoid $\sigma(z) = 1/(1+e^{-z})$.
- Multinomial: one score per class, $z_k = w_k \cdot x + b_k$, followed by softmax.

$$P(y=k \mid x) = \frac{e^{z_k}}{\sum_j e^{z_j}}, \quad L = -\log P(y_{\text{true}} \mid x)$$

- **Argmax only?** Skip softmax and take the largest logit, since softmax is monotonic.
- **Sampling or real probabilities?** You need the normalizer $\sum e^{z_j}$.
- Trained by gradient descent on cross-entropy. Neural classifiers stack hidden layers before the softmax.

	Naive Bayes	Logistic regression
Type	Generative: models $P(x y)P(y)$	Discriminative: models $P(y x)$ directly
Training	Count + smooth (closed form)	Gradient descent
Feature independence	Assumed	Not assumed; handles correlated features
Small data	Often better	Needs more data

Evaluation: accuracy can mislead on imbalanced classes, so use precision $TP/(TP+FP)$, recall $TP/(TP+FN)$, and $F1 = 2PR/(P+R)$.

Module 4: Language models

The exam point: a language model predicts the next token. RNNs struggle with long-range context because of vanishing gradients, LSTMs fix this with gates and an additive cell state, and perplexity measures how well it all works.

The task. By the chain rule, $P(w_1, \dots, w_n) = \prod P(w_t | w_{<t})$. A language model estimates each factor.

N-gram models use the Markov assumption: condition only on the last $n-1$ words.

$$P(w_t | w_{<t}) \approx P(w_t | w_{t-n+1}, \dots, w_{t-1}) = \frac{\text{count}(w_{t-n+1}, \dots, w_t)}{\text{count}(w_{t-n+1}, \dots, w_{t-1})}$$

N-gram problems: sparsity (unseen n-grams get 0, so smoothing is needed), no context beyond n , and the table grows as $|V|^n$.

RNN language model. The hidden state carries the whole history: $h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$, and $P(w_{t+1}) = \text{softmax}(W_o h_t)$.

- **Teacher forcing:** during training, feed the true previous token rather than the model's own prediction.
- **Backprop through time (BPTT):** the gradient back to step $t-k$ multiplies k Jacobians, each involving W_h and $\tanh' \leq 1$. So it **vanishes** (or explodes) and early context is forgotten.
- Gradient clipping fixes *exploding* gradients only; it does nothing for vanishing ones.

LSTM. A cell state c_t plus three sigmoid gates:

$$f_t, i_t, o_t = \sigma(W[h_{t-1}, x_t] + b), \quad \tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \quad h_t = o_t \odot \tanh(c_t)$$

The additive update with $f \approx 1$ creates a gradient "highway", so information survives many steps. GRUs are a lighter variant with two gates.

Perplexity. The inverse probability of the test set, normalized per token. Lower is better.

$$\text{PPL} = \exp\left(-\frac{1}{N} \sum_t \log P(w_t | w_{<t})\right)$$

Intuition: a PPL of k means the model is as unsure as if it were choosing uniformly among k words. A uniform model over V gives $\text{PPL} = |V|$.

Decoding (generation)

Method	How	Trade-off
Greedy	Take the argmax at each step	Fast; repetitive; can miss better sequences
Beam search	Keep the top-k partial sequences	Better likelihood; still bland; good for MT
Sampling	Draw from softmax	Diverse; can go off the rails
Temperature T	$\text{softmax}(z/T)$	$T < 1$ sharper, $T > 1$ flatter
Top-k / nucleus (top-p)	Sample only from the top k tokens, or the smallest set with mass $\geq p$	Cuts off the unlikely tail

Seq2seq (encoder-decoder RNN): the encoder compresses the source into its final hidden state, and the decoder generates from it. That single vector is a bottleneck, which motivates attention in Module 6.

Module 5: Semantics

The exam point: meaning comes from context (the distributional hypothesis). You may need to compute TF-IDF by hand and contrast CBOW with Skip-Gram.

Distributional hypothesis: "You shall know a word by the company it keeps." Words in similar contexts get similar vectors.

TF-IDF. Weights a term up when it is frequent in *this* document and down when it is common across the corpus.

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times \log \frac{N}{\text{df}(t)}$$

- tf = count of t in d (sometimes log-scaled or length-normalized).
 - df = number of *documents* containing t . N = number of documents.
 - A term in every document gets $\log(N/N) = 0$ (e.g., "the").
 - Terms with the same df have the same IDF, whatever their counts inside a document.
- Show your work:** list which documents contain each term.

Cosine similarity compares document or word vectors regardless of length:

$$\cos(a, b) = \frac{a \cdot b}{\|a\| \|b\|}$$

Retrieval: represent the query and the documents as TF-IDF vectors, then rank by cosine.

Word2Vec learns dense vectors (e.g., 100–300 dimensions) from a sliding window. There are two matrices: center embeddings v and context embeddings u .

	CBOW	Skip-Gram
Input	Context words (averaged)	Center word
Predicts	Center word	Each context word
Loss	$-\log P(\text{center} \mid \text{context})$	$-\sum \log P(\text{context}_j \mid \text{center})$
Strengths	Faster; smooths over the window	Better on rare words and small data

- $P(o|c) = \text{softmax over } V \text{ of } u_o \cdot v_c$. That sum over V is expensive.
- **Negative sampling:** turn it into binary classification. Real (center, context) pair $\rightarrow 1$; k random "noise" words $\rightarrow 0$. Loss = $-\log \sigma(u_o \cdot v_c) - \sum \log \sigma(-u_k \cdot v_c)$.

Embedding properties: similar words cluster together, and analogies work by vector arithmetic (king – man + woman $\approx$ queen).

Limits:

- **One vector per word type**, so the senses of "bank" get blended. Contextual embeddings (Module 6) fix this.
- Embeddings absorb biases from their training text.

	Sparse (BoW / TF-IDF)	Dense (Word2Vec)
Dimension	$ V $ (10k–100k+)	~100–300
Synonyms	Unrelated dimensions	Nearby vectors
Built by	Counting	Learning a prediction task

Module 6: Modern neural architecture

The exam point: attention lets every token look directly at every other token. Positional embeddings restore word order. The decoder uses masked self-attention plus cross-attention. RLHF trains a reward model rather than learning from raw human labels.

Attention. Introduced to fix the seq2seq bottleneck: the decoder takes a weighted mix of *all* encoder states instead of only the last one.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^{\text{top}}}{\sqrt{d_k}}\right)V$$

- Q = what I'm looking for, K = what each token offers for matching, V = what each token contributes.
- Divide by $\sqrt{d_k}$ so large dot products don't saturate the softmax.
- **Multi-head:** several attention heads in parallel, each with its own projections. Their outputs are concatenated, so each head can learn a different relationship (syntax, coreference, and so on).

Transformer block: multi-head self-attention, then add & layer-norm, then position-wise feed-forward, then add & layer-norm. Residual connections keep gradients flowing.

There's no recurrence, so all positions are processed in parallel and any token is one step from any other. That is the long-range win over RNNs. Cost: attention is $O(n^2)$ in sequence length.

Positional embeddings. Self-attention is permutation-equivariant: shuffle the input and the outputs shuffle the same way. Without position information, the model sees a bag of words. The fix is to add a position vector to each token embedding, either sinusoidal (fixed) or learned.

Three kinds of attention in the original encoder-decoder

Sub-layer	Q from	K, V from	Mask
Encoder self-attention	Encoder	Encoder	None (bidirectional)
Decoder masked self-attention	Decoder	Decoder (earlier positions)	Causal: no peeking at future tokens
Decoder cross-attention	Decoder	Encoder output	None over the source

Model families

- **Encoder-only (BERT):** bidirectional, trained with masked-LM. Good for classification and tagging; contextual embeddings fix the "one vector per word" problem.

- **Decoder-only (GPT):** causal language model that generates text. Today's chat LLMs are this type.
- **Encoder-decoder (T5, original Transformer):** translation and summarization.

From pretrained LM to assistant

1. **Pretraining:** next-token prediction on a huge corpus.
2. **Supervised fine-tuning / instruction tuning:** train on (prompt, good response) pairs.
3. **RLHF:**
 - a. Humans rank or compare model outputs.
 - b. Train a **reward model** (a classifier/regressor) on those preferences.
 - c. Optimize the LM against the reward with RL (PPO), plus a KL penalty to stay close to the original model and avoid reward hacking.

Why a reward model instead of raw feedback: human labels are scarce and costly, and preferences are noisy. People compare more reliably than they score. A trained reward model generalizes and can score unlimited samples during RL.

Other terms: in-context / few-shot learning (examples in the prompt, no weight updates); fine-tuning vs. prompting; sub-word tokenization (BPE).

Practice exam: worked answers

These answers rely on standard NLP material. Q2 depends most on how the lectures phrase things, so check it against your notes.

Q1. Why is language hard? (select all)

A, D, E, G, J. Polysemy and domain meaning (A), creative composition like portmanteaus (D), metaphor (E), syntactic ambiguity (G), and constantly new words (J).

Traps: B, C, F, H and I are absolute claims that are false. Computers handle sentences, learn new words, and model grammar statistically.

Q2. Correct application of Generalized Bayes

Most likely 2: determining how much event A affects event B (or vice versa). Bayes' rule flips a conditional. You estimate $P(\text{cause} \mid \text{effects})$ from $P(\text{effects} \mid \text{cause})$, which is easier to measure.

Why not 4: converting between conditionals and joints is just the definition $P(A,B) = P(A|B)P(B)$. Why not 3: mutually exclusive events need no Bayes. Confidence: medium. Confirm with your Module 2 notes.

Q3. Assumption for Generalized Bayes

1: observable effects are independent, more precisely *conditionally* independent given the cause. That is what lets the likelihood factor into a product:

$$P(A \mid B_1, \dots, B_n) \propto P(A) \prod_{i=1}^n P(B_i \mid A)$$

Not "mutually exclusive" (independent effects can co-occur), and no normality assumption.

Q4. Why take the log in Bayesian classification?

- Underflow.** The score multiplies many probabilities below 1, so it quickly rounds to 0 in floating point.
- Useful properties.** The log is strictly monotonic, so the argmax class doesn't change. It turns products into sums, which are stable and cheap. It also makes the classifier linear in the features: $\log P(y) + \sum \text{count}(w) \cdot \log P(w|y)$.

Q5. Softmax normalization

A: Not if you are sampling. Naive Bayes can drop $P(x)$ because it only takes an argmax, and a shared constant doesn't change which class wins. Sampling needs real probabilities, so you must divide by $Z = \sum \exp(z_k)$. That said, the class with the larger score is still the more likely one; normalization rescales scores but doesn't reorder them.

$$P(y=k \mid x) = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

B: Yes. Softmax is monotonic, so argmax over the logits z equals argmax over the probabilities. Skip the exponentials and the sum entirely and pick the largest logit.

Q6. Bag-of-words (true statements)

1 and 4. The vectors are vocabulary-sized and sparse (1), and word order is lost (4).

False: BoW has no grammar or roles (2). Unseen words get no feature, so they are not "easily handled" (3). Synonyms land in separate dimensions (5); grouping them is what embeddings fix.

Q7. How LSTMs capture long-range dependencies

A basic RNN pushes its hidden state through the same weight matrix and squashing function at every step. Backpropagated gradients get multiplied by that matrix again and again, so they vanish or explode.

An LSTM adds a **cell state** that is updated *additively*, with three **gates** (sigmoid values in $[0,1]$):

- **Forget gate:** how much of the old cell state to keep.
- **Input gate:** how much new candidate information to write.
- **Output gate:** how much of the cell to expose as the hidden state.

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

When f is close to 1, information and gradients flow across many steps almost unchanged. The network *learns* what to remember and what to forget.

Q8. Perplexity

2. Perplexity is the model's average "surprise": the inverse probability of the test set, normalized per token. Lower is better. Better long-range modeling assigns more probability to the true next tokens, so perplexity goes down.

$$\text{PPL} = P(w_1, \dots, w_N)^{-1/N} = \exp\left(-\frac{1}{N} \sum_{t=1}^N \log P(w_t \mid w_{<t})\right)$$

Traps: option 1 says long-range modeling *raises* perplexity. Option 3 is the geometric mean without the inverse; perplexity is its reciprocal. Option 4 describes accuracy.

Q9. Word2Vec: CBOW vs. Skip-Gram

A. Learning problem

- **CBOW:** input is the context words in a window, averaged or summed into one vector. Output is the center word.
- **Skip-Gram:** input is the center word. Output is each surrounding context word. It does better on rare words but is slower to train.

B. Loss. Both use cross-entropy (negative log-likelihood) over a softmax across the vocabulary.

$$L_{\text{CBOW}} = -\log P(w_t \mid w_{t-c}, \dots, w_{t+c})$$

$$L_{\text{SG}} = -\sum_{-c \leq j \leq c, j \neq 0} \log P(w_{t+j} \mid w_t), \\ \text{quad } P(o \mid c) = \frac{\exp(u_o^{\text{top}} v_c)}{\sum_{w \in V} \exp(u_w^{\text{top}} v_c)}$$

The full softmax over V is expensive, so in practice **negative sampling** replaces it. Push the true pair's score up and the scores of k random words down:

$$L = -\log \sigma(u_o^{\text{top}} v_c) - \sum_{k=1}^K \log \sigma(-u_k^{\text{top}} v_c)$$

Q10. Same IDF as "bank"

IDF depends only on **document frequency** (how many documents contain the term), not how often it appears within them. $N = 3$.

Term	Docs containing it	df	IDF = $\log(N/\text{df})$
bank	d3	1	$\log 3 \approx 1.099$ (ln)
barked	d2 (twice, still one doc)	1	$\log 3 \approx 1.099$
fox	d1	1	$\log 3 \approx 1.099$
and	d2, d3	2	$\log 1.5 \approx 0.405$
dog	d1, d2	2	$\log 1.5 \approx 0.405$

Answer: a (barked) and c (fox). The trap is "barked": it appears twice in d2, but that raises its *TF*, not its df.

Q11. Positional embeddings

A. Without them, a query's score against each key, $q \cdot k / \sqrt{d}$, depends only on *which* words are present, not *where* they are. Shuffling the sequence just permutes the scores. The same word at two positions gets identical scores, and each output is a function of the multiset of words, like a bag of words.

B. Use two sentences with the same words in a different order:

- "Alice gave Bob a book. The one who received it was ..." $\rightarrow$ *Bob*
- "Bob gave Alice a book. The one who received it was ..." $\rightarrow$ *Alice*

With positions, the model knows who gave and who received, so the predictions differ. Without them, both inputs are the same bag of tokens, so the prediction is the same.

Nuance if pressed: a causal mask in a decoder-only model leaks some position information, but the expected answer ignores this.

Q12. Cross-attention in the decoder

1. **Role:** it connects the decoder to the input. Queries come from the decoder's current states, while **keys and values come from the encoder's output**. Each target position can then pull in the relevant source tokens, much like alignment in translation.
2. **Difference from masked self-attention:** masked self-attention takes Q, K and V all from the decoder's *own* previous tokens, with a causal mask so a position can't see the future. Cross-attention draws K and V from a different sequence and needs no causal mask, since the full source is known.

Q13. Why RLHF trains a reward model (classifier)

- **Scale and cost:** human labels are slow and expensive. RL needs a reward for many thousands of sampled outputs, and a learned reward model scores any output instantly.
- **Generalization:** a few comparisons train a model that scores outputs no human ever saw.
- **Comparisons beat absolute scores:** people agree more on "A is better than B" than on a 1-10 rating. The reward model turns pairwise preferences into a scalar reward for the RL step (e.g., PPO).
- **Noise smoothing:** individual labelers are inconsistent, and a trained model averages over them.
- **Gradient signal:** raw feedback is a one-off label, not a function you can query during training. The reward model is always available inside the training loop.

Extra practice (computation drills)

The practice exam has only one calculation, but the real exam may have more. These five cover the computable topics. Try each one before reading the answer.

P1. Naive Bayes with add-1 smoothing. Training data: Pos = "good good great", Neg = "bad good". Equal priors. Classify "good bad".

$V = \{\text{good, great, bad}\}$, so $|V| = 3$. Pos has 3 tokens, Neg has 2. Pos: $P(\text{good}) = (2+1)/(3+3) = 0.5$, $P(\text{bad}) = (0+1)/6 \approx 0.167$. Score = $0.5 \times 0.5 \times 0.167 \approx \mathbf{0.042}$. Neg: $P(\text{good}) = (1+1)/(2+3) = 0.4$, $P(\text{bad}) = (1+1)/5 = 0.4$. Score = $0.5 \times 0.4 \times 0.4 = \mathbf{0.080} \rightarrow$ **Neg**. Without smoothing, $P(\text{bad} | \text{Pos}) = 0$, so Pos is impossible. That's why smoothing exists.

P2. Softmax. Logits $z = [2, 1, 0]$. Give the probabilities and the argmax.

$e^z = [7.389, 2.718, 1.000]$, sum = 11.107. $P = [\mathbf{0.665}, \mathbf{0.245}, \mathbf{0.090}]$. Argmax = class 0, the same as the largest logit, so the softmax computation wasn't needed to find it.

P3. Perplexity. A model assigns 0.5, 0.25, 0.125 and 0.5 to the four true tokens of a test sentence.

Product = $2^{-1} \times 2^{-2} \times 2^{-3} \times 2^{-1} = 2^{-7}$. PPL = $(2^{-7})^{(-1/4)} = 2^{1.75} \approx \mathbf{3.36}$. The model is as uncertain as choosing uniformly among about 3.4 words.

P4. Scaled dot-product attention. $q = [1, 0]$; $k_1 = [1, 0]$, $k_2 = [0, 1]$; $v_1 = [10, 0]$, $v_2 = [0, 10]$; $d_k = 2$.

Scores: $q \cdot k_1 / \sqrt{2} = 0.707$, $q \cdot k_2 / \sqrt{2} = 0$. Softmax gives $[\mathbf{0.67}, \mathbf{0.33}]$. Output = $0.67 \times v_1 + 0.33 \times v_2 = [\mathbf{6.7}, \mathbf{3.3}]$. The output leans toward the value of the best-matching key. Follow-up: swap the order of the two keys (and their values) and the output doesn't change. That is the missing-position problem from Q11.

P5. TF-IDF, both parts. Using the practice exam's three documents, what is the tf-idf of "barked" in d2? Use natural log and raw counts.

tf = 2 (it appears twice in d2). idf = $\ln(3/1) = 1.099$. tf-idf = $\mathbf{2.197}$. For comparison, "dog" in d2 is $1 \times \ln(3/2) = 0.405$. If your class uses log10, $IDF(df = 1) = 0.477$; check which base the lectures use.

Quick-fire concept checks

- Why does gradient clipping help RNNs with exploding gradients but not vanishing ones? (It caps large gradients; it can't enlarge tiny ones.)
- Why can't Word2Vec tell "river bank" from "bank loan"? (It learns one static vector per word type.)

- Why does a GPT-style model need a causal mask during training? (Otherwise each position could see the token it is meant to predict.)
- Why does RLHF add a KL penalty? (To keep the policy near the SFT model and stop it gaming the reward model.)