# CS 376 Study Guide

When two options are given, e.g., \_\_\_a / b\_\_\_\_, circle the appropriate one.

## Language Modeling

The first task of language modeling is to represent a document as a sequence of numbers. We do this by splitting the document into units called *tokens*, then assigning each token a number based on looking it up in the *vocabulary*. We have a few options of how to split documents into tokens. We could use one token for each *character* (or, e.g., each byte of the document encoded in UTF-8). We could use one token for each *word*. Or we could split the difference and use subword tokens. List a pro and a con for each approach:

Character-level

Word-level

Subword-level

For a running example, let’s suppose that we’re asking GPT-2 to predict the next token after 5-token sequence. We’ll use a vocabulary size of 50k BPE (subword) tokens. So our input tensor has a single axis; input.shape = \_\_\_\_\_\_\_.

Inside of this model, language is not represented symbolically. Instead, each token is represented by a(n) \_\_\_\_\_\_\_\_\_\_\_\_, which is a vector than has several hundred dimensions (768 in GPT-2). The first thing the model does to the input is to look up the \_\_\_\_\_\_ of each token. This gives us a tensor of shape \_\_\_\_\_\_\_\_\_\_\_ that we’ll use in the actual neural net.

The model does lots of computation, which we’ll describe in a moment, but when it’s done it gives us a tensor of next-word scores, more formally called \_\_\_\_\_\_\_\_, with shape \_\_\_\_\_\_\_\_\_. Even though we only needed the predictions for what token follows the *last* token, we got other predictions basically for free along the way; each one of those represents \_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. We change those scores into probabilities for each token by applying the \_\_\_\_\_\_\_\_\_\_\_ operation, which we used before for image classification; language modeling is just a bunch of classification tasks. These are the probabilities that we see when we click on a single token in the OpenAI Playground.

We can compute the probability that the model would assign to an entire sequence of tokens by \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. We usually do this computation in log space because \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. When we click a single token in the OpenAI Playground, we see a log-prob that we could compute by taking the log of \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. When we select several tokens (or a whole document), the Playground shows us the logprob of that whole sequence (conditioned on all of the tokens that came \_\_\_before / after\_\_ it). To compute that logprob number, the playground aggregated all of the individual logprobs by taking their \_\_\_\_\_\_\_\_\_\_\_\_.

To evaluate how well a language model fits data, we use the same loss function that we used for image classification, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, which for a categorical distribution like this is computed by taking the negative of the \_\_\_\_\_\_\_ of the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. So we can train a model maximize the probability that the model assigns to each document in the training set, which is equivalent to minimizing the average of the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ loss for each token. We can compare multiple models by evaluating what logprobs they give to the same example.

Recall that the *input* to the main part of the model is an embedding for each token, looked up in a table. This embedding represents the token’s meaning \_\_\_in isolation / in its surrounding context\_\_\_. The model’s output is an embedding of \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

## The Language *Model[[1]](#footnote-2)*

The architecture of the model that underpins ChatGPT, Gemini, etc. is called a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_. It processes input tokens by alternating between two main types of layers: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_\_\_\_\_.

The input and output shapes of each layer are conventionally the same because each layer computes what to *add* to the embedding vector, i.e., output = input + layer(input). This arrangement is called a *residual network*; it helps stabilize training of very deep networks. This means that the input and output to each layer in our running example would have shape \_\_\_\_\_\_\_\_\_\_\_\_.

### Self-Attention Layer

In terms of information flow, the purpose of a self-attention layer is to:

To do this, each “attention head” computes three vectors at each position (i.e., for each token), called the \_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_, and \_\_\_\_\_\_\_\_\_\_. Of these, the shape of the \_\_\_\_\_\_\_\_\_ and the \_\_\_\_\_\_\_\_\_ must match each other; the \_\_\_\_\_\_\_\_\_ doesn’t necessarily need to match the others, but often does; a typical shape is on the order of 128 numbers per token. Typically the mathematical operation used to compute each of these vectors is a \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, meaning that we can implement each one in PyTorch using nn.Linear.

At each position, that position’s \_\_\_\_\_\_\_\_\_ vector is compared with the key vectors for each other position (including itself, actually) by computing the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ between the two vectors. The results are collected in a vector, which (after being multiplied by a constant) is passed through a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_ operation to convert them to attention weights.

But in a next-word prediction model, tokens should only be able to attend to tokens that come \_\_\_\_\_\_\_\_\_\_\_\_\_\_ them, so before converting to probabilities, we might need to apply a \_\_\_\_\_\_\_\_\_\_ to make sure that some attention weights come out to be zero.

We then compute the *output* of the attention head at some position by computing, for each other position, the sum of that position’s \_\_\_\_\_\_\_\_\_\_\_, weighted by the attention weight between the two tokens.

A single self-attention layer may have many attention heads; their results are concatenated together and passed through a final linear layer to get back to the model’s embedding dimension.[[2]](#footnote-3)

In practice, it’s much more efficient to do all these computations for all tokens in the sequence at a time, so we compute arrays with a row for each token, each row containing the key/query/value for that token. The self-attention weights can then be organized into a square matrix. For example, in our 5-token example, the self-attention layer got an input of 784 numbers for each of the 5 tokens (so input shape is 5 x 784); each of the heads computes a 128-dimensional vector for each token, and the self-attention weights matrix is 5x5 (with half of it masked out).

In summary, for a self-attention layer in a Transformer with embedding dimension of 768 and head dimension of 128 applied to a 7-token sequence, the shapes of the activations are:

|  |  |  |
| --- | --- | --- |
| Array | Rows | Columns |
| Input embeddings | 7 (one per \_\_\_\_\_\_\_\_) | 768 (since it’s the \_\_\_\_\_\_\_\_) |
| Keys | Same | 128 (since it’s the \_\_\_\_\_\_\_) |
| Queries | Same | Same as keys |
| Values | Same | Same as keys |
| Self-Attention Weights | \_\_\_ (one per \_\_\_\_\_\_) | \_\_\_\_\_ (one per \_\_\_\_\_\_) |

These are computed by linear layers with the following input and output dimensionalities:

|  |  |  |
| --- | --- | --- |
| Linear layer | Input Dim | Output Dim |
| Input-to-keys | \_\_\_\_\_\_\_ | \_\_\_\_\_\_\_ |
| Input-to-queries | \_\_\_\_\_\_\_ | \_\_\_\_\_\_\_ |
| Input-to-values | \_\_\_\_\_\_\_ | \_\_\_\_\_\_\_ |
| Values to layer output (assuming one head) | \_\_\_\_\_\_\_ | \_\_\_\_\_\_\_ |

*Bonus:* There is typically also a *layer normalization* operation either before or after the main part of the layer. This operation first “standardizes” the input and output embeddings by subtracting the \_\_\_\_\_\_\_\_ and then dividing by the \_\_\_\_\_\_\_\_\_\_\_\_\_\_. Then it multiplies by a *learnable* scale factor and adds a *learnable* offset.

### Feed-Forward Layer (FFN / MLP)

The self-attention layer effectively exchanges information between positions but does not allow for much nuance in the computation at each position since its output can only be composed of sums of linear transformations of other tokens’ embeddings. To allow more complex logic, we intersperse self-attention layers with *feed-forward layers*, which are just are old friends the Multi-Layer Perceptron (MLP). (You may also see this implemented using a *Convolution* operator, but that’s purely an implementation detail; don’t get confused.)

Recall that the shape of the input to each layer of the model (including the MLP) in our running example is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. And the shape of the output is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

The MLP is applied on \_\_\_the concatenation of all input positions / each input position independently\_\_\_. This means that, in terms of the information flow between different tokens, an MLP \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

The Transformer MLP typically first computes a linear projection of its input to a hidden space of 4 times its embedding dimension (768\*4=3072 for GPT-2), applies a nonlinearity like ReLU (or fancier variants like GeLU or Swish), then projects back into the embedding space. A PyTorch implementation might look like:

nn.Sequential(  
 nn.Linear(in\_features=\_\_\_\_\_\_\_\_\_, out\_features=\_\_\_\_\_\_\_\_\_),  
 nn.ReLU(),  
 nn.Linear(in\_features=\_\_\_\_\_\_\_\_\_\_, out\_features=\_\_\_\_\_\_\_\_\_)

)

## Training

We can train a Transformer model by stochastic gradient descent, like other ML models. Gradients flow particularly well in Transformers because of the *residual* connections that bypass each layer. If, for each layer, output = input + layer(input), then during backpropagation (ignoring the layer norm) we can compute input\_grad = input\_grad\_from\_layer + \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (in words; don’t try to derive the exact formula).

## Dealing with Position

Recurrent neural networks (RNNs) process a sequence one token at a time. The processing stays the same for all steps, but the output of one step is connected to the input for the next step. That input is a “hidden state” vector that is analogous to the *accumulator variable* from CS108.

To get a recurrent neural net to always output what the *previous* token was, we could set the hidden state vector to \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. So the RNN can trivially keep track of position.

Implementing the same operation (echo previous token) in Transformers is a bit more tricky. We would need to get each token to *attend to* the prior token. So we’d need the self-attention matrix to look like (*draw it*):  
  
  
  
  
  
To get that matrix, we could set the key to the \_\_\_\_\_\_\_\_\_\_\_\_\_\_ and the query to \_\_\_\_\_\_\_\_\_\_. Then, to get the output to be the previous token’s embedding, we would set the value to \_\_\_\_\_\_\_\_\_\_\_\_. To actually implement this using the Transformer architecture described above, we need to be able to access and compare the *indices* of each position. So we *add* to each token embedding an embedding corresponding to the *position* of that token. In the original Transformer paper (Attention is All You Need), position embeddings were hard-coded; in the GPT model these embeddings are learned just like token embeddings; more recent models generally tweak the hard-coding process.

## Generating from a Language Model

Describe how a language model can be used to complete a sentence.

What is temperature in language modeling? What is the effect of setting temperature to be nearly 0? What would happen if we set temperature to a very large number?

# Prompt Engineering and Instruction Tuning

A language model is generally trained to mimic text in its training set. When trained on the Internet, learning to mimic is very useful because it has to learn skills and concepts like (list 3):

1. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
2. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
3. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

However, the text generated by a model trained this way may be not useful because (describe a few problems).

To make a causal language model into a dialogue agent, we first fine-tune it on high-quality dialogues, often curated by human labelers. A dialogue can be viewed as a back-and-forth between a “user” and an “assistant”. To turn a dialogue into a “document” that a language model can train on, we just insert special tokens for whose turn it is in the dialogue. So a document might look like (sketch an example):

Then when a user asks a question like “write a Python program to check if a number is prime”, the code would call model.generate() on a sequence that looks like:

### Prompt Engineering

Tweaking the prompts (prefixes) that we give the model to complete can help a model give more useful generations. One approach is few-shot learning (in-context learning). For example, we might give a prompt like:

We can also help a model to reason by using chain-of-thought prompting. For example:

Finally, we might use retrieval-augmented generation, where we search for some reference documents that might be relevant (e.g., encyclopedia pages, software manual pages, an organization’s knowledge base, etc.) and then include them in the context. For example:

Despite all of these attempts, language models still can generate incorrect results, which is called \_\_\_\_\_\_\_\_\_\_\_\_\_\_.

## Impacts of Language Modeling Systems

Describe how language models can *hallucinate* content that sounds plausible but is in fact untrue, even if its training data consists of only true statements. (Consider, for a toy example, a model that was trained only on many-digit addition problems, like 548106+1058234=1606340.)

## Image Generation

Various approaches have been tried to generate images. Three examples are autoregressive, GAN, and Diffusion.

### Autoregressive Image Generation

An autoregressive image generator works like a language model: it treats each image as an ordered sequence, trains a model to output the conditional probability of the next item in the sequence, then generates a new image start-to-end by sampling from the conditional distribution as if it were the next word. This model is attractive in its simplicity. But it is not popular in practice because \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

### GAN

GAN stands for \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Draw the relationship between: the random vector z, the Generator network, the Discriminator network, and true images.

The Discriminator is trained to minimize the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ loss between its predicted labels and \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

To train the Generator network, we take gradient steps that \_\_\_\_\_\_\_\_\_\_\_\_ the loss of the discriminator network.

Describe the shape of the data that back-propagates from the Discriminator network to the Generator network. How would you interpret it?

Once a GAN is trained, we generate an image by \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

### Diffusion

The basic task of a Diffusion model is to transform a noisy image into a \_\_\_\_\_\_more / less\_\_\_\_\_ noisy image.

### Social Impacts of Image/Video/Audio Generation

Pick one or more of the following impacts and analyze it through the lens of some Reformed Christian conceptual framework.

* Generating images of newsworthy events that did not occur
* Generating pornographic images of people
* Synthesizing a recording of someone saying something they didn’t say
* Generating artwork in the style of an artist who did not license their work that way
* List a few more of your own.

1. Note: This section describes the common decoder-only Transformer, often called GPT. Encoder-only models like BERT just omit the masking; encoder-decoder models like T5 add a “cross-attention” layer between the encoder and decoder. For *many* other variants, see https://github.com/lucidrains/x-transformers/ [↑](#footnote-ref-2)
2. Strictly speaking this last linear layer is redundant; we could subsume it into the computation of the value instead. [↑](#footnote-ref-3)