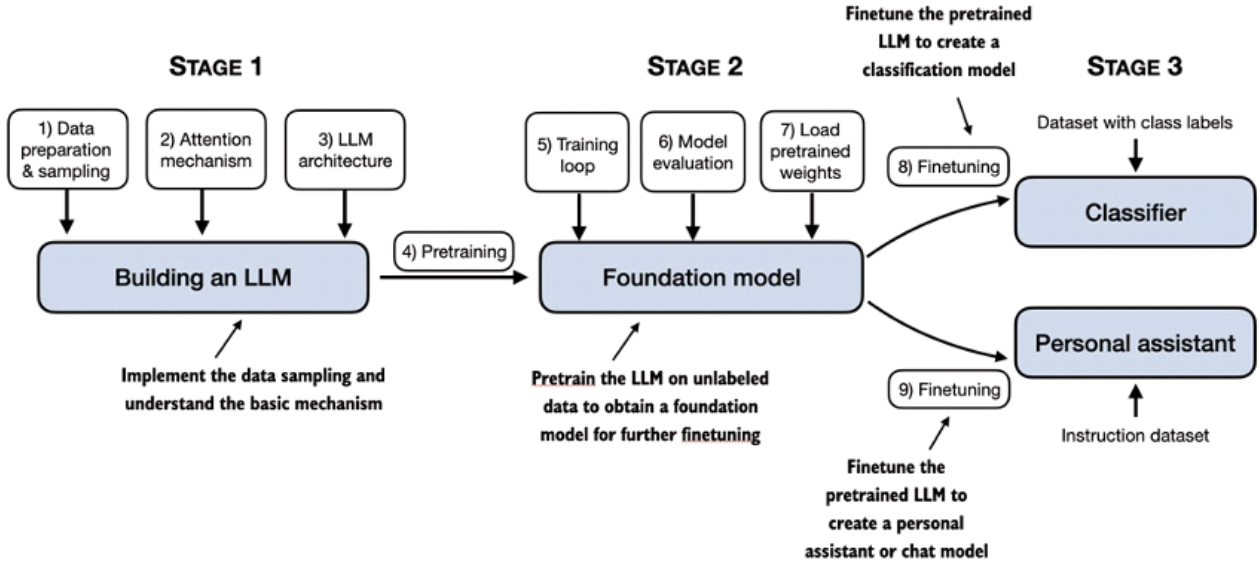
1. Intro

2025年5月8日 17:34

Pretraining: trained on raw, unlabeled data.
fine-tuning: trained on a smaller set, classified, labeled data. Two types are instructional and classification

Decoder/Encoder, each process the input/output. Transformer is decoder only model since it only need to generate not translate.
Next word prediction task is quite neat. It can classify/label the data by looking at the next word in actual text, compared with the generated text. It allows the usage of unlabeled, raw data as classified data.



2. Embedding & Tokenization

2025年5月9日 8:38

This chapter we focus on preparing data for LLM like parsing text into tokens.

PyTorch is a python library for deep learning. It has three components:
1: Tenser library: tenser is n-dimensional array-like structure.
2: Automatic differentiation(autograd): Allow us to efficiently train NN with backprop.
3: Batch: A list(or tenser) of tokenized, encoded tokens sequences(sentences).

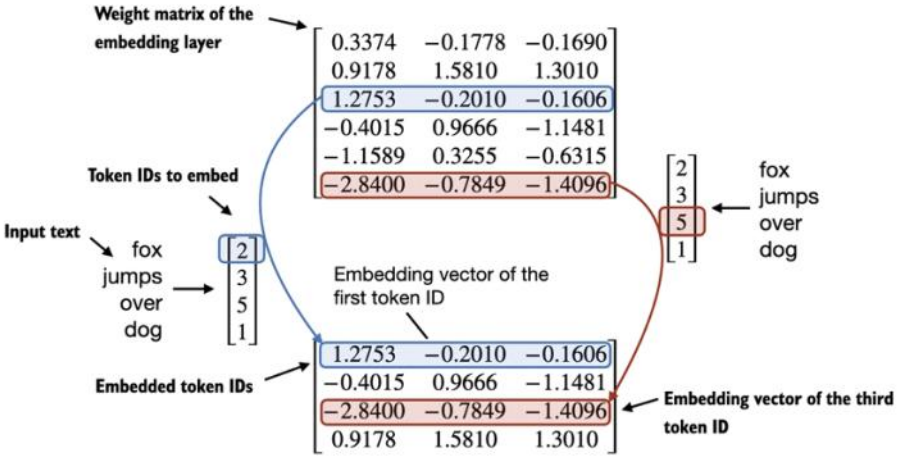
Embedding: converting each token into a vector. This can be done on different types of info, using different models. Word2vec is a popular embedding model though most train their own embedding model for specific tasks.
Text -> List of tokens(words) -> List of unique ids -> List of vectors

Basically a tokenizer parse the text and punctuations in to a list. Ignore space(or not). For each element in the list give i t a token ID and we have out vocabulary. With this vocabulary we build a tokenizer that turn any text into list of token ID's and vise versa. Note we often add extra vocabularies for special meanings:

- `[BOS]` (beginning of sequence) marks the beginning of text
- `[EOS]` (end of sequence) marks where the text ends (this is usually used to concatenate multiple unrelated texts, e.g., two different Wikipedia articles or two different books, and so on)
- `[PAD]` (padding) if we train LLMs with a batch size greater than 1 (we may include multiple texts with different lengths; with the padding token we pad the shorter texts to the longest length so that all texts have an equal length)
- `[UNK]` to represent words that are not included in the vocabulary
- Note that GPT-2 does not need any of these tokens mentioned above but only uses an `<|endoftext|>` token to reduce complexity

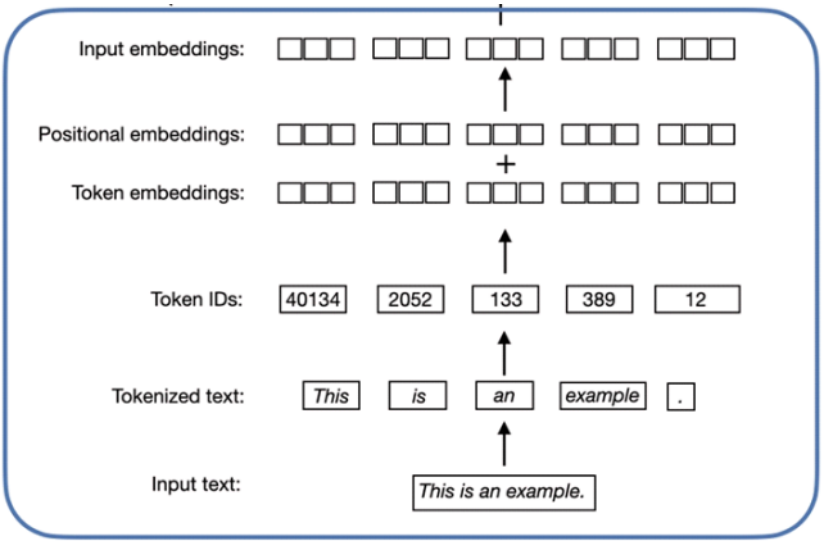
GPT-2 uses bytepair encoding(BPE) where it breaks words into smaller, known, or even single letters to deal with unfamiliar words.
We use a sliding window approach on tokenized data to generate input target pairs for LLM training

Embedded matrix are lookup tables for each token ID.



Embedding layer are initialized randomly to begin with.
This is not enough. To implement the attention mechanism, we need to also store the position information somewhere.
This can be done with relative position or absolute positional embeddings. GPT use absolute embeddings.

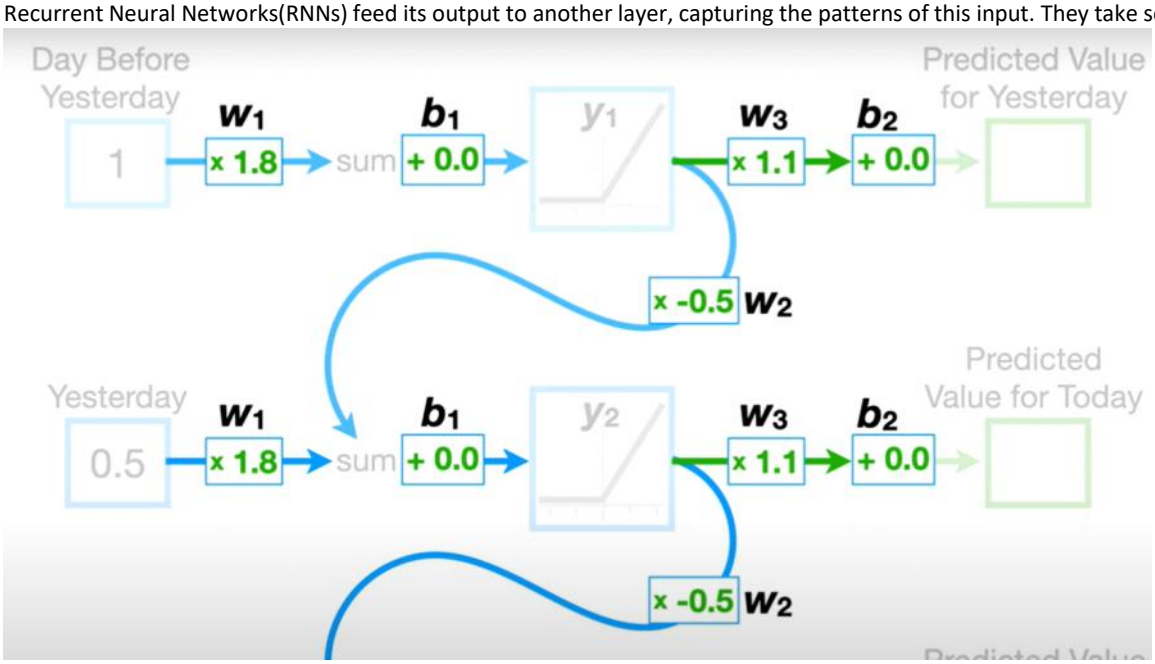
The absolute positional embeddings have the same dimension as the token embeddings.
We do this so that word in different positions can obtain different meanings by modifying it with positional embeddings.
Note that positional embeddings takes the position number not the token id, returns a vector, add it to the token embedding, and return the final input embeddings.


Input embeddings:
Positional embeddings:
Token embeddings:
Token IDs:
40134
2052
133
389
12
Tokenized text:
This
is
an
example
.
Input text:
This is an example.

3. Attention

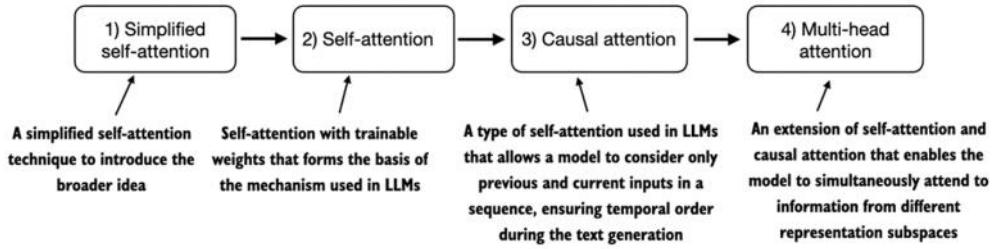
2025年5月13日 10:11

Motivation:
Recurrent Neural Networks(RNNs) feed its output to another layer, capturing the patterns of this input. They take sequential input.

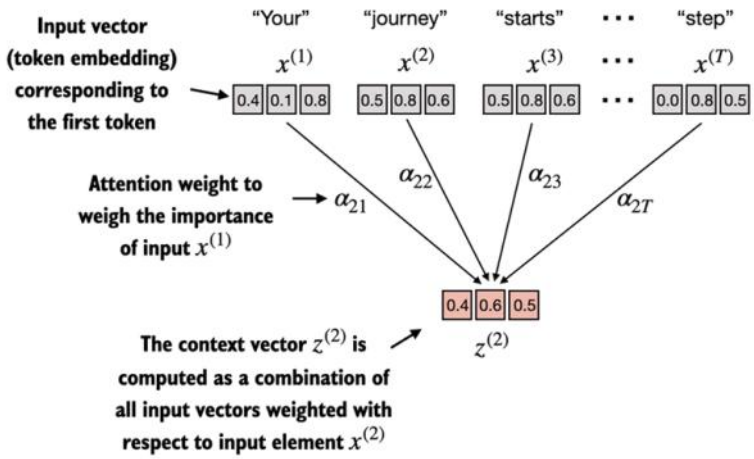


RNN's have defects. They don't capture the information earlier well, that is, they "forget" earlier information. Also, the W terms can lead to gradient explode & gradient vanishing. This issue is addressed by Long short term mem model. Attention & transformer is another solution.

We implement the following one by one

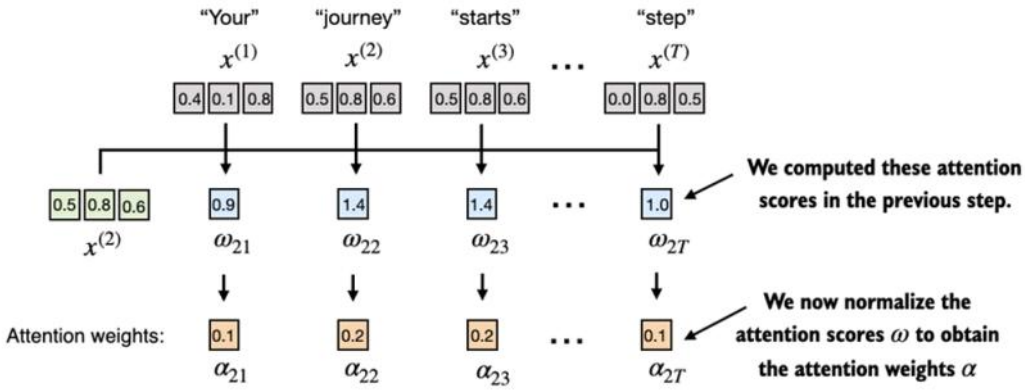


Simplified self-attention: The goal is to compute a context vector, for each input element, that combines information from all other input elements. In the example depicted in this figure, we compute the context vector z (2) . The importance or contribution of each input element for computing z (2) is determined by the attention weights α21 to α2T. When computing z (2) , the attention weights are calculated with respect to input element x(2) and all other inputs.

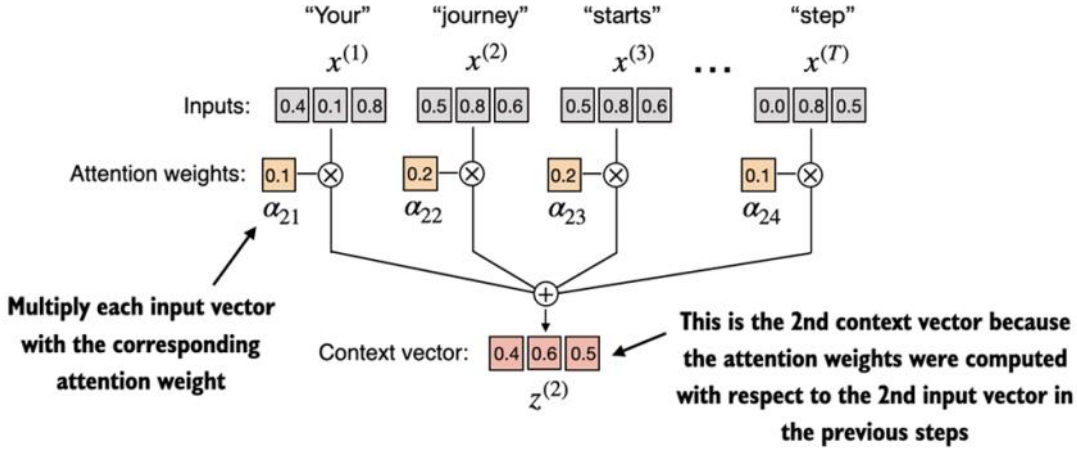


Context vector's purpose is to create enriched representations of each element in an input sequence (like a sentence) by incorporating information from all other elements in the sequence.

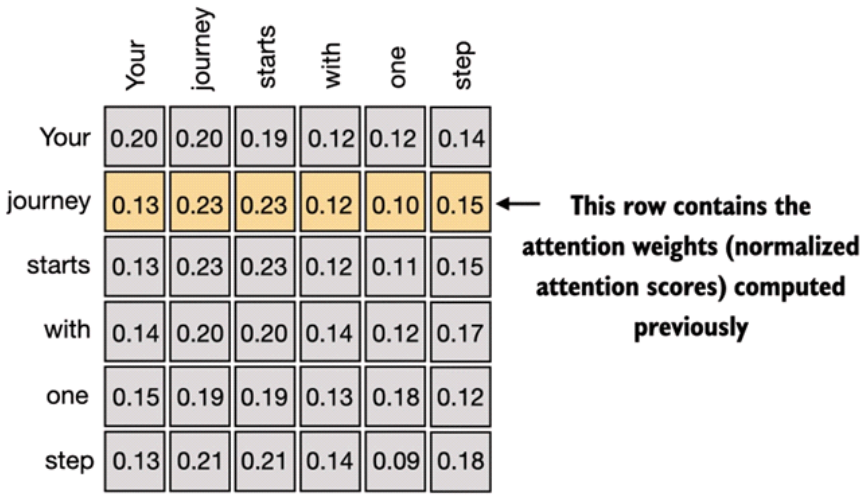
Attention score(ω): \omega_ij measures the strength of relevance between two tokens, for example, man and boy will have a strong w_ij. We say x(i) the query, the token currently focusing on , x(j) is a key. The \omega is simply calculated as the dot product between the query and a key.
Attention weights(α): For each attention value, we use the softmax function to normalize it so that their sum adds up to 1. Also the weights are always position, so they can be interpreted as probabilities.



Context Vetor(z): Finally, we multiply each token vector with its attention weights, sum it together to obtain the context vector for this query("journey"). Intuitively, z captures the information in the sentence relative to the query. For example, when looking at "bank" and there's a lot of financial tokens, the context vector would represent something like "finance" which allows the model to use that information to discriminate the meanings of "bank". More importantly, z is the final vector we pass into the next step, so z can be understood as a reinterpretation of the original vector with respect to the context.



Finally we can compute z for all tokens, note this is just a very simplified version of attention.

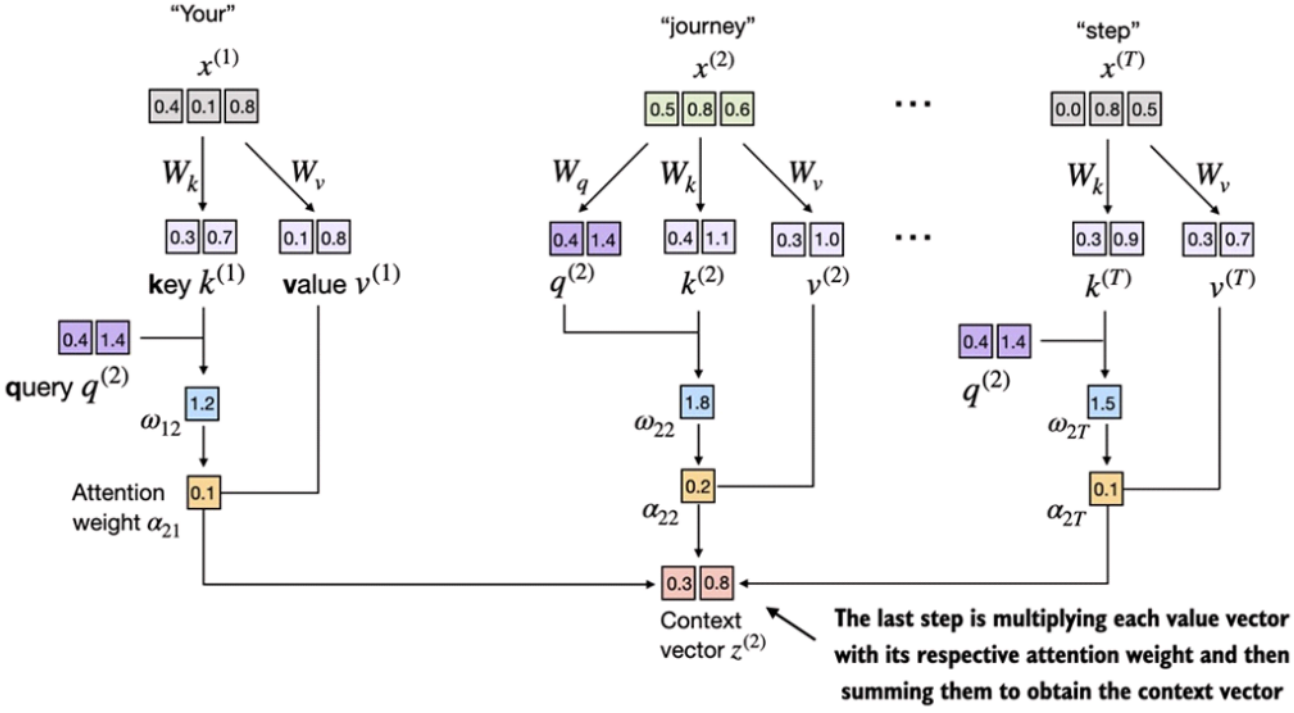


2. Self-Attention

Now we're interested in implementing part 2, the self-attention which can be trained to generate better context vectors. This is done using three trainable matrices, namely: Queue, key and value.
For each token, we can compute its query(q), key(k), and value(v) vectors by multiplying its vector with the corresponding Q, K,V matrices. Then when computing attention score ω, instead of calculating the dot product, we take the query vector of the query, and the key vector of the key, then takes its dot product. Note that this process of multiplying with Q,K,V is called a linear projection.

Next we calculate the attention weights using softmax. The difference to earlier is that we now scale the attention scores by dividing them by the square root of the embedding dimension of the keys. This is done to avoid small gradients.

The last step is to create context vectors. Instead of using the embedding vector, we use the value vector and multiply with its attention weights.



It's important to know that dimension of output depends on QKV dimensions, not embedding dimensions. We in fact have to pass the context vector into another Output matrix to turn it back into embedding dimension.
Intuitively, Q,K,V can be understood as:

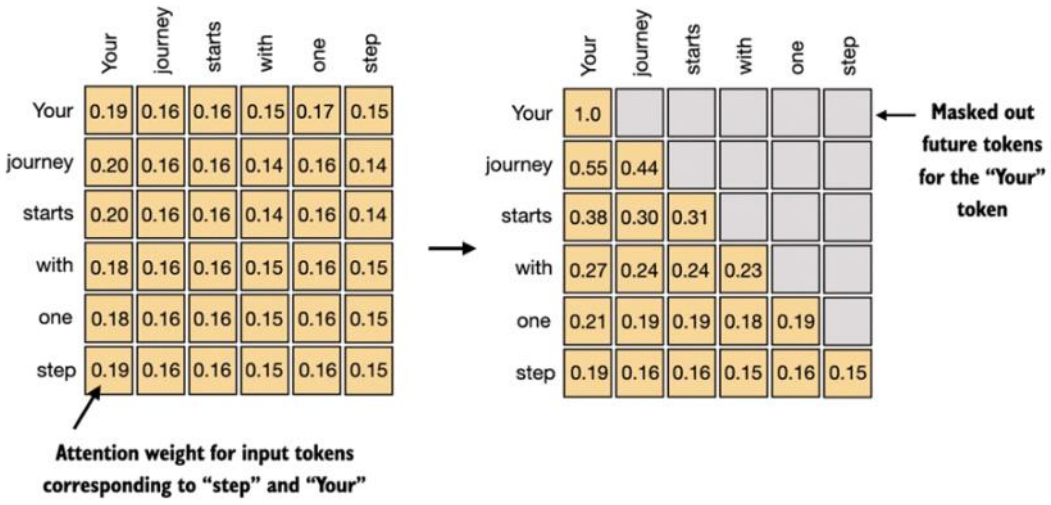
- Query: what this token is looking for
- Key: what this token has to offer
- Value: what this token will contribute if attended to

nn.Linear(input_dim,output_dim) from pytorch creates a linear mapping, we often don't need bias in QKV.

3. Casual Attention / Masked Attention

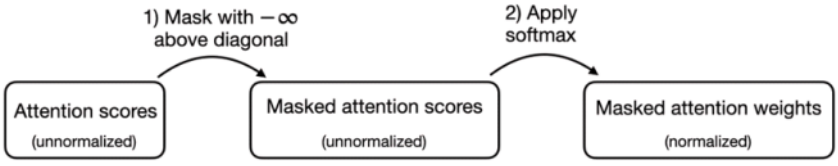
It restricts the model to only consider previous input when processing any give token. So when you calculate the context vect or, which is

essentially the original token reinterpreted with respect to context, you don't consider words later. Note that masking happens before normalization, so each row should still add up to 1.

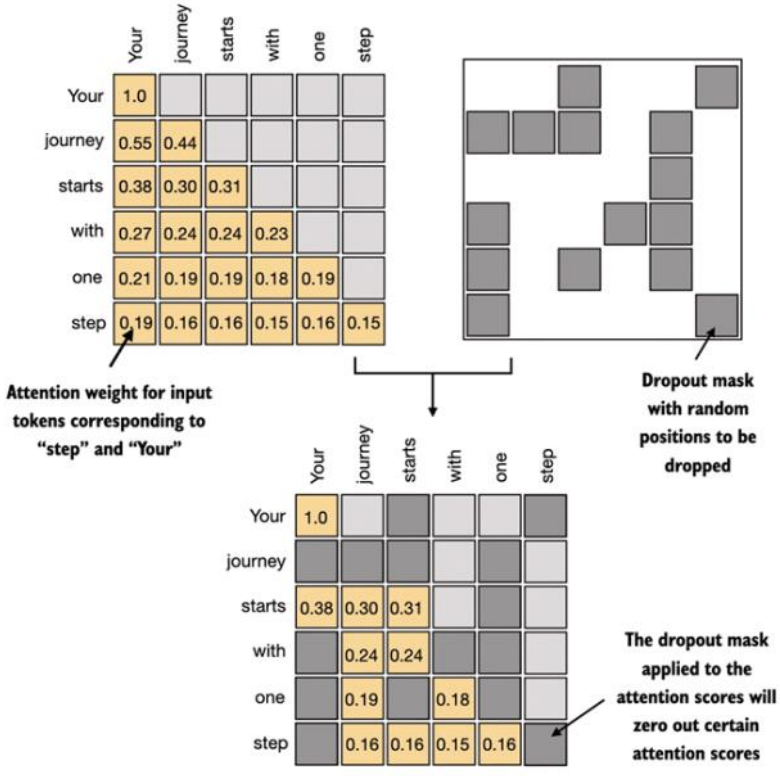


Note you renormalize, mask the weights, and then renormalize again with softmax. But you can also take advantage of the property of softmax, and mask with -inf in the first place

Figure 3.21 A more efficient way to obtain the masked attention weight matrix in causal attention is to mask the attention scores with negative infinity values before applying the softmax function.



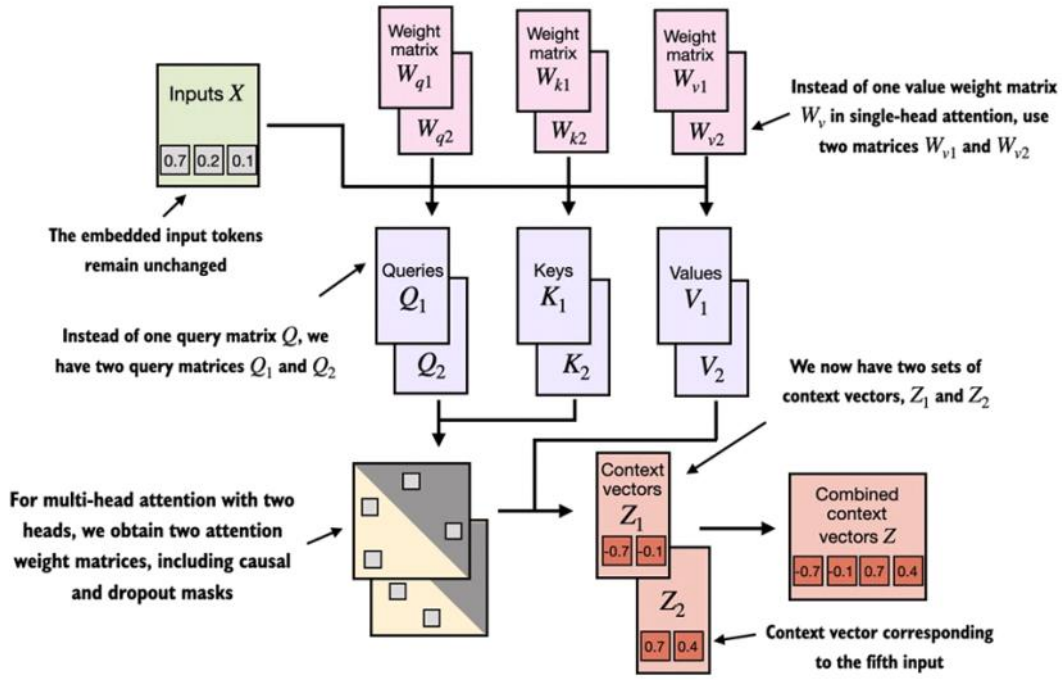
We also use a technique called dropout to prevent overfitting. As shown in the image below, we use a dropout mask to drop 50% of the weights, though in practice we usually drop 10-20%. We use: torch.nn.Dropout(0.5). This is called attention dropout.



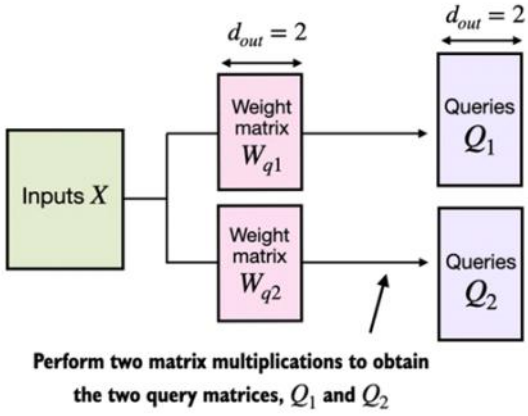
4. Mult-head attention

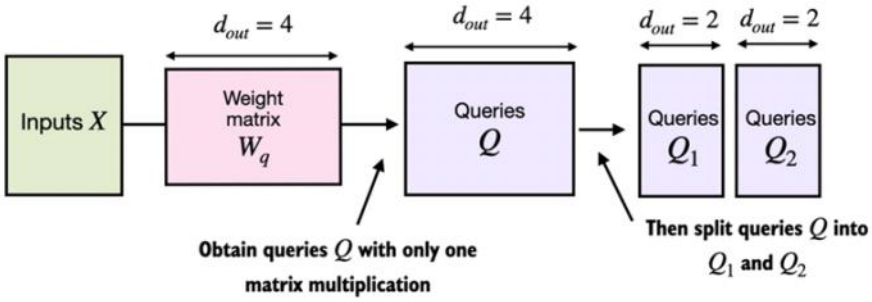
Essentially, it is running the input on multiple single-head attention(what we just implemented) in parallel, where each have their own trainable Q,K,V matrices. Then we get multiple context vectors z. We concatenate these vectors, so output dim would be seq_len

X (d_v*n) (where d_v is value matrix output dimension, and n is the number of heads). Finally we project it back to the **embedding** dimension(d_in = d_out) with the Output projection matrix W^O. This is to preserve the dimensionality throughout the model.



In terms of implementation, this can be done by running single-head attention class multiple times sequentially. But for better performance, we can actually run it in parallel by matrix multiplication. This is done by adding a dimension for Q,K,V matrices and perform matrix multiplication one time, then split the result afterwards. Image below only shows for Q, but we do the same thing on K and V as well.



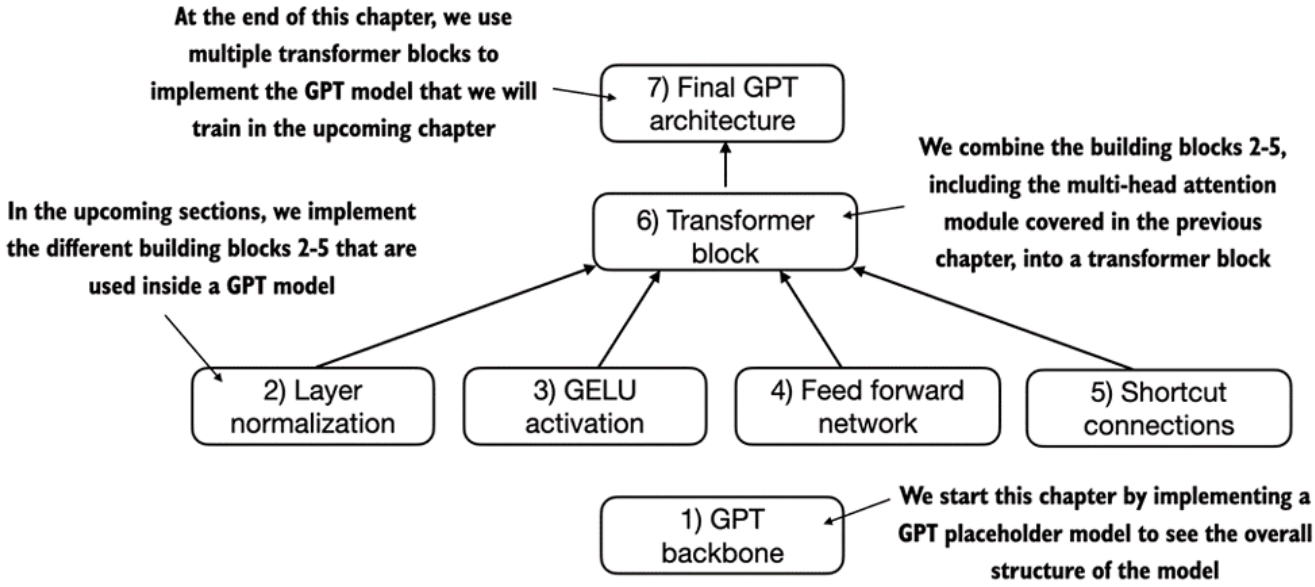


Cross attention: Instead of looking at itself, it seeks context information from another set of data. This approach is often used to model relationships across different modalities.

4. GPT from Scratch

2025年5月14日 13:33

A roadmap:



This forms a backbone of our GPT.

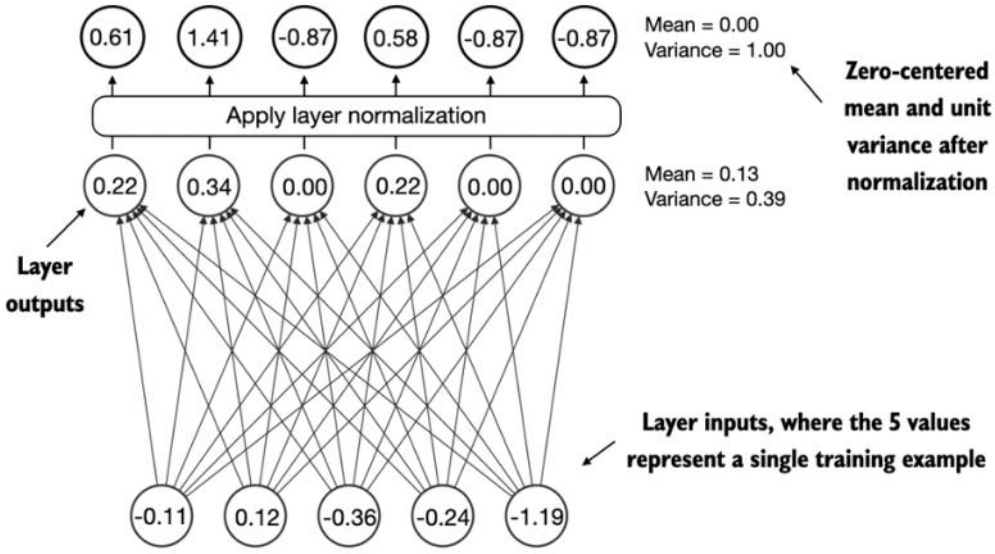
```
def forward(self, in_idx):
    batch_size, seq_len = in_idx.shape
    tok_embeds = self.tok_emb(in_idx)
    pos_embeds = self.pos_emb(torch.arange(seq_len, device=in_idx.device))
    x = tok_embeds + pos_embeds
    x = self.drop_emb(x)
    x = self.trf_blocks(x)
    x = self.final_norm(x)
    logits = self.out_head(x)
    return logits
```

Every line should be self-explanatory. Attention is a part of transformer block. Logits are just raw scores for the possible next-word in vocabulary. We apply softmax and pick a word to complete the model. It's notable that GPT predicts the next word for **each** position.

2. Layer Normalization

Why? One major utility of layer normalization is to avoid gradient explode/vanish as described in RNN. Also sped up the convergence to effective weights.

The main idea is, for the output of each layer(per token, not batch, not sequence), normalize it to have a mean of 0 and variance of 1 (I don't know why this doesn't break the pattern within, unless meaning in vectors is somehow relative, not absolute?). We can also add



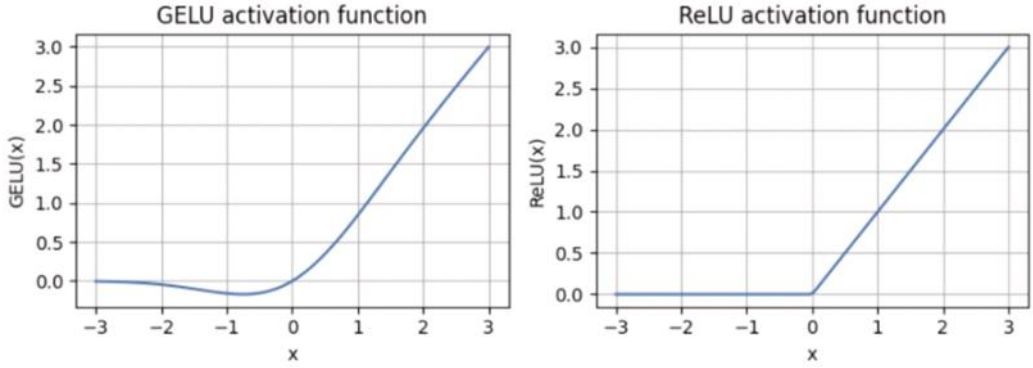
A typical LayerNorm class with learnable weights:

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift
```

dim=-1 refers to the last dimension, cause dim 0 is batch, 1 is seq, and 2(-1) is actual embedding
Eps is a small constant added to var to avoid divide by zero.
Scale and shift are trainable weights so that model can adjust if it thinks doing so would increase performance.

3. Feed forward network with GELU activation function

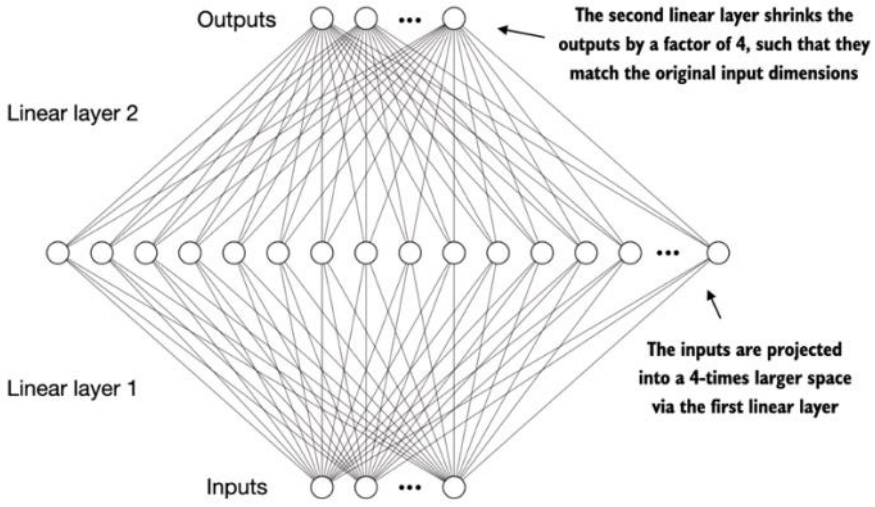
The activation functions are essentially neurons(nodes) in a neural network. There are ReLU(Rectified Linear Unit), Sigmoid, Tanh, Softmax… GELU(Gaussian Error Linear Unit) is the one GPT uses. The formal definition of GELU(x)=x Φ(x) where Φ(x) is the cumulative distribution function of the standard Gaussian distribution. However in practice, we use the approximation:
GELU(x) ≈ 0.5 · x · (1 + tanh[$\sqrt{((2/\pi))}$ · (x + 0.044715 · x^3])



Two differences:
1. It is differentiable on all domain whereas ReLu not differentiable at x=0
2. Negative values can still contribute to the overall activation, which offers more flexibility

Next we in the FFW layer we project it into a higher dimension, in the hope that it can capture more patterns within.

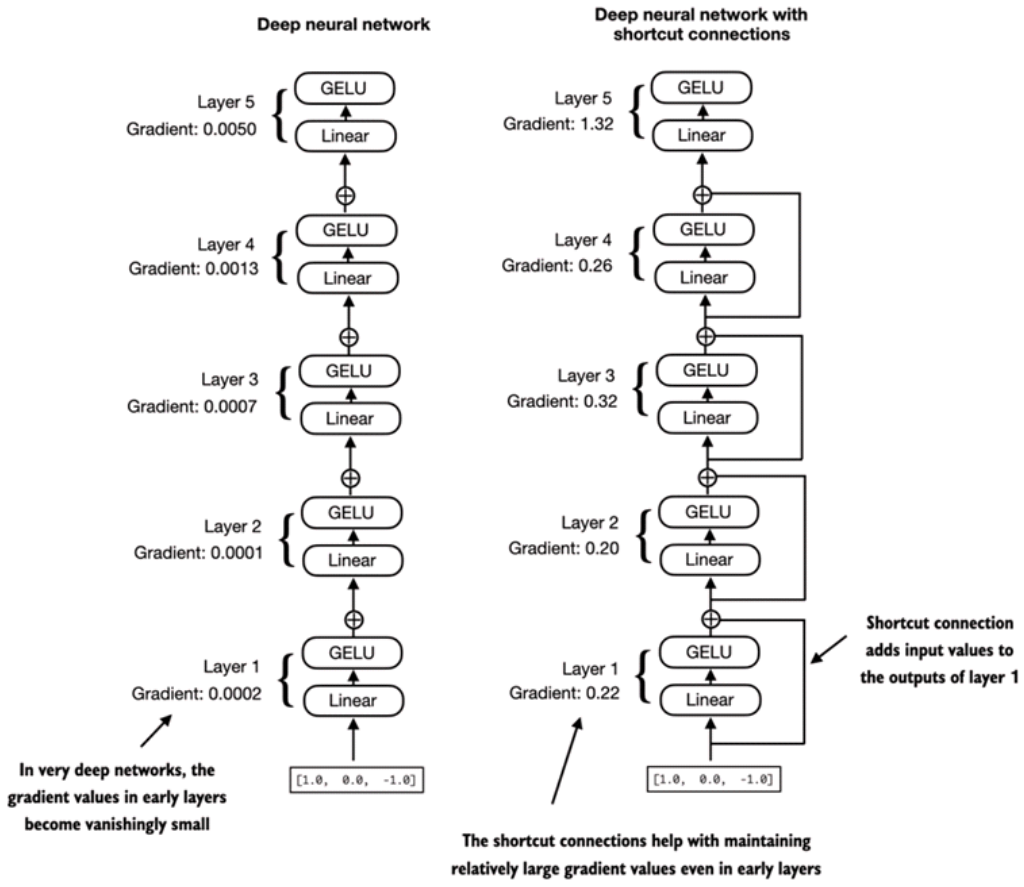
```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

    def forward(self, x):
        return self.layers(x)
```



Here the input is the context vector of one token. It is projected into a 4x larger space and applied GELU for each neuron, then projected back. Note the feedfoward layer works on all tokens, so it doesn't learn token specific patters, but some general insight, more like thinking. We do this in parallel for all tokens using matrix multiplication.

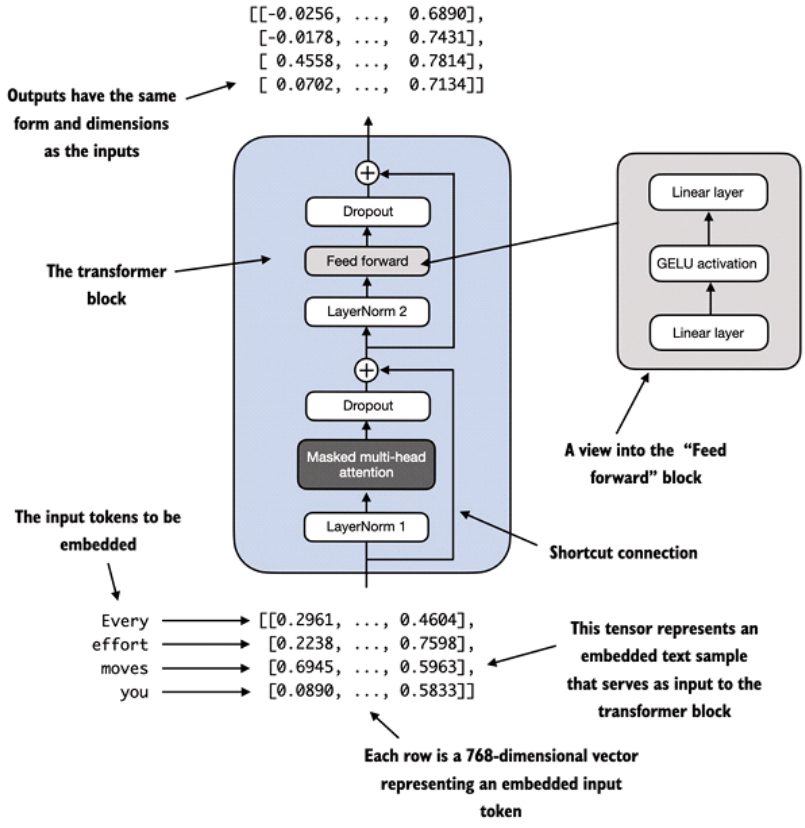
4. Shortcut Connections

A practice used between multiple layers to avoid vanishing gradient problem. Shortcut connections involve adding the inputs of a layer to its outputs, effectively creating an alternate path that bypasses certain layers. Like skipping the layer.



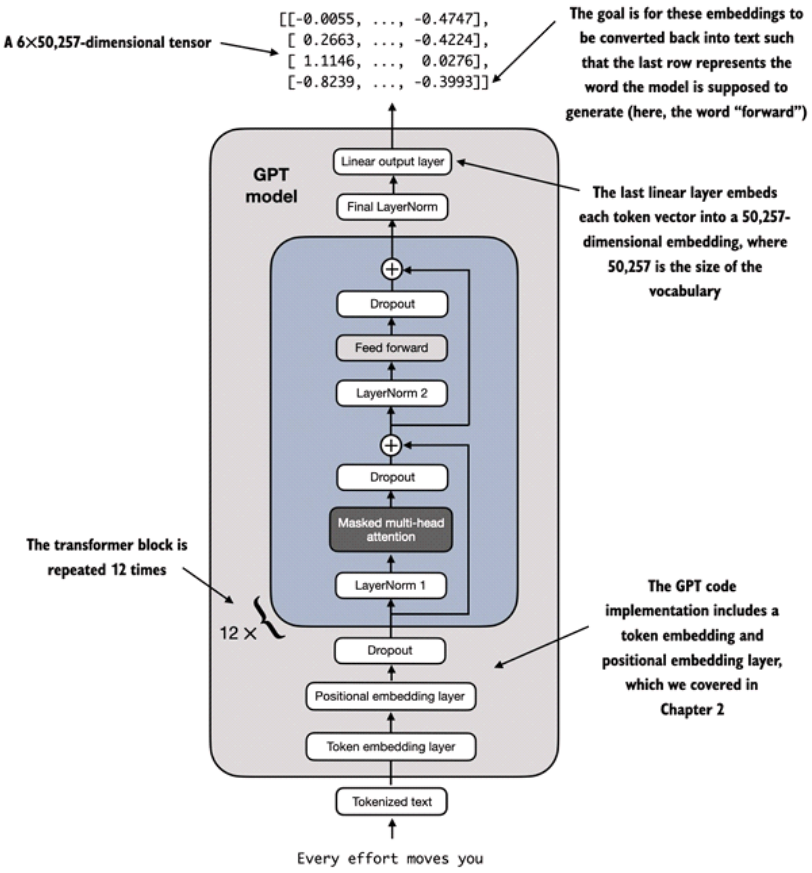
5. Assemble

Finally we can assemble the transformer block with what we have implemented:



Note LayerNorm we talked about before is applied to before each component(attention & ffd). This is called Pre-LayerNorm. Also dropout is applied after each component to avoid overfitting(only during training ofc). Note that this is not the same dropout within the attention which drops the weights, this drops some values of the vector, so some features is lost.

Finally, a GPT model architecture:



Note that the 12 transformer block will have different parameters.
Next word generation is implemented by appending the chosen output token id to a input list and pass it into GPT again, until we get EOF.
A lot of implementation details using pytorch is omitted here since I'm tired rn, but the implementation follows this graph precisely.
In next chapter we learn how to train it with unlabeled data to obtain a foundational model.

Multimodal:

Other type of information is tokenized and embedded. For 4o, it uses a single transformer block and feed in the concatenation of the embedding vectors of different modalities, in the hope that the transformer block learns how to identify and process different modalities.

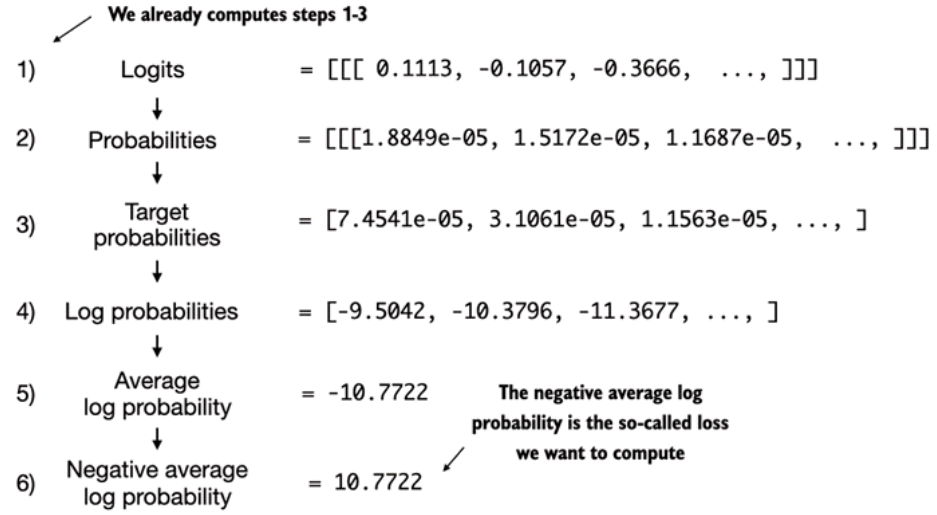
For example, image is first feed into a CNN encoder to turn it lower res feature map, then flattened and then embedded. (VQ-VAE)

Intelligence don't emerge out of language, language emerge out of intelligence. Multimodal transformer trained with RL seems the way to intelligence.

5. Pretraining on Unlabeled Data

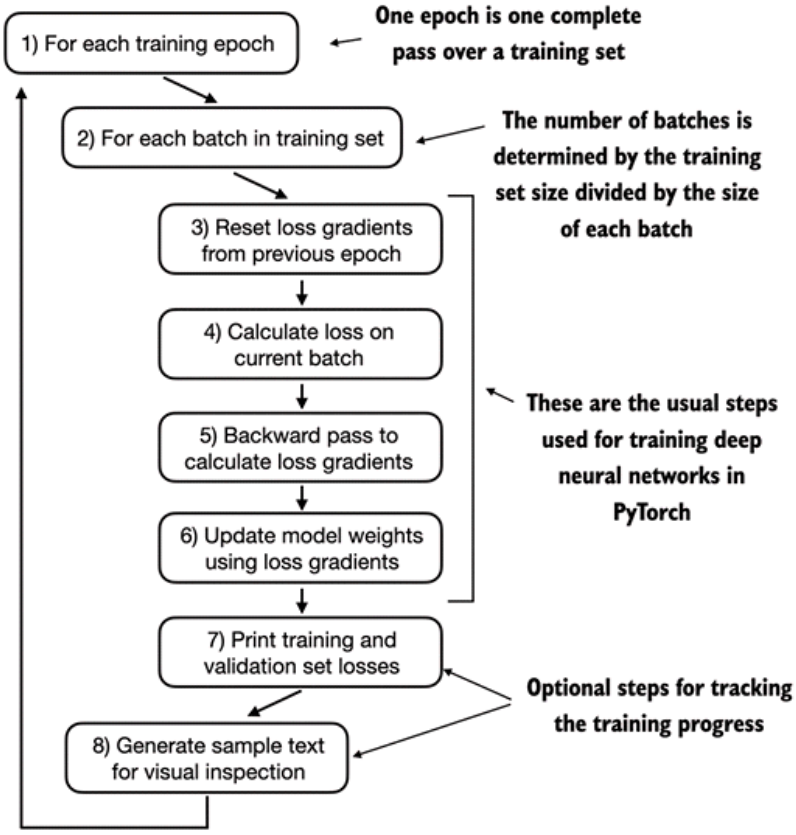
2025年5月15日 13:01

Recall from chapter 2 that we split data into input and target using the sliding window approach ([1,2,3,4] -> [2,3,4,5]). We can't really evaluate on 5 easily, but we can calculate loss for 2,3,4 as they are the "correct" answer. Below is a diagram of the loss calculation:



In 2) we take the probability of the correct answer.
We take log because its more manageable in mathematical optimization.
Taking the negative of the avg probability is known as cross entropy in deep learning. 6) is the cross entropy loss.
We calculate error for every batch, and take the average error.
Note that batching is just to improve performance by utilizing parallel computing power of GPU.
Our goal is to minimize this error.
Another metric is perplexity: torch.exp(loss). This is said to be more interpretable since it takes account of the vocab size.
Training loss: Average loss computed on the training dataset over an epoch. (on data it has seen)
Validation loss: On a held-out validation dataset it has never seen before. (may decrease due to overfit)

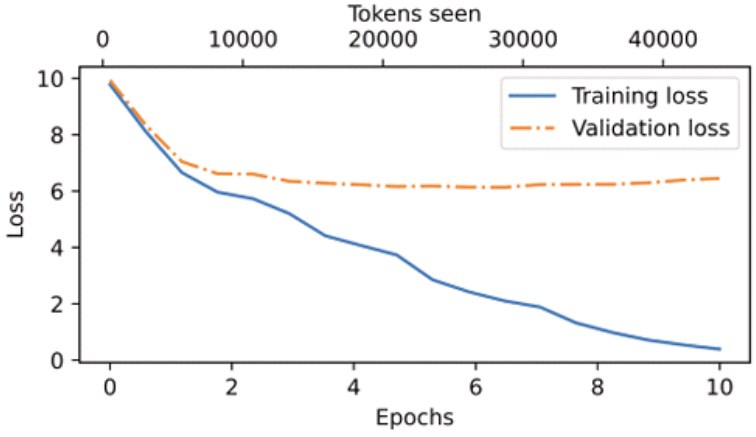
Next we are interested in the actual training using backprop, this is a typical training loop:



1,2): An epoch is a full pass of all the batches, note that the loop 3-6 loops for different batches. The model only looks at data per epoch times.
3): Reset gradient of the previous batch (sentence cutting is a minor issue due to scale)
4): Feed the current batch into model, gets prediction, calculate loss
5): Use backpropagation to compute how much each weight in the model contribute to the error, this returns the gradient of the loss with respect to each parameter. It assigns loss gradient to every parameters
6): Applies the gradients to the model's parameters using an optimizer (e.g., Adam, SGD)

Note GPU and large vram allow us to run each batch in parallel (so seq_len * batch_len times). Also it's important to shuffle the batches to prevent forgetting, since every batch after will modify the parameters. (I don't see why this does not cause catastrophic forgetting)

Doing multiple epoch decrease train loss, but risks overfitting, resulting in a bad validation loss, the industry standard is 1-2 epochs. But in fine-tuning we often do 5-20 or more, since dataset is much smaller and a bit of overfitting is sometimes beneficial.



Next we will talk about text generation strategies/decoding strategies (irrelevant to training).
Our model right now is a deterministic one, meaning it will generate the same message for same input.
There are two concepts to control randomness and diversity: temperature scaling and top-k sampling

Temperature scaling: Basically replace the argmax function, which chooses the highest probability, with multinomial which chooses probabilistically based on distribution. Furthermore, we divide the logits with a temperature (constant > 0), so the distribution is more uniform (or sharper if t<1). The downside of the this approach is that it might lead to grammatically incorrectness.

Top-k sampling: A smarter modification to temperature scaling, where we only consider the top k logits, and masking all other ones with -inf, so that after soft-max, those have probability of 0. This prevents non-sensical generations and only keeps the most correct ones.

We now talk about how to save and load a pretrained model, such as GPT2. To save an existing model:

```
torch.save(model.state_dict(), "model.pth")
```

If we want to train it later, we also want to save the optimizer too.
To load a model:

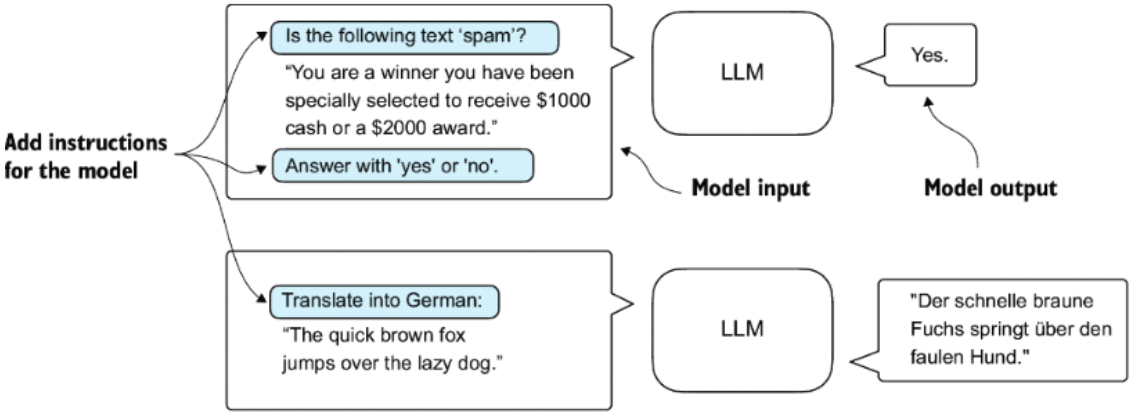
```
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(torch.load("model.pth"))
model.eval()
```

How to download GPT2 from open ai is included in code

6. Fine-tuning for classification

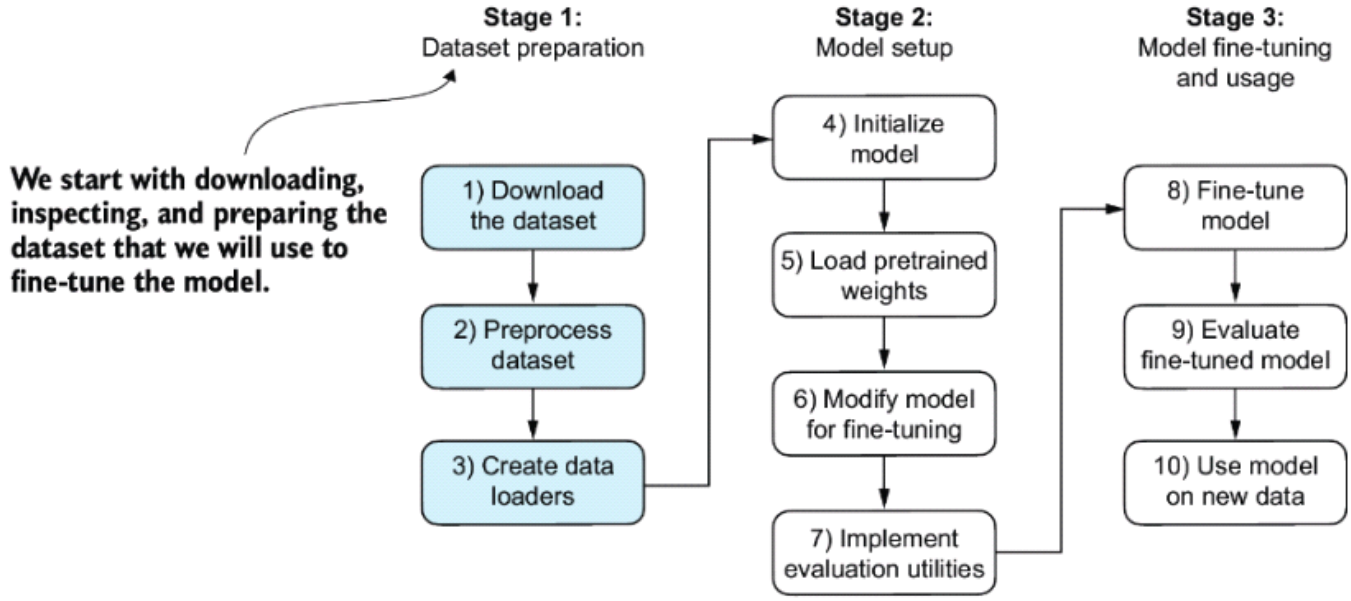
2025年6月9日 12:24

Two common ways are instruction fine-tuning and classification fine-tuning:



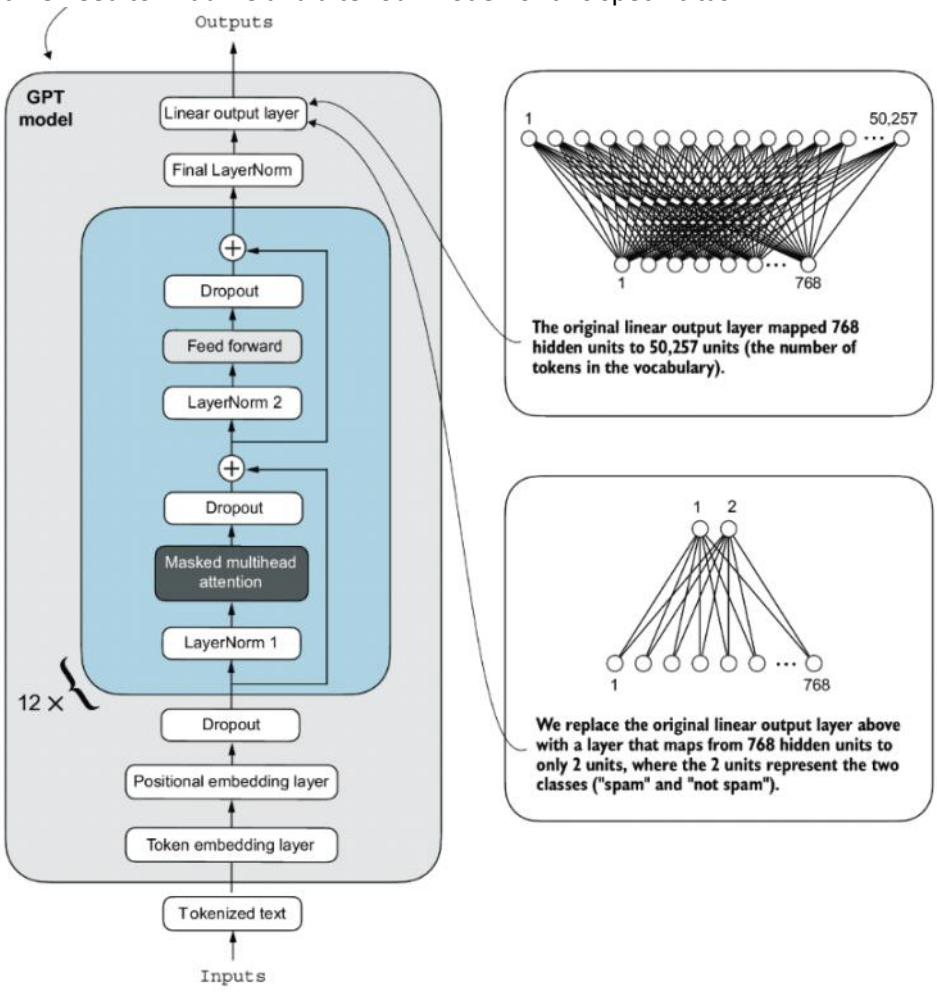
Classification fine-tuned models takes only message input, and output spam or not spam, whereas Instruction models are required to execute a boarder range of tasks. Classification models are more specialized, smaller, and cheaper to train.

Three stages in classification fine-tuning:



Unlike pretraining, fine-tuning is more "fine". So when we batch dataset we use padding to make sure batch doesn't split message. We use EOT token. In our case, the longest message size is 120, so we batch per 120 tokens, i.e. max sequence length = 120. Data preparation is pretty standard, basically split into 7:2:1 for training, validation and evaluation sets and batch them as discussed above. Also give them labels of 1 and 0 for spam and not spam.

Next we need to initialize and alter our model for this specific task:



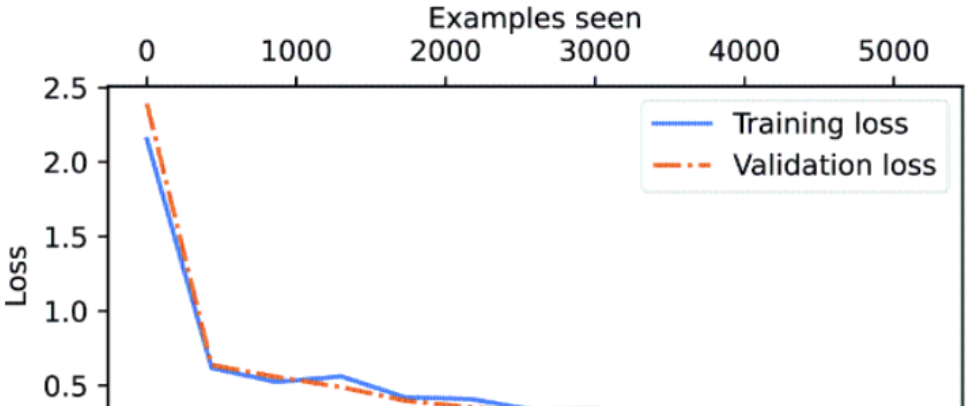
We could've output 1 instead, but that would break the error calculation we have.

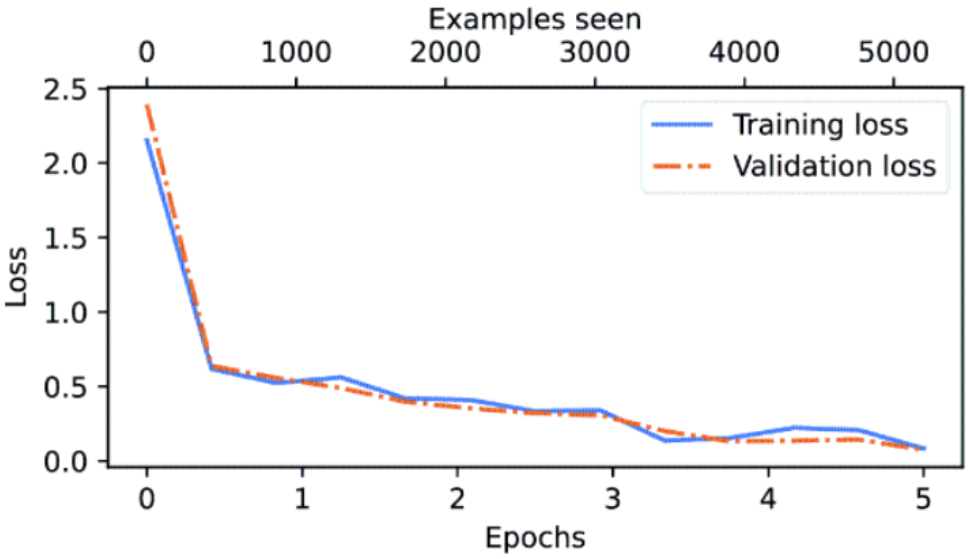
Since we started with a pretrained model, which captures basic language structures And meanings. So fine-tuning only the last few layer is often enough for specific tasks. Here we freeze all the layers except the final linear output layer, final LayerNorm and the last transformer block.

Note that we still use the last logit, since it includes all the information in the input.

Accuracy is calculated by correct predictions / total predictions
Error is calculated by cross entropy, just as before

The training loop is pretty much identical to pretraining. Though one crucial difference is that fine-tuning is often trained for multiple epochs.

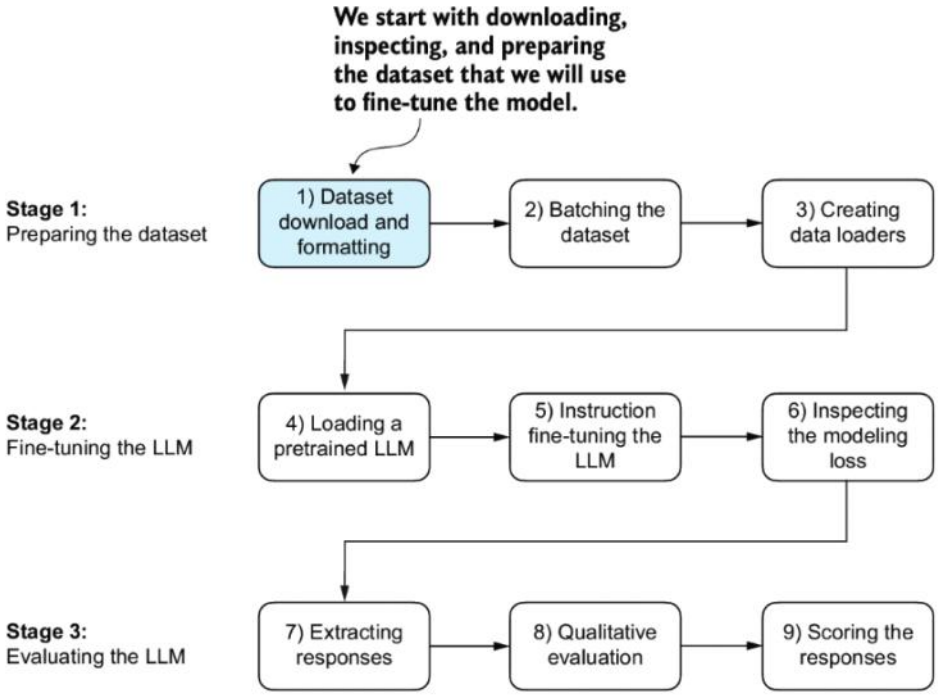




This is because during pretraining, the model is expected to learn a lot, that is, the entropy of the dataset is high, whereas in fine-tuning, the entropy is much smaller. Also, we freeze the parameter during fine-tuning so it will have less parameters to overfit, and this also means its much cheaper to run multiple epochs.

7. Fine-tuning to follow instructions

2025年8月18日　13:31



Three stage training process.

Firstly data formatting and batching:

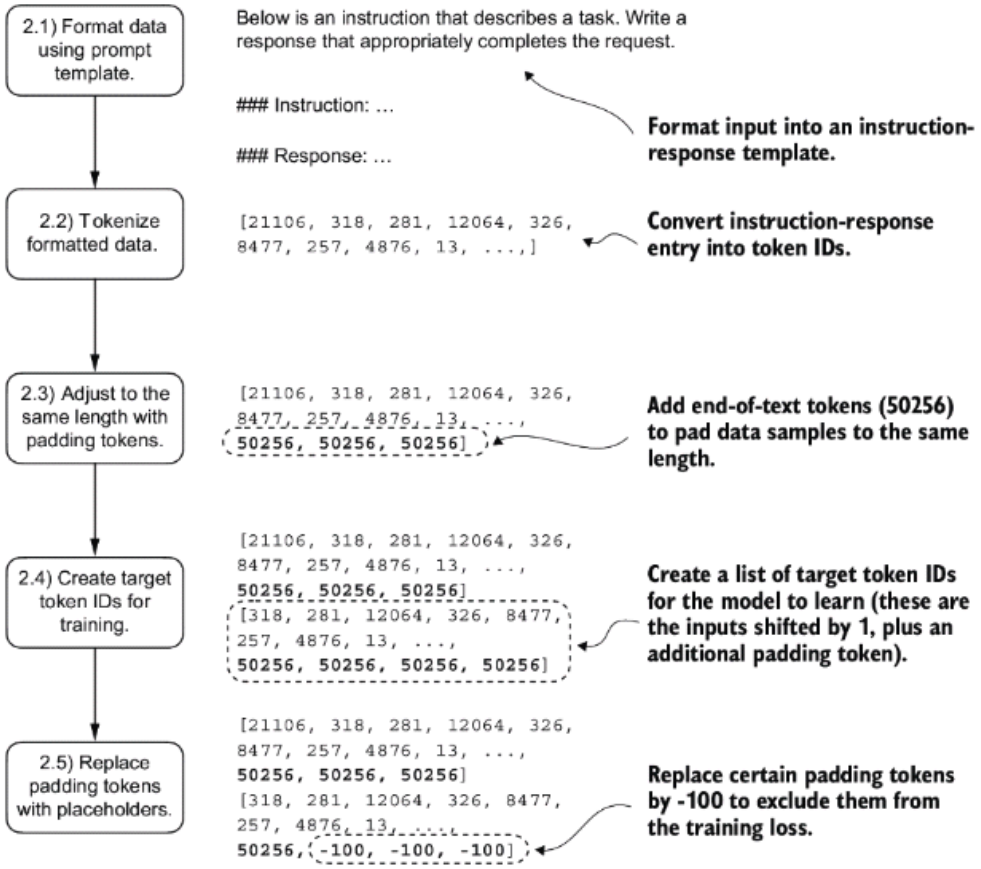


Figure 7.6 The five substeps involved in implementing the batching process: (2.1) applying the prompt template, (2.2) using tokenization from previous chapters, (2.3) adding padding tokens, (2.4) creating target token IDs, and (2.5) replacing `-100` placeholder tokens to mask padding tokens in the loss function

chapters, (2.3) adding padding tokens, (2.4) creating target token IDs, and (2.5) replacing `-100` placeholder tokens to mask padding tokens in the loss function.

The training process is similar to chapter 5, we feed in formatted instruction and response pairs and calculate loss and do backprop. The Alpaca dataset is one of the most popular publicly available instruction sets.

To evaluate our model, we can use human response or, more efficiently, another LLM to evaluate. He uses Ollama with llama3, which is an open source application for text generation for LLMs and an open LLM from Meta.

To further improve our model's performance, we can explore various strategies, such as

- Adjusting the hyperparameters during fine-tuning, such as the learning rate, batch size, or number of epochs
- Increasing the size of the training dataset or diversifying the examples to cover a broader range of topics and styles
- Experimenting with different prompts or instruction formats to guide the model's responses more effectively
- Using a larger pretrained model, which may have greater capacity to capture complex patterns and generate more accurate responses

2025/10/1