CHAPTER

9 Masked Language Models

Larvatus prodeo [Masked, I go forward]
Descartes

In the previous two chapters we introduced the transformer and saw how to pretrain a transformer language model as a **causal** or left-to-right language model. In this chapter we'll introduce a second paradigm for pretrained language models, the **bidirectional transformer** encoder, and the most widely-used version, the **BERT** family of models (Devlin et al., 2019). This model is trained via **masked language modeling**, where instead of predicting the following word, we mask a word in the middle and ask the model to guess the word given the words on both sides. This method thus allows the model to see both the right and left context.

BERT

masked language modeling

We also introduced **fine-tuning** in the prior chapter. Here we describe a new kind of fine-tuning, in which we take the transformer network learned by these pretrained models, add a neural net classifier after the top layer of the network, and train it on some additional labeled data to perform some downstream task like named entity tagging or natural language inference. As before, the intuition is that the pretraining phase learns a language model that instantiates rich representations of word meaning, that thus enables the model to more easily learn ('be finetuned to') the requirements of a downstream language understanding task. This aspect of the pretrain-finetune paradigm is an instance of what is called **transfer learning** in machine learning: the method of acquiring knowledge from one task or domain, and then applying it (transferring it) to solve a new task.

fine-tuning

transfer learning

9.1 Three architectures for language models

The architecture we described in Chapter 7 for a left-to-right or autoregressive language model is actually only one of three common LM architectures.

The three architectures are the **encoder**, the **decoder**, and the **encoder-decoder**. Fig. 9.1 gives a schematic picture of the three.

The **decoder** is the architecture we've introduced earlier. It takes as input a series of tokens, and iteratively generates an output token one at a time. The decoder is the architecture used to create large language models like GPT, Claude, Llama, and Mistral. The information flow in decoders goes left-to-right, meaning that the model predicts the next word only from the prior words. Decoders are generative models, meaning that, given input tokens, they generate novel output tokens.

decoder

The **encoder** takes as input a sequence of tokens and outputs a vector representation for each token. Encoders are usually masked language models, meaning they are trained by masking out a word, and learning to predict it by looking at surrounding words on both sides. Masked language models like BERT, RoBERTA, and others in the BERT family are encoder models. Encoder models are not generative models; they aren't used to generate text. Instead encoder models are often used to

encoder

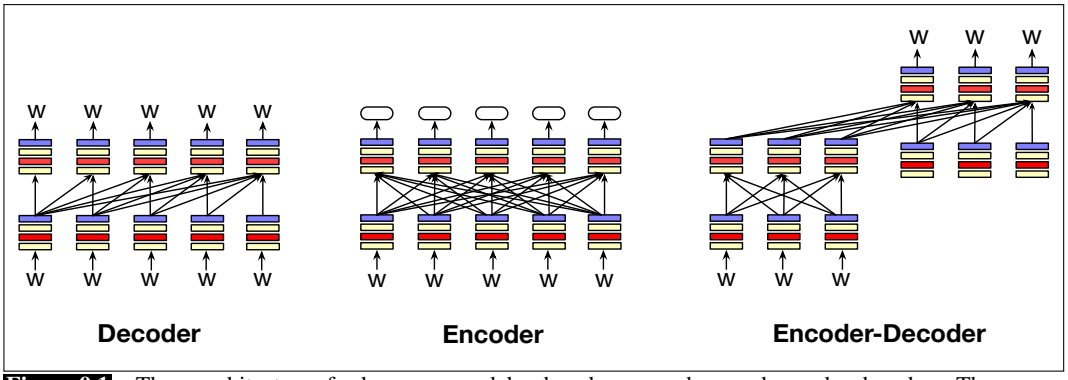

Figure 9.1 Three architectures for language models: decoders, encoders, and encoder-decoders. The arrows sketch out the information flow in the three architectures. Decoders take tokens as input and generate tokens as output. Encoders take tokens as input and produce an encoding (a vector representation of each token) as output. Encoder-decoders take tokens as input and generate a series of tokens as output.

create classifiers, for example where the input is text and the output is a label, for example for sentiment or topic or other classes. This is done by fine-tuning them (training them on supervised data). We'll introduce encoder models in this chapter.

encoder-decoder

The **encoder-decoder** takes as input a sequence of tokens and outputs a series of tokens. What makes it different than the decoder-only models, is that an encoder-decoder has a much looser relationship between the input tokens and the output tokens, and they are used to map between different kinds of tokens. That is, in an encoder-decoder, the output tokens might be from a very different token-set or be a much longer or shorter sequence than the input token sequence. For example encoder-decoder architectures are used for machine translation, where the input tokens are in one language and the output tokens (probably more or less of them) are in another language. Encoder-decoder architectures are also used for speech recognition, where the input is tokens representing speech, and the output is tokens representing text. We'll introduce the encoder-decoder architecture for machine translation in Chapter 13, and for speech recognition in Chapter 16.

These three architectures can be built out of many kinds of neural networks, but we'll continue to assume here the use of the transformer.

9.2 Bidirectional Transformer Encoders

Let's begin by introducing the bidirectional transformer encoder that underlies models like BERT and its descendants like **RoBERTa** (Liu et al., 2019). The left-to-right nature of the decoder LLMs of Chapter 7 can be a limitation, because there are tasks for which it would be useful, when processing a token, to be able to peek at future tokens. This is especially true for **sequence labeling** tasks in which we want to tag each token with a label, such as the **named entity tagging** task we'll introduce in Section 9.5, or tasks like part-of-speech tagging or parsing that come up in later chapters.

The **bidirectional** encoders that we introduce here are a different kind of beast than causal models. The causal models of Chapter 7 are generative models, designed to easily generate the next token in a sequence. But the focus of bidirectional encoders is instead on computing contextualized representations of the input

tokens. Bidirectional encoders use self-attention to map sequences of input embeddings $(\mathbf{x}_1, ..., \mathbf{x}_n)$ to sequences of output embeddings of the same length $(\mathbf{h}_1, ..., \mathbf{h}_n)$, where the output vectors have been contextualized using information from the entire input sequence. These output embeddings are contextualized representations of each input token that are useful across a range of applications where we need to do a classification or a decision based on the token in context.

Remember that we said the models of Chapter 7 are sometimes called **decoder-only**, because they correspond to the decoder part of the encoder-decoder model we will introduce in Chapter 13. By contrast, the masked language models of this chapter are sometimes called **encoder-only**, because they produce an encoding for each input token but generally aren't used to produce running text by decoding/sampling. Although it's possible to use masked language models for some generation tasks, for example in recent work involving diffusion, we don't discuss this yet here. We'll focus on the use of masked language models for interpretative tasks.

9.2.1 The architecture for bidirectional masked models

Let's first discuss the overall architecture. Bidirectional transformer-based language models differ in two ways from the causal transformers in the previous chapters. The first is that the attention function isn't causal; the attention for a token i can look at following tokens $i+1$ and so on. The second is that the training is slightly different since we are predicting something in the middle of our text rather than at the end. We'll discuss the first here and the second in the following section.

Fig. 9.2a, reproduced here from Chapter 7, shows the information flow in the left-to-right approach of Chapter 7. The attention computation at each token is based on the preceding (and current) input tokens, ignoring potentially useful information located to the right of the token under consideration. Bidirectional encoders overcome this limitation by allowing the attention mechanism to range over the entire input, as shown in Fig. 9.2b.

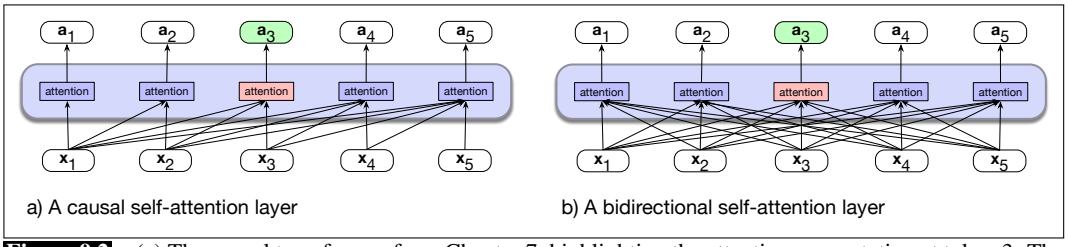

Figure 9.2 (a) The causal transformer from Chapter 7, highlighting the attention computation at token 3. The attention value at each token is computed using only information seen earlier in the context. (b) Information flow in a bidirectional attention model. In processing each token, the model attends to all inputs, both before and after the current one. So attention for token 3 can draw on information from following tokens.

The implementation is very simple! We simply remove the attention masking step that we introduced in Eq. 7.35. Recall from Chapter 7 that we had to mask the $\mathbf{QK}^\intercal$ matrix for causal transformers so that attention couldn't look at future tokens (repeated from Eq. 7.35 for a single attention head):

$$\textbf{head} = \text{softmax}\left(\text{mask}\left(\frac{\mathbf{QK}^\intercal}{\sqrt{d_k}}\right)\right)\mathbf{V} \tag{9.1}$$

Fig. 9.3 shows the masked version of $\mathbf{QK}^\intercal$ and the unmasked version. For bidirectional attention, we use the unmasked version of Fig. 9.3b. Thus the attention

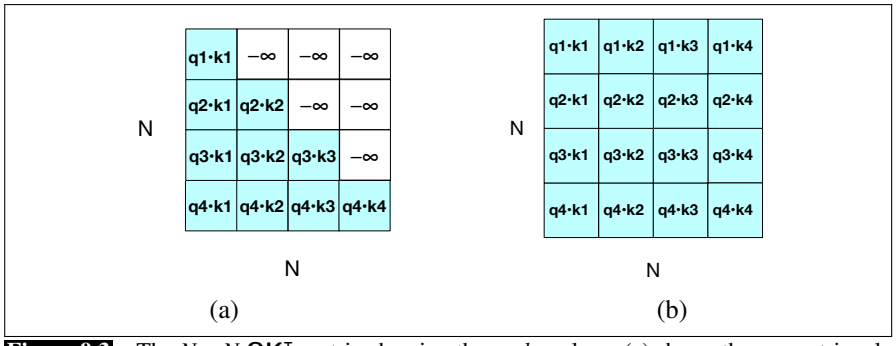

Figure 9.3 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing the $q_i \cdot k_j$ values. (a) shows the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero), while (b) shows the unmasked version.

computation for bidirectional attention is exactly the same as Eq. 9.1 but with the mask removed:

$$\textbf{head} = \text{softmax}\left(\frac{\mathbf{QK}^\intercal}{\sqrt{d_k}}\right)\mathbf{V} \tag{9.2}$$

Otherwise, the attention computation is identical to what we saw in Chapter 7, as is the transformer block architecture (the feedforward layer, layer norm, and so on). As in Chapter 7, the input is also a series of subword tokens, usually computed by one of the 3 popular tokenization algorithms (including the BPE algorithm that we already saw in Chapter 2 and two others, the WordPiece algorithm and the SentencePiece Unigram LM algorithm). That means every input sentence first has to be tokenized, and all further processing takes place on subword tokens rather than words. This will require, as we'll see in the third part of the textbook, that for some NLP tasks that require notions of words (like parsing) we will occasionally need to map subwords back to words.

To make this more concrete, the original English-only bidirectional transformer encoder model, BERT (Devlin et al., 2019), consisted of the following:

- An English-only subword vocabulary consisting of 30,000 tokens generated using the WordPiece algorithm (Schuster and Nakajima, 2012).
- Input context window N=512 tokens, and model dimensionality d=768
- So $\mathbf{X}$, the input to the model, is of shape $[N \times d] = [512 \times 768]$.
- L=12 layers of transformer blocks, each with A=12 (bidirectional) multihead attention layers.
- The resulting model has about 100M parameters.

The larger multilingual XLM-RoBERTa model, trained on 100 languages, has

- A multilingual subword vocabulary with 250,000 tokens generated using the SentencePiece Unigram LM algorithm (Kudo and Richardson, 2018).
- Input context window N=512 tokens, and model dimensionality d=1024, hence $\mathbf{X}$, the input to the model, is of shape $[N \times d] = [512 \times 1024]$.
- L=24 layers of transformer blocks, with A=16 multihead attention layers each
- The resulting model has about 550M parameters.

Note that 550M parameters is relatively small as large language models go (Llama 3 has 405B parameters, so is 3 orders of magnitude bigger). Indeed, masked language models tend to be much smaller than causal language models.

9.3 Training Bidirectional Encoders

We trained causal transformer language models in Chapter 7 by making them iteratively predict the next word in a text. But eliminating the causal mask in attention makes the guess-the-next-word language modeling task trivial—the answer is directly available from the context—so we're in need of a new training scheme. Instead of trying to predict the next word, the model learns to perform a fill-in-the-blank task, technically called the **cloze task** (Taylor, 1953). To see this, let's return to the motivating example from Chapter 3. Instead of predicting which words are likely to come next in this example:

cloze task

`The water of Walden Pond is so beautifully ____`

we're asked to predict a missing item given the rest of the sentence.

`The ____ of Walden Pond is so beautifully ...`

That is, given an input sequence with one or more elements missing, the learning task is to predict the missing elements. More precisely, during training the model is deprived of one or more tokens of an input sequence and must generate a probability distribution over the vocabulary for each of the missing items. We then use the cross-entropy loss from each of the model's predictions to drive the learning process.

This approach can be generalized to any of a variety of methods that corrupt the training input and then asks the model to recover the original input. Examples of the kinds of manipulations that have been used include masks, substitutions, reorderings, deletions, and extraneous insertions into the training text. The general name for this kind of training is called **denoising**: we corrupt (add noise to) the input in some way (by masking a word, or putting in an incorrect word) and the goal of the system is to remove the noise.

denoising

9.3.1 Masking Words

Masked Language Modeling

Let's describe the **Masked Language Modeling** (MLM) approach to training bidirectional encoders (Devlin et al., 2019). As with the language model training methods we've already seen, **MLM** uses unannotated text from a large corpus. In MLM training, the model is presented with a series of sentences from the training corpus in which a percentage of tokens (15% in the BERT model) have been randomly chosen to be manipulated by the masking procedure. Given an input sentence `lunch was delicious` and assume we randomly chose the 3rd token `delicious` to be manipulated,

- 80% of the time: The token is replaced with the special vocabulary token named `[MASK]`, e.g. `lunch was delicious` → `lunch was [MASK]`.
- 10% of the time: The token is replaced with another token, randomly sampled from the vocabulary based on token unigram probabilities. e.g. `lunch was delicious` → `lunch was gasp`.
- 10% of the time: the token is left unchanged. e.g. `lunch was delicious` → `lunch was delicious`.

We then train the model to guess the correct token for the manipulated tokens. Why the three possible manipulations? Adding the `[MASK]` token creates a mismatch between pretraining and downstream fine-tuning or inference, since when we employ the MLM model to perform a downstream task, we don't use any `[MASK]` tokens. If

we just replaced tokens with the [MASK], the model might only predict tokens when it sees a [MASK], but we want the model to try to always predict the input token.

To train the model to make the prediction, the original input sequence is tokenized using a subword model and tokens are sampled to be manipulated. Word embeddings for all of the tokens in the input are retrieved from the **E** embedding matrix and combined with positional embeddings to form the input to the transformer, passed through the stack of bidirectional transformer blocks, and then the language modeling head. The MLM training objective is to predict the original inputs for each of the masked tokens and the cross-entropy loss from these predictions drives the training process for all the parameters in the model. That is, all of the input tokens play a role in the self-attention process, but only the sampled tokens are used for learning.

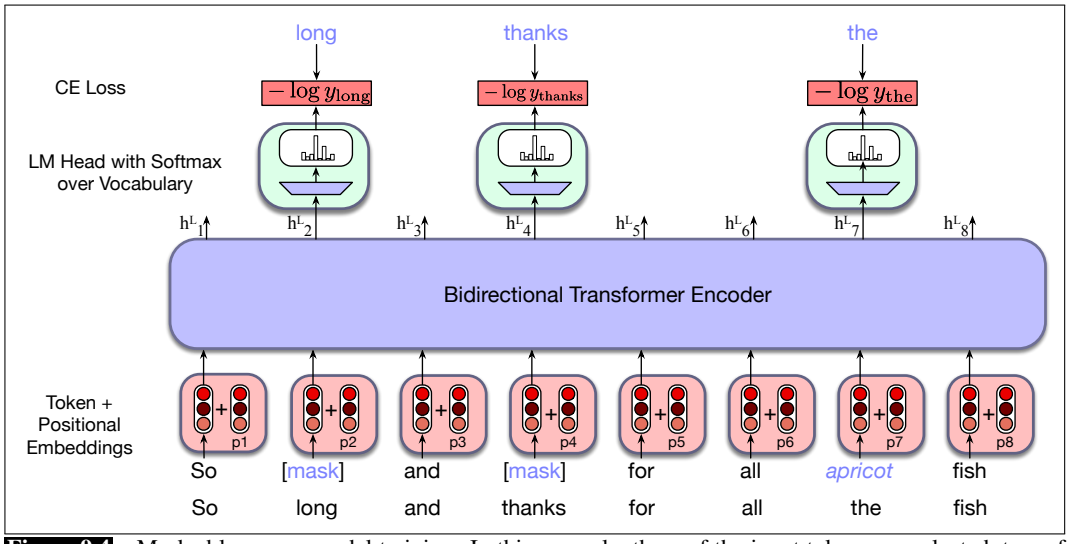

Figure 9.4 Masked language model training. In this example, three of the input tokens are selected, two of which are masked and the third is replaced with an unrelated word. The probabilities assigned by the model to these three items are used as the training loss. The other 5 tokens don't play a role in training loss.

Fig. 9.4 illustrates this approach with a simple example. Here, *long*, *thanks* and *the* have been sampled from the training sequence, with the first two masked and *the* replaced with the randomly sampled token *apricot*. The resulting embeddings are passed through a stack of bidirectional transformer blocks. Recall from Section 7.5 in Chapter 7 that to produce a probability distribution over the vocabulary for each of the masked tokens, the **language modeling head** takes the output vector $\mathbf{h}_i^L$ from the final transformer layer L for each masked token i, multiplies it by the unembedding layer $\mathbf{E}^T$ to produce the logits $\mathbf{u}$, and then uses softmax to turn the logits into probabilities $\mathbf{y}$ over the vocabulary:

$$\mathbf{u}_i = \mathbf{h}_i^L \mathbf{E}^T \tag{9.3}$$

$$\mathbf{y}_i = \text{softmax}(\mathbf{u}_i) \tag{9.4}$$

With a predicted probability distribution for each masked item, we can use cross-entropy to compute the loss for each masked item—the negative log probability assigned to the actual masked word, as shown in Fig. 9.4. More formally, for a given vector of input tokens in a sentence or batch $\mathbf{x}$, let the set of tokens that are masked be M, the version of that sentence with some tokens replaced by masks be

$\mathbf{x}^{mask}$, and the sequence of output vectors be $\mathbf{h}$. For a given input token x_i, such as the word *long* in Fig. 9.4, the loss is the probability of the correct word *long*, given $\mathbf{x}^{mask}$ (as summarized in the single output vector $\mathbf{h}_i^L$):

$$L_{MLM}(x_i) = -\log P(x_i|\mathbf{h}_i^L)$$

The gradients that form the basis for the weight updates are based on the average loss over the sampled learning items from a single training sequence (or batch of sequences).

$$L_{MLM} = -\frac{1}{|M|}\sum_{i \in M} \log P(x_i|\mathbf{h}_i^L)$$

Note that only the tokens in M play a role in learning; the other words play no role in the loss function, so in that sense BERT and its descendents are inefficient; only 15% of the input samples in the training data are actually used for training weights.[1]

9.3.2 Next Sentence Prediction

The focus of mask-based learning is on predicting words from surrounding contexts with the goal of producing effective word-level representations. However, an important class of applications involves determining the relationship between pairs of sentences. These include tasks like paraphrase detection (detecting if two sentences have similar meanings), entailment (detecting if the meanings of two sentences entail or contradict each other) or discourse coherence (deciding if two neighboring sentences form a coherent discourse).

Next Sentence Prediction

To capture the kind of knowledge required for applications such as these, some models in the BERT family include a second learning objective called **Next Sentence Prediction** (NSP). In this task, the model is presented with pairs of sentences and is asked to predict whether each pair consists of an actual pair of adjacent sentences from the training corpus or a pair of unrelated sentences. In BERT, 50% of the training pairs consisted of positive pairs, and in the other 50% the second sentence of a pair was randomly selected from elsewhere in the corpus. The NSP loss is based on how well the model can distinguish true pairs from random pairs.

To facilitate NSP training, BERT introduces two special tokens to the input representation (tokens that will prove useful for fine-tuning as well). After tokenizing the input with the subword model, the token `[CLS]` is prepended to the input sentence pair, and the token `[SEP]` is placed between the sentences and after the final token of the second sentence. There are actually two more special tokens, a 'First Segment' token, and a 'Second Segment' token. These tokens are added in the input stage to the word and positional embeddings. That is, each token of the input $\mathbf{X}$ is actually formed by summing 3 embeddings: word, position, and first/second segment embeddings.

During training, the output vector h^L_{CLS} from the final layer associated with the `[CLS]` token represents the next sentence prediction. As with the MLM objective, we add a special head, in this case an NSP head, which consists of a learned set of classification weights $\mathbf{W_{NSP}} \in \mathbb{R}^{d\times 2}$ that produces a two-class prediction from the raw `[CLS]` vector h^L_{CLS}:

$$\mathbf{y}_i = \text{softmax}(\mathbf{h}^L_{\text{CLS}}\mathbf{W_{NSP}})$$

[1] ELECTRA, another BERT family member, does use all examples for training (Clark et al., 2020b).

Cross entropy is used to compute the NSP loss for each sentence pair presented to the model. Fig. 9.5 illustrates the overall NSP training setup. In BERT, the NSP loss was used in conjunction with the MLM training objective to form final loss.

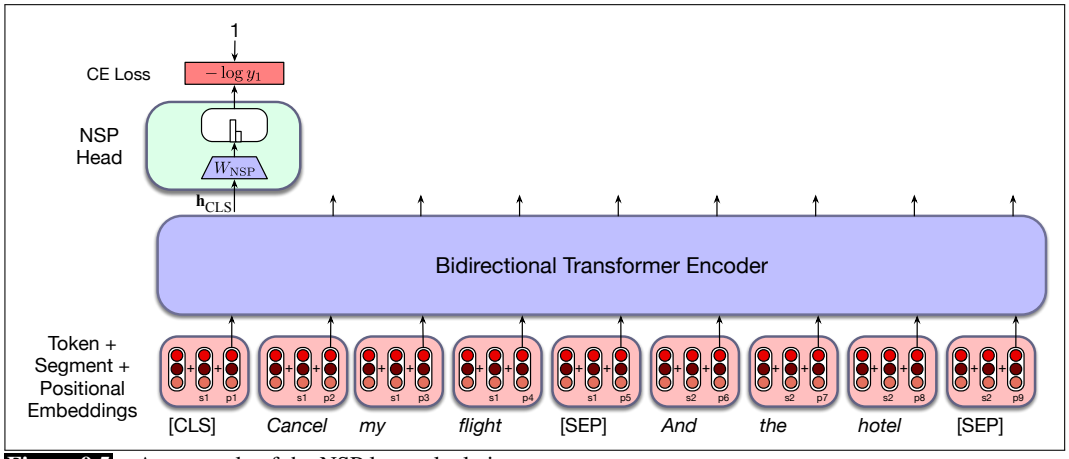

Figure 9.5 An example of the NSP loss calculation.

9.3.3 Training Regimes

BERT and other early transformer-based language models were trained on about 3.3 billion words (a combination of English Wikipedia and a corpus of book texts called BooksCorpus (Zhu et al., 2015) that is no longer used for intellectual property reasons). Modern masked language models are now trained on much larger datasets of web text, filtered a bit, and augmented by higher-quality data like Wikipedia, the same as those we discussed for the causal large language models of Chapter 7. Multilingual models similarly use webtext and multilingual Wikipedia. For example the XLM-R model was trained on about 300 billion tokens in 100 languages, taken from the web via Common Crawl (`https://commoncrawl.org/`).

To train the original BERT models, pairs of text segments were selected from the training corpus according to the next sentence prediction 50/50 scheme. Pairs were sampled so that their combined length was less than the 512 token input. Tokens within these sentence pairs were then masked using the MLM approach with the combined loss from the MLM and NSP objectives used for a final loss. Because this final loss is backpropagated through the entire transformer, the embeddings at each transformer layer will learn representations that are useful for predicting words from their neighbors. Since the `[CLS]` tokens are the direct input to the NSP classifier, their learned representations will tend to contain information about the sequence as a whole. Approximately 40 passes (epochs) over the training data was required for the model to converge.

Some models, like the RoBERTa model, drop the next sentence prediction objective, and therefore change the training regime a bit. Instead of sampling pairs of sentences, the input is simply a series of contiguous sentences, still beginning with the special `[CLS]` token. If the document runs out before 512 tokens are reached, an extra separator token is added, and sentences from the next document are packed in, until we reach a total of 512 tokens. Usually large batch sizes are used, between 8K and 32K tokens.

Multilingual models have an additional decision to make: what data to use to

build the vocabulary? Recall that all language models use subword tokenization (BPE or SentencePiece Unigram LM are the two most common algorithms). What text should be used to learn this multilingual tokenization, given that it's easier to get much more text in some languages than others? One option would be to create this vocabulary-learning dataset by sampling sentences from our training data (perhaps web text from Common Crawl), randomly. In that case we will choose a lot of sentences from languages with lots of web representation like English, and the tokens will be biased toward rare English tokens instead of creating frequent tokens from languages with less data. Instead, it is common to divide the training data into subcorpora of N different languages, compute the number of sentences n_i of each language i, and readjust these probabilities so as to upweight the probability of less-represented languages (Lample and Conneau, 2019). The new probability of selecting a sentence from each of the N languages (whose prior frequency is n_i) is $\{q_i\}_{i=1...N}$, where:

$$q_i = \frac{p_i^\alpha}{\sum_{j=1}^{N} p_j^\alpha} \quad \text{with} \quad p_i = \frac{n_i}{\sum_{k=1}^{N} n_k} \tag{9.5}$$

Recall from Eq. 5.19 in Chapter 5 that an α value between 0 and 1 will give higher weight to lower probability samples. Conneau et al. (2020) show that $\alpha = 0.3$ works well to give rare languages more inclusion in the tokenization, resulting in better multilingual performance overall.

The result of this pretraining process consists of both learned word embeddings, as well as all the parameters of the bidirectional encoder that are used to produce contextual embeddings for novel inputs.

For many purposes, a pretrained multilingual model is more practical than a monolingual model, since it avoids the need to build many (a hundred!) separate monolingual models. And multilingual models can improve performance on low-resourced languages by leveraging linguistic information from a similar language in the training data that happens to have more resources. Nonetheless, when the number of languages grows very large, multilingual models exhibit what has been called the **curse of multilinguality** (Conneau et al., 2020): the performance on each language degrades compared to a model training on fewer languages. Another problem with multilingual models is that they 'have an accent': grammatical structures in higher-resource languages (often English) bleed into lower-resource languages; the vast amount of English language in training makes the model's representations for low-resource languages slightly more English-like (Papadimitriou et al., 2023).

9.4 Fine-Tuning for Classification

The power of pretrained language models lies in their ability to extract generalizations from large amounts of text—generalizations that are useful for myriad downstream applications. There are two ways to make practical use of the generalizations to solve downstream tasks. The most common way is to use natural language to **prompt** the model, putting it in a state where it contextually generates what we want.

fine-tuning

In this section we explore an alternative way to use pretrained language models for downstream applications, a kind of **fine-tuning**. In the kind of fine-tuning used for masked language models, we add application-specific circuitry (often called a

special **head**) on top of pretrained models, taking their output as its input. The fine-tuning process consists of using labeled data about the application to train these additional application-specific parameters. Typically, this training will either freeze or make only minimal adjustments to the pretrained language model parameters.

The following sections introduce fine-tuning methods for some common classes of text-based applications: sequence classification, sentence-pair classification, and sequence labeling.

9.4.1 Sequence Classification

The task of **sequence classification** is to classify an entire sequence of text with a single label. This set of tasks is commonly called **text classification**, like sentiment analysis or spam detection (Appendix B) in which we classify a text into two or three classes (like positive or negative), as well as classification tasks with a large number of categories, like document-level topic classification.

For sequence classification we represent the entire input to be classified by a single vector. We can represent a sequence in various ways. One way is to take the sum or the mean of the last output vector from each token in the sequence. For BERT, we instead add a new unique token to the vocabulary called `[CLS]`, and prepended it to the start of all input sequences, both during pretraining and encoding. The output vector in the final layer of the model for the `[CLS]` input represents the entire input sequence and serves as the input to a **classifier head**, a logistic regression or neural network classifier that makes the relevant decision.

classifier head

As an example, let's return to the problem of sentiment classification. Finetuning a classifier for this application involves learning a set of weights, $\mathbf{W_C}$, to map the output vector for the `[CLS]` token—$\mathbf{h}^{\mathbf{L}}_{\mathbf{CLS}}$—to a set of scores over the possible sentiment classes. Assuming a three-way sentiment classification task (positive, negative, neutral) and dimensionality d as the model dimension, $\mathbf{W_C}$ will be of size $[d \times 3]$. To classify a document, we pass the input text through the pretrained language model to generate $\mathbf{h}^{\mathbf{L}}_{\mathbf{CLS}}$, multiply it by $\mathbf{W_C}$, and pass the resulting vector through a softmax.

$$\mathbf{y} = \text{softmax}(\mathbf{h}^{\mathbf{L}}_{\mathbf{CLS}}\mathbf{W_C}) \tag{9.6}$$

Finetuning the values in $\mathbf{W_C}$ requires supervised training data consisting of input sequences labeled with the appropriate sentiment class. Training proceeds in the usual way; cross-entropy loss between the softmax output and the correct answer is used to drive the learning that produces $\mathbf{W_C}$.

This loss can be used to not only learn the weights of the classifier, but also to update the weights for the pretrained language model itself. In practice, reasonable classification performance is typically achieved with only minimal changes to the language model parameters, often limited to updates over the final few layers of the transformer. Fig. 9.6 illustrates this overall approach to sequence classification.

9.4.2 Sequence-Pair Classification

As mentioned in Section 9.3.2, an important type of problem involves the classification of pairs of input sequences. Practical applications that fall into this class include paraphrase detection (are the two sentences paraphrases of each other?), logical entailment (does sentence A logically entail sentence B?), and discourse coherence (how coherent is sentence B as a follow-on to sentence A?).

Fine-tuning an application for one of these tasks proceeds just as with pretraining using the NSP objective. During fine-tuning, pairs of labeled sentences from a

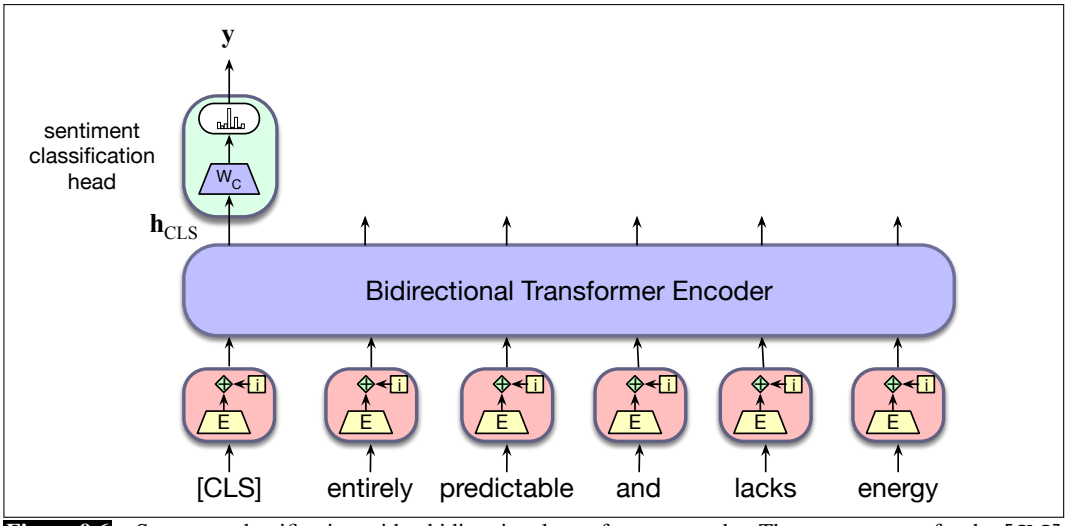

Figure 9.6 Sequence classification with a bidirectional transformer encoder. The output vector for the [CLS] token serves as input to a simple classifier.

supervised fine-tuning set are presented to the model, and run through all the layers of the model to produce the **h** outputs for each input token. As with sequence classification, the output vector associated with the prepended [CLS] token represents the model's view of the input pair. And as with NSP training, the two inputs are separated by the [SEP] token. To perform classification, the [CLS] vector is multiplied by a set of learned classification weights and passed through a softmax to generate label predictions, which are then used to update the weights.

natural language inference

As an example, let's consider an entailment classification task with the Multi-Genre Natural Language Inference (MultiNLI) dataset (Williams et al., 2018). In the task of **natural language inference** or **NLI**, also called **recognizing textual entailment**, a model is presented with a pair of sentences and must classify the relationship between their meanings. For example in the MultiNLI corpus, pairs of sentences are given one of 3 labels: *entails*, *contradicts* and *neutral*. These labels describe a relationship between the meaning of the first sentence (the premise) and the meaning of the second sentence (the hypothesis). Here are representative examples of each class from the corpus:

- `Neutral`
  - a: Jon walked back to the town to the smithy.
  - b: Jon traveled back to his hometown.
- `Contradicts`
  - a: Tourist Information offices can be very helpful.
  - b: Tourist Information offices are never of any help.
- `Entails`
  - a: I'm confused.
  - b: Not all of it is very clear to me.

A relationship of *contradicts* means that the premise contradicts the hypothesis; *entails* means that the premise entails the hypothesis; *neutral* means that neither is necessarily true. The meaning of these labels is looser than strict logical entailment or contradiction indicating that a typical human reading the sentences would most likely interpret the meanings in this way.

To finetune a classifier for the MultiNLI task, we pass the premise/hypothesis pairs through a bidirectional encoder as described above and use the output vector for the [CLS] token as the input to the classification head. As with ordinary sequence classification, this head provides the input to a three-way classifier that can be trained on the MultiNLI training corpus.

9.5 Fine-Tuning for Sequence Labeling: Named Entity Recognition

In sequence labeling, the network's task is to assign a label chosen from a small fixed set of labels to each token in the sequence. One of the most common sequence labeling task is **named entity recognition**.

9.5.1 Named Entities

named entity
named entity recognition
NER

A **named entity** is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization. The task of **named entity recognition** (**NER**) is to find spans of text that constitute proper names and tag the type of the entity. Four entity tags are most common: **PER** (person), **LOC** (location), **ORG** (organization), or **GPE** (geo-political entity). However, the term **named entity** is commonly extended to include things that aren't entities per se, including temporal expressions like dates and times, and even numerical expressions like prices. Here's an example of the output of an NER tagger:

> Citing high fuel prices, [ORG **United Airlines**] said [TIME **Friday**] it has increased fares by [MONEY **$6**] per round trip on flights to some cities also served by lower-cost carriers. [ORG **American Airlines**], a unit of [ORG **AMR Corp.**], immediately matched the move, spokesman [PER **Tim Wagner**] said. [ORG **United**], a unit of [ORG **UAL Corp.**], said the increase took effect [TIME **Thursday**] and applies to most routes where it competes against discount carriers, such as [LOC **Chicago**] to [LOC **Dallas**] and [LOC **Denver**] to [LOC **San Francisco**].

The text contains 13 mentions of named entities including 5 organizations, 4 locations, 2 times, 1 person, and 1 mention of money. Figure 9.7 shows typical generic named entity types. Many applications will also need to use specific entity types like proteins, genes, commercial products, or works of art.

Type	Tag	Sample Categories	Example sentences
People	PER	people, characters	**Turing** is a giant of computer science.
Organization	ORG	companies, sports teams	The **IPCC** warned about the cyclone.
Location	LOC	regions, mountains, seas	**Mt. Sanitas** is in **Sunshine Canyon**.
Geo-Political Entity	GPE	countries, states	**Palo Alto** is raising the fees for parking.

Figure 9.7 A list of generic named entity types with the kinds of entities they refer to.

Named entity recognition is a useful step in various natural language processing tasks, including linking text to information in structured knowledge sources like Wikipedia, measuring sentiment or attitudes toward a particular entity in text, or even as part of anonymizing text for privacy. The NER task is difficult because of the ambiguity of segmenting NER spans, figuring out which tokens are entities

and which aren't, since most words in a text will not be named entities. Another difficulty is caused by type ambiguity. The mention `Washington` can refer to a person, a sports team, a city, or the US government, as we see in Fig. 9.8.

[PER Washington] was born into slavery on the farm of James Burroughs.
[ORG Washington] went up 2 games to 1 in the four-game series.
Blair arrived in [LOC Washington] for what may well be his last state visit.
In June, [GPE Washington] passed a primary seatbelt law.

Figure 9.8 Examples of type ambiguities in the use of the name *Washington*.

9.5.2 BIO Tagging

BIO tagging

One standard approach to sequence labeling for a span-recognition problem like NER is **BIO tagging** (Ramshaw and Marcus, 1995). This is a method that allows us to treat NER like a word-by-word sequence labeling task, via tags that capture both the boundary and the named entity type. Consider the following sentence:

[PER **Jane Villanueva**] of [ORG **United**] , a unit of [ORG **United Airlines Holding**] , said the fare applies to the [LOC **Chicago**] route.

BIO

Figure 9.9 shows the same excerpt represented with **BIO** tagging, as well as variants called **IO** tagging and **BIOES** tagging. In BIO tagging we label any token that *begins* a span of interest with the label B, tokens that occur *inside* a span are tagged with an I, and any tokens *outside* of any span of interest are labeled O. While there is only one O tag, we'll have distinct B and I tags for each named entity class. The number of tags is thus $2n+1$, where n is the number of entity types. BIO tagging can represent exactly the same information as the bracketed notation, but has the advantage that we can represent the task in the same simple sequence modeling way as part-of-speech tagging: assigning a single label y_i to each input word x_i:

Words	IO Label	BIO Label	BIOES Label
Jane	I-PER	B-PER	B-PER
Villanueva	I-PER	I-PER	E-PER
of	O	O	O
United	I-ORG	B-ORG	B-ORG
Airlines	I-ORG	I-ORG	I-ORG
Holding	I-ORG	I-ORG	E-ORG
discussed	O	O	O
the	O	O	O
Chicago	I-LOC	B-LOC	S-LOC
route	O	O	O
.	O	O	O

Figure 9.9 NER as a sequence model, showing IO, BIO, and BIOES taggings.

We've also shown two variant tagging schemes: IO tagging, which loses some information by eliminating the B tag, and BIOES tagging, which adds an end tag E for the end of a span, and a span tag S for a span consisting of only one word.

9.5.3 Sequence Labeling

In sequence labeling, we pass the final output vector corresponding to each input token to a classifier that produces a softmax distribution over the possible set of

tags. For a single feedforward layer classifier, the set of weights to be learned is $\mathbf{W_K}$ of size $[d \times k]$, where k is the number of possible tags for the task. A greedy approach, where the argmax tag for each token is taken as a likely answer, can be used to generate the final output tag sequence. Fig. 9.10 illustrates an example of this approach, where $\mathbf{y_i}$ is a vector of probabilities over tags, and k indexes the tags.

$$\mathbf{y_i} = \text{softmax}(\mathbf{h_i^L W_K}) \tag{9.7}$$

$$\mathbf{t_i} = \text{argmax}_k(\mathbf{y}_i) \tag{9.8}$$

Alternatively, the distribution over labels provided by the softmax for each input token can be passed to a conditional random field (CRF) layer which can take global tag-level transitions into account (see Chapter 18 on CRFs).

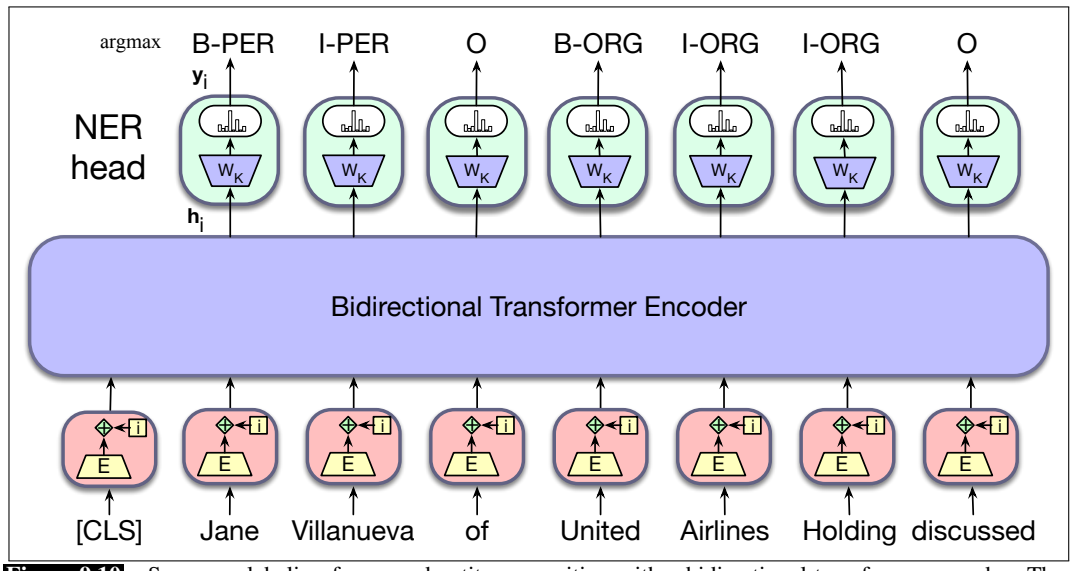

Figure 9.10 Sequence labeling for named entity recognition with a bidirectional transformer encoder. The output vector for each input token is passed to a simple k-way classifier.

Tokenization and NER

Note that supervised training data for NER is typically in the form of BIO tags associated with text segmented at the word level. For example the following sentence containing two named entities:

[$_{\text{LOC}}$ **Mt. Sanitas**] is in [$_{\text{LOC}}$ **Sunshine Canyon**] .

would have the following set of per-word BIO tags.

(9.9) *Mt. Sanitas is in Sunshine Canyon .*
B-LOC I-LOC O O B-LOC I-LOC O

Unfortunately, the sequence of WordPiece tokens for this sentence doesn't align directly with BIO tags in the annotation:

`'Mt', '.', 'San', '##itas', 'is', 'in', 'Sunshine', 'Canyon' '.'`

To deal with this misalignment, we need a way to assign BIO tags to subword tokens during training and a corresponding way to recover word-level tags from

subwords during decoding. For training, we can just assign the gold-standard tag associated with each word to all of the subword tokens derived from it.

For decoding, the simplest approach is to use the argmax BIO tag associated with the first subword token of a word. Thus, in our example, the BIO tag assigned to "Mt" would be assigned to "Mt." and the tag assigned to "San" would be assigned to "Sanitas", effectively ignoring the information in the tags assigned to "." and "##itas". More complex approaches combine the distribution of tag probabilities across the subwords in an attempt to find an optimal word-level tag.

9.5.4 Evaluating Named Entity Recognition

Named entity recognizers are evaluated by **recall**, **precision**, and $\mathbf{F_1}$ **measure**. Recall that recall is the ratio of the number of correctly labeled responses to the total that should have been labeled; precision is the ratio of the number of correctly labeled responses to the total labeled; and F_1 measure is the harmonic mean of the two.

To know if the difference between the F_1 scores of two NER systems is a significant difference, we use the paired bootstrap test, or the similar randomization test (Section 4.11).

For named entity tagging, the *entity* rather than the word is the unit of response. Thus in the example in Fig. 9.9, the two entities *Jane Villanueva* and *United Airlines Holding* and the non-entity *discussed* would each count as a single response.

The fact that named entity tagging has a segmentation component which is not present in tasks like text categorization or part-of-speech tagging causes some problems with evaluation. For example, a system that labeled *Jane* but not *Jane Villanueva* as a person would cause two errors, a false positive for O and a false negative for I-PER. In addition, using entities as the unit of response but words as the unit of training means that there is a mismatch between the training and test conditions.

9.6 Summary

This chapter has introduced the **bidirectional encoder** and the **masked language model**. Here's a summary of the main points that we covered:

- There are three major architectures for language models: the **encoder** (this chapter), the **decoder**, and the **encoder-decoder**. We saw decoder models in Chapter 1 and Chapter 7, and we'll see encoder-decoders in Chapter 13.
- Bidirectional encoders can be used to generate contextualized representations of input embeddings using the entire input context.
- Pretrained language models based on bidirectional encoders can be learned using a masked language model objective where a model is trained to guess the missing information from an input.
- The vector output of each transformer block or component in a particular token column is a **contextual embedding** that represents some aspect of the meaning of a token in context.
- A **word sense** is a discrete representation of one aspect of the meaning of a word. Contextual embeddings offer a continuous high-dimensional model of meaning that is richer than fully discrete senses.

- The cosine between contextual embeddings can be used as one way to model the similarity between two words in context, although some transformations to the embeddings are required first.
- Pretrained language models can be finetuned for specific applications by adding lightweight classifier layers on top of the outputs of the pretrained model.
- These applications can include **sequence classification** tasks like sentiment analysis, **sequence-pair classification** tasks like natural language inference, or **sequence labeling** tasks like **named entity recognition**.

Historical Notes

History TBD.