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** model (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 **finetuning** in the prior chapter. Here we describe a new kind of finetuning, 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.

finetuning

transfer learning

The second idea that we introduce in this chapter is the idea of **contextual embeddings**: representations for words in context. The methods of Chapter 5 like word2vec or GloVe learned a single vector embedding for each unique word w in the vocabulary. By contrast, with contextual embeddings, such as those learned by masked language models like BERT, each word w will be represented by a different vector each time it appears in a different context. While the causal language models of Chapter 8 also use contextual embeddings, the embeddings created by masked language models seem to function particularly well as representations.

9.1 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) or **SpanBERT** (Joshi et al., 2020). In Chapter 7 we introduced the idea of left-to-right language models that can be applied to autoregressive contextual generation problems like question answering or summarization, and in Chapter 8 we saw how to implement language models with causal (left-to-right) transformers. But this left-to-right nature of these models is also 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 espe-

cially 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 8 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 8 are sometimes called **decoder-only**, because they correspond to the decoder part of the encoder-decoder model we will introduce in Chapter 12. 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. That's an important point: masked language models are not used for generation. They are generally instead used for interpretative tasks.

9.1.1 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. 9.1a, reproduced here from Chapter 8, shows the information flow in the left-to-right approach of Chapter 8. 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. 9.1b.

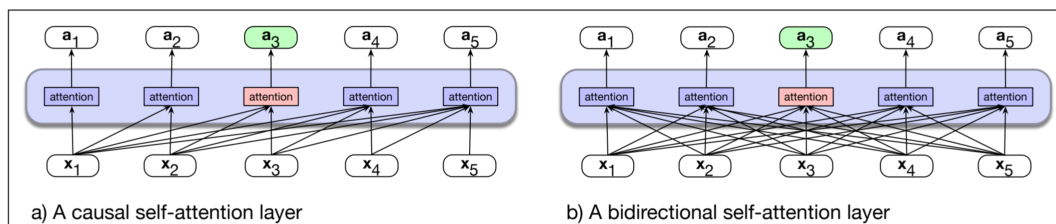

Figure 9.1 (a) The causal transformer from Chapter 8, highlighting the attention computation at token 3. The attention value at each token is computed using only information seen earlier in the context. (b) Information flow in a bidirectional attention model. In processing each token, the model attends 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. 8.34. Recall from Chapter 8 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. 8.34 for a single attention head):

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

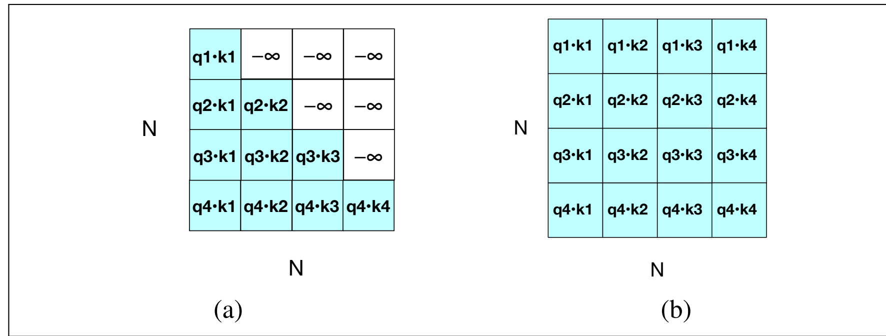

Figure 9.2 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing the $q_i \cdot k_j$ values. (a) shows the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero), while (b) shows the unmasked version.

Fig. 9.2 shows the masked version of $\mathbf{QK}^\intercal$ and the unmasked version. For bidirectional attention, we use the unmasked version of Fig. 9.2b. Thus the attention computation for bidirectional attention is exactly the same as Eq. 9.1 but with the mask removed:

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

Otherwise, the attention computation is identical to what we saw in Chapter 8, as is the transformer block architecture (the feedforward layer, layer norm, and so on). As in Chapter 8, 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, 2018b).

- 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.

9.2 Training Bidirectional Encoders

We trained causal transformer language models in Chapter 8 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

9.2.1 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 finetuning 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.

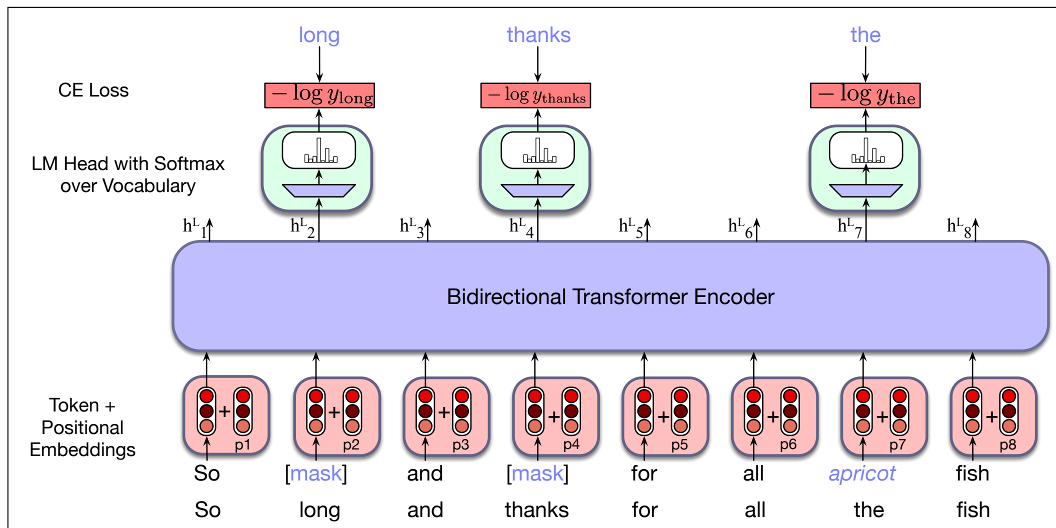

Figure 9.3 Masked language model training. In this example, three of the input tokens are selected, two of which are masked and the third is replaced with an unrelated word. The probabilities assigned by the model to these three items are used as the training loss. The other 5 tokens don't play a role in training loss.

Fig. 9.3 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 8.5 in Chapter 8 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. 9.3. 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. 9.3, 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]

9.2.2 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).

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.

Next Sentence Prediction

To facilitate NSP training, BERT introduces two special tokens to the input representation (tokens that will prove useful for finetuning 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

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

$\mathbf{X}$ 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}:

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

Cross entropy is used to compute the NSP loss for each sentence pair presented to the model. Fig. 9.4 illustrates the overall NSP training setup. In BERT, the NSP loss was used in conjunction with the MLM training objective to form final loss.

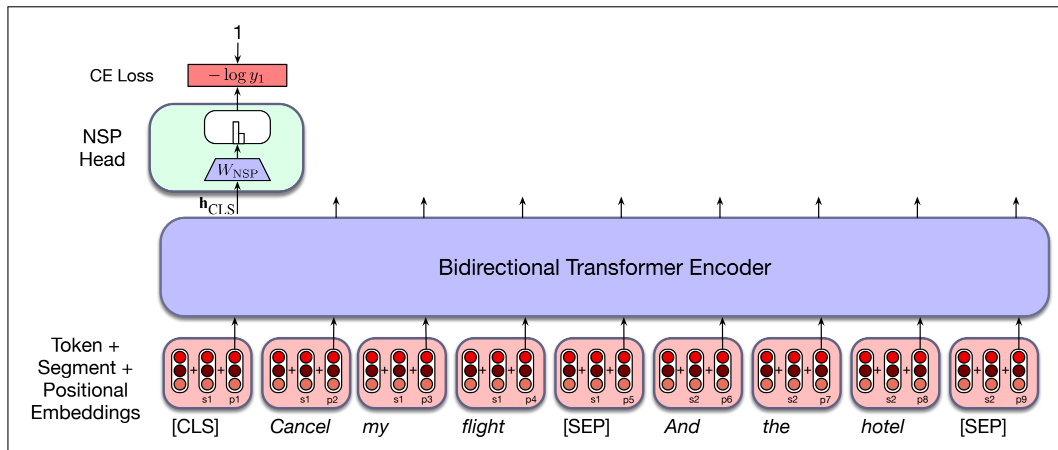

Figure 9.4 An example of the NSP loss calculation.

9.2.3 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 8. 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 sentence, 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).

9.3 Contextual Embeddings

contextual embeddings

Given a pretrained language model and a novel input sentence, we can think of the sequence of model outputs as constituting **contextual embeddings** for each token in the input. These contextual embeddings are vectors representing some aspect of the meaning of a token in context, and can be used for any task requiring the meaning of tokens or words. More formally, given a sequence of input tokens $x_1, \ldots, x_n$, we can use the output vector $\mathbf{h^L}_i$ from the final layer L of the model as a representation of the meaning of token x_i in the context of sentence $x_1, \ldots, x_n$. Or instead of just using the vector $\mathbf{h^L}_i$ from the final layer of the model, it's common to compute a representation for x_i by averaging the output tokens $\mathbf{h}_i$ from each of the last four layers of the model, i.e., $\mathbf{h^L}_i$, $\mathbf{h^{L-1}}_i$, $\mathbf{h^{L-2}}_i$, and $\mathbf{h^{L-3}}_i$.

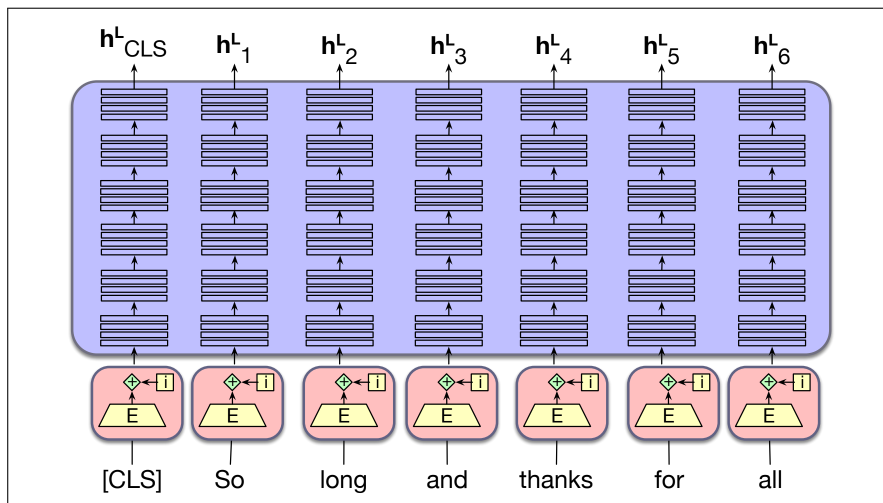

Figure 9.5 The output of a BERT-style model is a contextual embedding vector $\mathbf{h}_i^L$ for each input token x_i.

Just as we used static embeddings like word2vec in Chapter 5 to represent the meaning of words, we can use contextual embeddings as representations of word meanings in context for any task that might require a model of word meaning. Where static embeddings represent the meaning of word *types* (vocabulary entries), contextual embeddings represent the meaning of word *instances*: instances of a particular word type in a particular context. Thus where word2vec had a single vector for each word type, contextual embeddings provide a single vector for each instance of that word type in its sentential context. Contextual embeddings can thus be used for tasks like measuring the semantic similarity of two words in context, and are useful in linguistic tasks that require models of word meaning.

9.3.1 Contextual Embeddings and Word Sense

ambiguous

Words are **ambiguous**: the same word can be used to mean different things. In Chapter 5 we saw that the word "mouse" can mean (1) a small rodent, or (2) a hand-operated device to control a cursor. The word "bank" can mean: (1) a financial institution or (2) a sloping mound. We say that the words 'mouse' or 'bank' are

polysemous (from Greek 'many senses', *poly-* 'many' + *sema*, 'sign, mark').[2]

word sense

A **sense** (or **word sense**) is a discrete representation of one aspect of the meaning of a word. We can represent each sense with a superscript: **bank**^{1} and **bank**^{2}, **mouse**^{1} and **mouse**^{2}. These senses can be found listed in online thesauruses (or thesauri) like **WordNet** (Fellbaum, 1998), which has datasets in many languages listing the senses of many words. In context, it's easy to see the different meanings:

WordNet

mouse^1 : .... a *mouse* controlling a computer system in 1968.
mouse^2 : .... a quiet animal like a *mouse*
bank^1 : ...a *bank* can hold the investments in a custodial account ...
bank^2 : ...as agriculture burgeons on the east *bank*, the river ...

This fact that context disambiguates the senses of *mouse* and *bank* above can also be visualized geometrically. Fig. 9.6 shows a two-dimensional projection of many instances of the BERT embeddings of the word *die* in English and German. Each point in the graph represents the use of *die* in one input sentence. We can clearly see at least two different English senses of *die* (the singular of *dice* and the verb *to die*, as well as the German article, in the BERT embedding space.

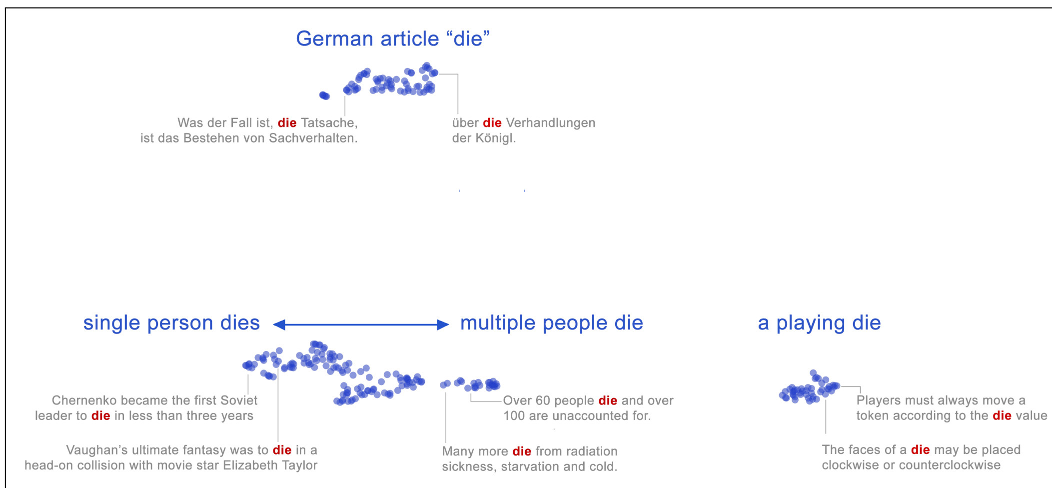

Figure 9.6 Each blue dot shows a BERT contextual embedding for the word *die* from different sentences in English and German, projected into two dimensions with the UMAP algorithm. The German and English meanings and the different English senses fall into different clusters. Some sample points are shown with the contextual sentence they came from. Figure from Coenen et al. (2019).

Thus while thesauruses like WordNet give discrete lists of senses, embeddings (whether static or contextual) offer a continuous high-dimensional model of meaning that, although it can be clustered, doesn't divide up into fully discrete senses.

Word Sense Disambiguation

word sense disambiguation
WSD

The task of selecting the correct sense for a word is called **word sense disambiguation**, or **WSD**. WSD algorithms take as input a word in context and a fixed inventory of potential word senses (like the ones in WordNet) and outputs the correct word sense in context. Fig. 9.7 sketches out the task.

[2] The word **polysemy** itself is ambiguous; you may see it used in a different way, to refer only to cases where a word's senses are related in some structured way, reserving the word **homonymy** to mean sense ambiguities with no relation between the senses (Haber and Poesio, 2020). Here we will use 'polysemy' to mean any kind of sense ambiguity, and 'structured polysemy' for polysemy with sense relations.

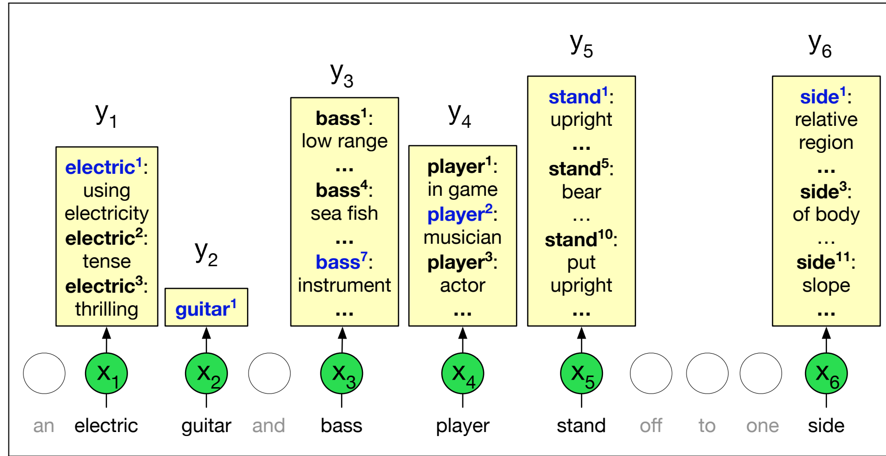

Figure 9.7 The all-words WSD task, mapping from input words (x) to WordNet senses (y). Figure inspired by Chaplot and Salakhutdinov (2018).

WSD can be a useful analytic tool for text analysis in the humanities and social sciences, and word senses can play a role in model interpretability for word representations. Word senses also have interesting distributional properties. For example a word often is used in roughly the same sense through a discourse, an observation called the **one sense per discourse** rule (Gale et al., 1992a).

one sense per discourse

The best performing WSD algorithm is a simple 1-nearest-neighbor algorithm using contextual word embeddings, due to Melamud et al. (2016) and Peters et al. (2018). At training time we pass each sentence in some sense-labeled dataset (like the SemCore or SenseEval datasets in various languages) through any contextual embedding (e.g., BERT) resulting in a contextual embedding for each labeled token. (There are various ways to compute this contextual embedding v_i for a token i; for BERT it is common to pool multiple layers by summing the vector representations of i from the last four BERT layers). Then for each sense s of any word in the corpus, for each of the n tokens of that sense, we average their n contextual representations v_i to produce a contextual **sense embedding** $\mathbf{v}_s$ for s:

$$\mathbf{v}_s = \frac{1}{n}\sum_i \mathbf{v}_i \qquad \forall \mathbf{v}_i \in \text{tokens}(s) \tag{9.6}$$

At test time, given a token of a target word t in context, we compute its contextual embedding $\mathbf{t}$ and choose its nearest neighbor sense from the training set, i.e., the sense whose sense embedding has the highest cosine with $\mathbf{t}$:

$$\text{sense}(t) = \underset{s \in \text{senses}(t)}{\text{argmax}}\ \text{cosine}(\mathbf{t}, \mathbf{v}_s) \tag{9.7}$$

Fig. 9.8 illustrates the model.

9.3.2 Contextual Embeddings and Word Similarity

In Chapter 5 we introduced the idea that we could measure the similarity of two words by considering how close they are geometrically, by using the cosine as a similarity function. The idea of meaning similarity is also clear geometrically in the meaning clusters in Fig. 9.6; the representation of a word which has a particular sense in a context is closer to other instances of the same sense of the word. Thus we

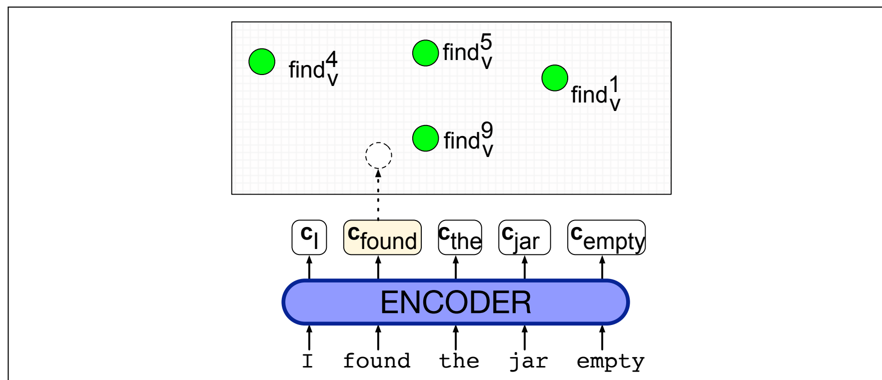

Figure 9.8 The nearest-neighbor algorithm for WSD. In green are the contextual embeddings precomputed for each sense of each word; here we just show a few of the senses for *find*. A contextual embedding is computed for the target word *found*, and then the nearest neighbor sense (in this case **find**$_v^9$) is chosen. Figure inspired by Loureiro and Jorge (2019).

often measure the similarity between two instances of two words in context (or two instances of the same word in two different contexts) by using the cosine between their contextual embeddings.

Usually some transformations to the embeddings are required before computing cosine. This is because contextual embeddings (whether from masked language models or from autoregressive ones) have the property that the vectors for all words are extremely similar. If we look at the embeddings from the final layer of BERT or other models, embeddings for instances of any two randomly chosen words will have extremely high cosines that can be quite close to 1, meaning all word vectors tend to point in the same direction. The property of vectors in a system all tending to point in the same direction is known as **anisotropy**. Ethayarajh (2019) defines the **anisotropy** of a model as the expected cosine similarity of any pair of words in a corpus. The word 'isotropy' means uniformity in all directions, so in an isotropic model, the collection of vectors should point in all directions and the expected cosine between a pair of random embeddings would be zero. Timkey and van Schijndel (2021) show that one cause of anisotropy is that cosine measures are dominated by a small number of dimensions of the contextual embedding whose values are very different than the others: these **rogue dimensions** have very large magnitudes and very high variance.

anisotropy

Timkey and van Schijndel (2021) shows that we can make the embeddings more isotropic by standardizing (z-scoring) the vectors, i.e., subtracting the mean and dividing by the variance. Given a set C of all the embeddings in some corpus, each with dimensionality d (i.e., $x \in \mathbb{R}^d$), the mean vector $\mu \in \mathbb{R}^d$ is:

$$\mu = \frac{1}{|C|} \sum_{\mathbf{x} \in C} \mathbf{x} \tag{9.8}$$

The standard deviation in each dimension $\sigma \in \mathbb{R}^d$ is:

$$\sigma = \sqrt{\frac{1}{|C|} \sum_{\mathbf{x} \in C} (\mathbf{x} - \mu)^2} \tag{9.9}$$

Then each word vector $\mathbf{x}$ is replaced by a standardized version $\mathbf{z}$:

$$\mathbf{z} = \frac{\mathbf{x} - \mu}{\sigma} \tag{9.10}$$

One problem with cosine that is not solved by standardization is that cosine tends to underestimate human judgments on similarity of word meaning for very frequent words (Zhou et al., 2022).

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.

finetuning

In this section we explore an alternative way to use pretrained language models for downstream applications: a version of the **finetuning** paradigm from Chapter 7. In the kind of finetuning 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 finetuning 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 finetuning methods for the most common kinds of 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 K) 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—$\mathbf{h^L_{CLS}}$—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 $\mathbf{h^L_{CLS}}$, multiply it by $\mathbf{W_C}$, and pass the resulting vector through a softmax.

$$\mathbf{y} = \text{softmax}(\mathbf{h^L_{CLS}} \mathbf{W_C}) \tag{9.11}$$

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. 9.9 illustrates this overall approach to sequence classification.

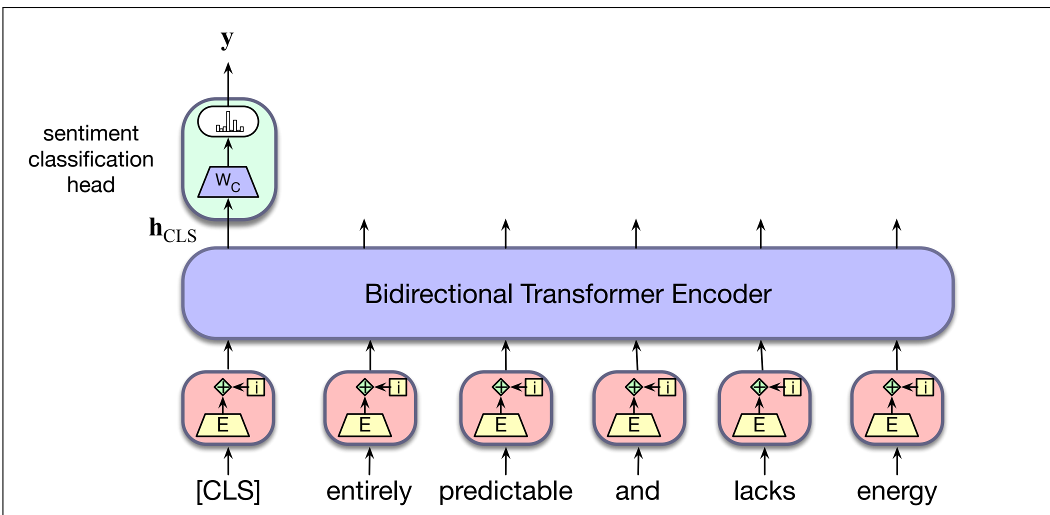

Figure 9.9 Sequence classification with a bidirectional transformer encoder. The output vector for the [CLS] token serves as input to a simple classifier.

9.4.2 Sequence-Pair Classification

As mentioned in Section 9.2.2, 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 finetuning, pairs of labeled sentences from a supervised finetuning set are presented to the model, and run through all the layers of the model to produce the $\mathbf{h}$ 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.

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

natural language inference

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 9.10 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 9.10 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. 9.11.

[$_{PER}$ 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 9.11 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 9.12 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:

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.

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

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. 9.13 illustrates an example of this approach, where $\mathbf{y_i}$ is a vector of probabilities over tags, and k indexes the tags.

$$\mathbf{y_i} = \text{softmax}(\mathbf{h_i^L W_K}) \tag{9.12}$$

$$\mathbf{t_i} = \text{argmax}_k(\mathbf{y}_i) \tag{9.13}$$

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 17 on CRFs).

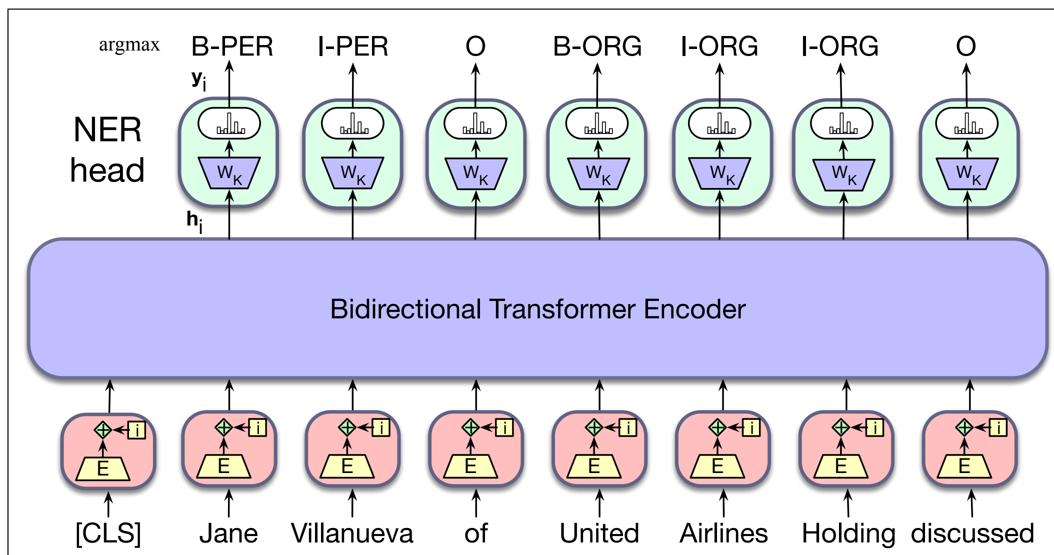

Figure 9.13 Sequence labeling for named entity recognition with a bidirectional transformer encoder. The output vector for each input token is passed to a simple k-way classifier.

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:

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

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

(9.14) *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 **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. 9.12, 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:

- 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.