Tong Xiao

Jingbo Zhu

Natural Language Processing

Neural Networks and Large Language Models

NATURAL LANGUAGE PROCESSING LAB
NORTHEASTERN UNIVERSITY

&

NIUTRANS RESEARCH

NATURAL LANGUAGE PROCESSING LAB, NORTHEASTERN UNIVERSITY
&
NIUTRANS RESEARCH

June 6, 2025

Contents

Chapter 7

Pre-training

The development of neural sequence models, such as Transformers, along with the improvements in large-scale self-supervised learning, has opened the door to universal language understanding and generation. This achievement is largely motivated by pre-training: we separate common components from many neural network-based systems, and then train them on huge amounts of unlabeled data using self-supervision. These pre-trained models serve as foundation models that can be easily adapted to different tasks via fine-tuning or prompting. As a result, the paradigm of NLP has been enormously changed. In many cases, large-scale supervised learning for specific tasks is no longer required, and instead, we only need to adapt pre-trained foundation models.

While pre-training has gained popularity in recent NLP research, this concept dates back decades to the early days of deep learning. For example, early attempts to pre-train deep learning systems include unsupervised learning for RNNs, deep feedforward networks, autoencoders, and others [Schmidhuber, 2015]. In the modern era of deep learning, we experienced a resurgence of pre-training, caused in part by the large-scale unsupervised learning of various word embedding models [Mikolov et al., 2013; Pennington et al., 2014]. During the same period, pre-training also attracted significant interest in computer vision, where the backbone models were trained on relatively large labeled datasets such as ImageNet, and then applied to different downstream tasks [He et al., 2019; Zoph et al., 2020]. Large-scale research on pre-training in NLP began with the development of language models using self-supervised learning. This family of models covers several well-known examples like **BERT** [Devlin et al., 2019] and **GPT** [Brown et al., 2020], all with a similar idea that general language understanding and generation can be achieved by training the models to predict masked words in a huge amount of text. Despite the simple nature of this approach, the resulting models show remarkable capability in modeling linguistic structure, though they are not explicitly trained to achieve this. The generality of the pre-training tasks leads to systems that exhibit strong performance in a large variety of NLP problems, even outperforming previously well-developed supervised systems. More recently, pre-trained large language models have achieved greater success, showing the exciting prospects for more general artificial intelligence [Bubeck et al., 2023].

This chapter discusses the concept of pre-training in the context of NLP. It begins with a general introduction to pre-training methods and their applications. BERT is then used as an example to illustrate how a sequence model is trained via a self-supervised task, called **masked language modeling**. This is followed by a discussion of methods for adapting pre-trained sequence models for various NLP tasks. Note that in this chapter, we will focus primarily on the pre-training paradigm in NLP, and therefore, we do not intend to cover details about generative large language models. A detailed discussion of these models will be left to subsequent chapters.

7.1 Pre-training NLP Models

The discussion of pre-training issues in NLP typically involves two types of problems: sequence modeling (or sequence encoding) and sequence generation. While these problems have different forms, for simplicity, we describe them using a single model defined as follows:

$$\begin{aligned} \mathbf{o} &= g(x_0, x_1, ..., x_m; \theta) \\ &= g_\theta(x_0, x_1, ..., x_m) \end{aligned} \tag{7.1}$$

where $\{x_0, x_1, ..., x_m\}$ denotes a sequence of input tokens[1], x_0 denotes a special symbol ($\langle s \rangle$ or [CLS]) attached to the beginning of a sequence, $g(\cdot;\theta)$ (also written as $g_\theta(\cdot)$) denotes a neural network with parameters θ, and $\mathbf{o}$ denotes the output of the neural network. Different problems can vary based on the form of the output $\mathbf{o}$. For example, in token prediction problems (as in language modeling), $\mathbf{o}$ is a distribution over a vocabulary; in sequence encoding problems, $\mathbf{o}$ is a representation of the input sequence, often expressed as a real-valued vector sequence.

There are two fundamental issues here.

- Optimizing θ on a pre-training task. Unlike standard learning problems in NLP, pre-training does not assume specific downstream tasks to which the model will be applied. Instead, the goal is to train a model that can generalize across various tasks.
- Applying the pre-trained model $g_{\hat{\theta}}(\cdot)$ to downstream tasks. To adapt the model to these tasks, we need to adjust the parameters $\hat{\theta}$ slightly using labeled data or prompt the model with task descriptions.

In this section, we discuss the basic ideas in addressing these issues.

7.1.1 Unsupervised, Supervised and Self-supervised Pre-training

In deep learning, pre-training refers to the process of optimizing a neural network before it is further trained/tuned and applied to the tasks of interest. This approach is based on an assumption that a model pre-trained on one task can be adapted to perform another task. As a result, we do not need to train a deep, complex neural network from scratch on tasks with

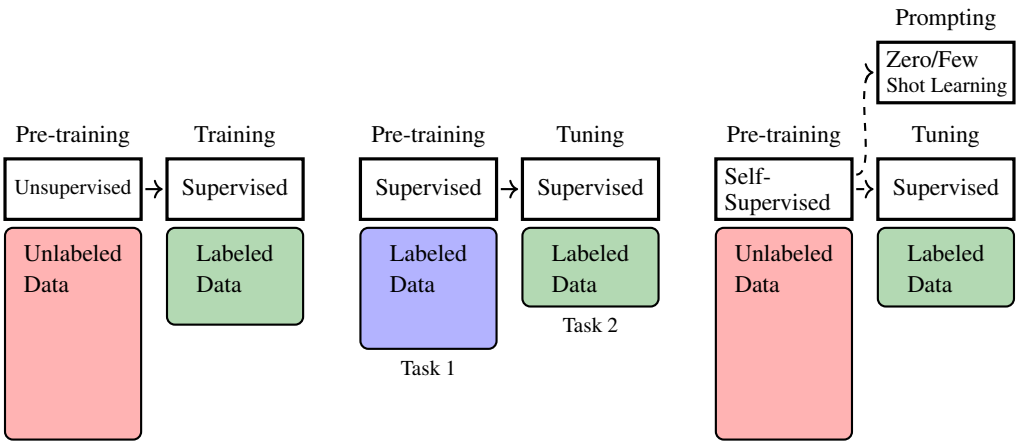
[1] Here we assume that tokens are basic units of text that are separated through tokenization. Sometimes, we will use the terms *token* and *word* interchangeably, though they have closely related but slightly different meanings in NLP.

limited labeled data. Instead, we can make use of tasks where supervision signals are easier to obtain. This reduces the reliance on task-specific labeled data, enabling the development of more general models that are not confined to particular problems.

During the resurgence of neural networks through deep learning, many early attempts to achieve pre-training were focused on **unsupervised learning**. In these methods, the parameters of a neural network are optimized using a criterion that is not directly related to specific tasks. For example, we can minimize the reconstruction cross-entropy of the input vector for each layer [Bengio et al., 2006]. Unsupervised pre-training is commonly employed as a preliminary step before supervised learning, offering several advantages, such as aiding in the discovery of better local minima and adding a regularization effect to the training process [Erhan et al., 2010]. These benefits make the subsequent supervised learning phase easier and more stable.

A second approach to pre-training is to pre-train a neural network on **supervised learning** tasks. For example, consider a sequence model designed to encode input sequences into some representations. In pre-training, this model is combined with a classification layer to form a classification system. This system is then trained on a pre-training task, such as classifying sentences based on sentiment (e.g., determining if a sentence conveys a positive or negative sentiment). Then, we adapt the sequence model to a downstream task. We build a new classification system based on this pre-trained sequence model and a new classification layer (e.g., determining if a sequence is subjective or objective). Typically, we need to fine-tune the parameters of the new model using task-specific labeled data, ensuring the model is optimally adjusted to perform well on this new type of data. The fine-tuned model is then employed to classify new sequences for this task. An advantage of supervised pre-training is that the training process, either in the pre-training or fine-tuning phase, is straightforward, as it follows the well-studied general paradigm of supervised learning in machine learning. However, as the complexity of the neural network increases, the demand for more labeled data also grows. This, in turn, makes the pre-training task more difficult, especially when large-scale labeled data is not available.

A third approach to pre-training is **self-supervised learning**. In this approach, a neural network is trained using the supervision signals generated by itself, rather than those provided by humans. This is generally done by constructing its own training tasks directly from unlabeled data, such as having the system create pseudo labels. While self-supervised learning has recently emerged as a very popular method in NLP, it is not a new concept. In machine learning, a related concept is **self-training** where a model is iteratively improved by learning from the pseudo labels assigned to a dataset. To do this, we need some seed data to build an initial model. This model then generates pseudo labels for unlabeled data, and these pseudo labels are subsequently used to iteratively refine and bootstrap the model itself. Such a method has been successfully used in several NLP areas, such as word sense disambiguation [Yarowsky, 1995] and document classification [Blum and Mitchell, 1998]. Unlike the standard self-training method, self-supervised pre-training in NLP does not rely on an initial model for annotating the data. Instead, all the supervision signals are created from the text, and the entire model is trained from scratch. A well-known example of this is training sequence models by successively predicting a masked word given its preceding or surrounding words in a text. This



(a) Unsupervised Pre-training (b) Supervised Pre-training (c) Self-supervised Pre-training

Figure 7.1: Illustration of unsupervised, supervised, and self-supervised pre-training. In unsupervised pre-training, the pre-training is performed on large-scale unlabeled data. It can be viewed as a preliminary step to have a good starting point for the subsequent optimization process, though considerable effort is still required to further train the model with labeled data after pre-training. In supervised pre-training, the underlying assumption is that different (supervised) learning tasks are related. So we can first train the model on one task, and transfer the resulting model to another task with some training or tuning effort. In self-supervised pre-training, a model is pre-trained on large-scale unlabeled data via self-supervision. The model can be well trained in this way, and we can efficiently adapt it to new tasks through fine-tuning or prompting.

enables large-scale self-supervised learning for deep neural networks, leading to the success of pre-training in many understanding, writing, and reasoning tasks.

Figure 7.1 shows a comparison of the above three pre-training approaches. Self-supervised pre-training is so successful that most current state-of-the-art NLP models are based on this paradigm. Therefore, in this chapter and throughout this book, we will focus on self-supervised pre-training. We will show how sequence models are pre-trained via self-supervision and how the pre-trained models are applied.

7.1.2 Adapting Pre-trained Models

As mentioned above, two major types of models are widely used in NLP pre-training.

- **Sequence Encoding Models**. Given a sequence of words or tokens, a sequence encoding model represents this sequence as either a real-valued vector or a sequence of vectors, and obtains a representation of the sequence. This representation is typically used as input to another model, such as a sentence classification system.
- **Sequence Generation Models**. In NLP, sequence generation generally refers to the problem of generating a sequence of tokens based on a given context. The term *context* has different meanings across applications. For example, it refers to the preceding

tokens in language modeling, and refers to the source-language sequence in machine translation[2].

We need different techniques for applying these models to downstream tasks after pre-training. Here we are interested in the following two methods.

1. Fine-tuning of Pre-trained Models

For sequence encoding pre-training, a common method of adapting pre-trained models is fine-tuning. Let $\mathrm{Encode}_\theta(\cdot)$ denote an encoder with parameters θ, for example, $\mathrm{Encode}_\theta(\cdot)$ can be a standard Transformer encoder. Provided we have pre-trained this model in some way and obtained the optimal parameters $\hat{\theta}$, we can employ it to model any sequence and generate the corresponding representation, like this

$$\mathbf{H} \quad = \quad \mathrm{Encode}_{\hat{\theta}}(\mathbf{x}) \tag{7.2}$$

where $\mathbf{x}$ is the input sequence $\{x_0, x_1, ..., x_m\}$, and $\mathbf{H}$ is the output representation which is a sequence of real-valued vectors $\{\mathbf{h}_0, \mathbf{h}_1, ..., \mathbf{h}_m\}$. Because the encoder does not work as a standalone NLP system, it is often integrated as a component into a bigger system. Consider, for example, a text classification problem in which we identify the polarity (i.e., positive, negative, and neutral) of a given text. We can build a text classification system by stacking a classifier on top of the encoder. Let $\mathrm{Classify}_\omega(\cdot)$ be a neural network with parameters ω. Then, the text classification model can be expressed in the form

$$\begin{aligned} \Pr_{\omega,\hat{\theta}}(\cdot|\mathbf{x}) &= \mathrm{Classify}_\omega(\mathbf{H}) \\ &= \mathrm{Classify}_\omega(\mathrm{Encode}_{\hat{\theta}}(\mathbf{x})) \end{aligned} \tag{7.3}$$

Here $\Pr_{\omega,\hat{\theta}}(\cdot|\mathbf{x})$ is a probability distribution over the label set $\{\mathrm{positive}, \mathrm{negative}, \mathrm{neutral}\}$, and the label with the highest probability in this distribution is selected as output. To keep the notation uncluttered, we will use $F_{\omega,\hat{\theta}}(\cdot)$ to denote $\mathrm{Classify}_\omega(\mathrm{Encode}_{\hat{\theta}}(\cdot))$.

Because the model parameters ω and $\hat{\theta}$ are not optimized for the classification task, we cannot directly use this model. Instead, we must use a modified version of the model that is adapted to the task. A typical way is to fine-tune the model by giving explicit labeling in downstream tasks. We can train $F_{\omega,\hat{\theta}}(\cdot)$ on a labeled dataset, treating it as a common supervised learning task. The outcome of the fine-tuning is the parameters $\tilde{\omega}$ and $\tilde{\theta}$ that are further optimized. Alternatively, we can freeze the encoder parameters $\hat{\theta}$ to maintain their pre-trained state, and focus solely on optimizing ω. This allows the classifier to be efficiently adapted to work in tandem with the pre-trained encoder.

Once we have obtained a fine-tuned model, we can use it to classify a new text. For example, suppose we have a comment posted on a travel website:

I love the food here. It's amazing!

[2]More precisely, in auto-regressive decoding of machine translation, each target-language token is generated based on both its preceding tokens and source-language sequence.

We first tokenize this text into tokens[3], and then feed the token sequence $\mathbf{x}_{\text{new}}$ into the fine-tuned model $F_{\tilde{\omega},\hat{\theta}}(\cdot)$. The model generates a distribution over classes by

$$F_{\tilde{\omega},\hat{\theta}}(\mathbf{x}_{\text{new}}) \quad = \quad \Big[\Pr(\text{positive}|\mathbf{x}_{\text{new}}) \quad \Pr(\text{negative}|\mathbf{x}_{\text{new}}) \quad \Pr(\text{neutral}|\mathbf{x}_{\text{new}})\Big] \tag{7.4}$$

And we select the label of the entry with the maximum value as output. In this example it is positive.

In general, the amount of labeled data used in fine-tuning is small compared to that of the pre-training data, and so fine-tuning is less computationally expensive. This makes the adaptation of pre-trained models very efficient in practice: given a pre-trained model and a downstream task, we just need to collect some labeled data, and slightly adjust the model parameters on this data. A more detailed discussion of fine-tuning can be found in Section 7.4.

2. Prompting of Pre-trained Models

Unlike sequence encoding models, sequence generation models are often employed independently to address language generation problems, such as question answering and machine translation, without the need for additional modules. It is therefore straightforward to fine-tune these models as complete systems on downstream tasks. For example, we can fine-tune a pre-trained encoder-decoder multilingual model on some bilingual data to improve its performance on a specific translation task.

Among various sequence generation models, a notable example is the large language models trained on very large amounts of data. These language models are trained to simply predict the next token given its preceding tokens. Although token prediction is such a simple task that it has long been restricted to "language modeling" only, it has been found to enable the learning of the general knowledge of languages by repeating the task a large number of times. The result is that the pre-trained large language models exhibit remarkably good abilities in token prediction, making it possible to transform numerous NLP problems into simple text generation problems through prompting the large language models. For example, we can frame the above text classification problem as a text generation task

I love the food here. It's amazing! I'm _____

Here __ indicates the word or phrase we want to predict (call it the **completion**). If the predicted word is *happy*, or *glad*, or *satisfied* or a related positive word, we can classify the text as positive. This example shows a simple prompting method in which we concatenate the input text with *I'm* to form a prompt. Then, the completion helps decide which label is assigned to the original text.

Given the strong performance of language understanding and generation of large language models, a prompt can instruct the models to perform more complex tasks. Here is a prompt where we prompt the LLM to perform polarity classification with an instruction.

[3]The text can be tokenized in many different ways. One of the simplest is to segment the text into English words and punctuations $\{\text{I}, \text{love}, \text{the}, \text{food}, \text{here}, ., \text{It}, \text{'s}, \text{amazing}, !\}$

> Assume that the polarity of a text is a label chosen from {positive, negative, neutral}. Identify the polarity of the input.
>
> Input: I love the food here. It's amazing!
>
> Polarity: _____

The first two sentences are a description of the task. Input and Polarity are indicators of the input and output, respectively. We expect the model to complete the text and at the same time give the correct polarity label. By using instruction-based prompts, we can adapt large language models to solve NLP problems without the need for additional training.

This example also demonstrates the zero-shot learning capability of large language models, which can perform tasks that were not observed during the training phase. Another method for enabling new capabilities in a neural network is few-shot learning. This is typically achieved through **in-context learning** (**ICT**). More specifically, we add some samples that demonstrate how an input corresponds to an output. These samples, known as **demonstrations**, are used to teach large language models how to perform the task. Below is an example involving demonstrations

> Assume that the polarity of a text is a label chosen from {positive, negative, neutral}. Identify the polarity of the input.
>
> Input: The traffic is terrible during rush hours, making it difficult to reach the airport on time.
>
> Polarity: Negative
>
> Input: The weather here is wonderful.
>
> Polarity: Positive
>
> Input: I love the food here. It's amazing!
>
> Polarity: _____

Prompting and in-context learning play important roles in the recent rise of large language models. We will discuss these issues more deeply in Chapter 9. However, it is worth noting that while prompting is a powerful way to adapt large language models, some tuning efforts are still needed to ensure the models can follow instructions accurately. Additionally, the fine-tuning process is crucial for aligning the values of these models with human values. More detailed discussions of fine-tuning can be found in Chapter 10.

7.2 Self-supervised Pre-training Tasks

In this section, we consider self-supervised pre-training approaches for different neural architectures, including decoder-only, encoder-only, and encoder-decoder architectures. We restrict our discussion to Transformers since they form the basis of most pre-trained models in NLP. However, pre-training is a broad concept, and so we just give a brief introduction to basic approaches in order to make this section concise.

7.2.1 Decoder-only Pre-training

The decoder-only architecture has been widely used in developing language models [Radford et al., 2018]. For example, we can use a Transformer decoder as a language model by simply removing cross-attention sub-layers from it. Such a model predicts the distribution of tokens at a position given its preceding tokens, and the output is the token with the maximum probability. The standard way to train this model, as in the language modeling problem, is to minimize a loss function over a collection of token sequences. Let $\text{Decoder}_\theta(\cdot)$ denote a decoder with parameters θ. At each position i, the decoder generates a distribution of the next tokens based on its preceding tokens $\{x_0, ..., x_i\}$, denoted by $\Pr_\theta(\cdot|x_0, ..., x_i)$ (or $\mathbf{p}^\theta_{i+1}$ for short). Suppose we have the gold-standard distribution at the same position, denoted by $\mathbf{p}^{\text{gold}}_{i+1}$. For language modeling, we can think of $\mathbf{p}^{\text{gold}}_{i+1}$ as a one-hot representation of the correct predicted word. We then define a loss function $\mathcal{L}(\mathbf{p}^\theta_{i+1}, \mathbf{p}^{\text{gold}}_{i+1})$ to measure the difference between the model prediction and the true prediction. In NLP, the log-scale cross-entropy loss is typically used.

Given a sequence of m tokens $\{x_0, ..., x_m\}$, the loss on this sequence is the sum of the loss over the positions $\{0, ..., m-1\}$, given by

$$\begin{aligned} \text{Loss}_\theta(x_0, ..., x_m) &= \sum_{i=0}^{m-1} \mathcal{L}(\mathbf{p}^\theta_{i+1}, \mathbf{p}^{\text{gold}}_{i+1}) \\ &= \sum_{i=0}^{m-1} \text{LogCrossEntropy}(\mathbf{p}^\theta_{i+1}, \mathbf{p}^{\text{gold}}_{i+1}) \end{aligned} \tag{7.5}$$

where $\text{LogCrossEntropy}(\cdot)$ is the log-scale cross-entropy, and $\mathbf{p}^{\text{gold}}_{i+1}$ is the one-hot representation of x_{i+1}.

This loss function can be extended to a set of sequences $\mathcal{D}$. In this case, the objective of pre-training is to find the best parameters that minimize the loss on $\mathcal{D}$

$$\hat{\theta} = \arg\min_\theta \sum_{\mathbf{x} \in \mathcal{D}} \text{Loss}_\theta(\mathbf{x}) \tag{7.6}$$

Note that this objective is mathematically equivalent to maximum likelihood estimation, and

can be re-expressed as

$$\begin{aligned}\hat{\theta} &= \arg\max_{\theta} \sum_{\mathbf{x}\in\mathcal{D}} \log \Pr_\theta(\mathbf{x}) \\ &= \arg\max_{\theta} \sum_{\mathbf{x}\in\mathcal{D}} \sum_{i=0}^{i-1} \log \Pr_\theta(x_{i+1}|x_0,...,x_i) \end{aligned} \tag{7.7}$$

With these optimized parameters $\hat{\theta}$, we can use the pre-trained language model $\text{Decoder}_{\hat{\theta}}(\cdot)$ to compute the probability $\Pr_{\hat{\theta}}(x_{i+1}|x_0,...,x_i)$ at each position of a given sequence.

7.2.2 Encoder-only Pre-training

As defined in Section 7.1.2, an encoder $\text{Encoder}_\theta(\cdot)$ is a function that reads a sequence of tokens $\mathbf{x} = x_0...x_m$ and produces a sequence of vectors $\mathbf{H} = \mathbf{h}_0...\mathbf{h}_m$[4]. Training this model is not straightforward, as we do not have gold-standard data for measuring how good the output of the real-valued function is. A typical approach to encoder pre-training is to combine the encoder with some output layers to receive supervision signals that are easier to obtain. Figure 7.2 shows a common architecture for pre-training Transformer encoders, where we add a Softmax layer on top of the Transformer encoder. Clearly, this architecture is the same as that of the decoder-based language model, and the output is a sequence of probability distributions

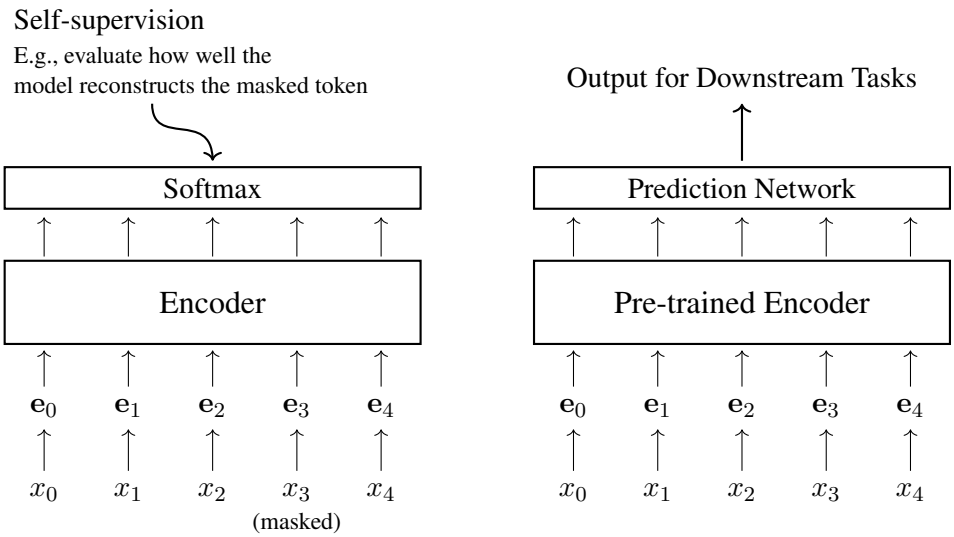
$$\begin{bmatrix} \mathbf{p}_1^{\mathbf{W},\theta} \\ \vdots \\ \mathbf{p}_m^{\mathbf{W},\theta} \end{bmatrix} = \text{Softmax}_{\mathbf{W}}(\text{Encoder}_\theta(\mathbf{x})) \tag{7.9}$$

Here $\mathbf{p}_i^{\mathbf{W},\theta}$ is the output distribution $\Pr(\cdot|\mathbf{x})$ at position i. We use $\text{Softmax}_{\mathbf{W}}(\cdot)$ to denote that the Softmax layer is parameterized by $\mathbf{W}$, that is, $\text{Softmax}_{\mathbf{W}}(\mathbf{H}) = \text{Softmax}(\mathbf{H}\cdot\mathbf{W})$. For notation simplicity, we will sometimes drop the superscripts $\mathbf{W}$ and θ affixed to each probability distribution.

The difference between this model and standard language models is that the output $\mathbf{p}_i$ has different meanings in encoder pre-training and language modeling. In language modeling, $\mathbf{p}_i$ is the probability distribution of predicting the next word. This follows an auto-regressive decoding process: a language model only observes the words up to position i and predicts the next. By contrast, in encoder pre-training, the entire sequence can be observed at once, and so it makes no sense to predict any of the tokens in this sequence.

[4] If we view $\mathbf{h}_i$ as a row vector, $\mathbf{H}$ can be written as

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_0 \\ \vdots \\ \mathbf{h}_m \end{bmatrix} \tag{7.8}$$



(a) Pre-training

(b) Applying the Pre-trained Encoder

Figure 7.2: Pre-training a Transformer encoder (left) and then applying the pre-trained encoder (right). In the pre-training phase, the encoder, together with a Softmax layer, is trained via self-supervision. In the application phase, the Softmax layer is removed, and the pre-trained encoder is combined with a prediction network to address specific problems. In general, for better adaptation to these tasks, the system is fine-tuned using labeled data.

1. Masked Language Modeling

One of the most popular methods of encoder pre-training is **masked language modeling**, which forms the basis of the well-known BERT model [Devlin et al., 2019]. The idea of masked language modeling is to create prediction challenges by masking out some of the tokens in the input sequence and training a model to predict the masked tokens. In this sense, the conventional language modeling problem, which is sometimes called **causal language modeling**, is a special case of masked language modeling: at each position, we mask the tokens in the right-context, and predict the token at this position using its left-context. However, in causal language modeling we only make use of the left-context in word prediction, while the prediction may depend on tokens in the right-context. By contrast, in masked language modeling, all the unmasked tokens are used for word prediction, leading to a bidirectional model that makes predictions based on both left and right-contexts.

More formally, for an input sequence $\mathbf{x} = x_0...x_m$, suppose that we mask the tokens at positions $\mathcal{A}(\mathbf{x}) = \{i_1, ..., i_u\}$. Hence we obtain a masked token sequence $\bar{\mathbf{x}}$ where the token at each position in $\mathcal{A}(\mathbf{x})$ is replaced with a special symbol $[\mathrm{MASK}]$. For example, for the following sequence

The early bird catches the worm

we may have a masked token sequence like this

The [MASK] bird catches the [MASK]

where we mask the tokens *early* and *worm* (i.e., $i_1 = 2$ and $i_2 = 6$).

Now we have two sequences $\mathbf{x}$ and $\bar{\mathbf{x}}$. The model is then optimized so that we can correctly predict $\mathbf{x}$ based on $\bar{\mathbf{x}}$. This can be thought of as an autoencoding-like process, and the training objective is to maximize the reconstruction probability $\Pr(\mathbf{x}|\bar{\mathbf{x}})$. Note that there is a simple position-wise alignment between $\mathbf{x}$ and $\bar{\mathbf{x}}$. Because an unmasked token in $\bar{\mathbf{x}}$ is the same as the token in $\mathbf{x}$ at the same position, there is no need to consider the prediction for this unmasked token. This leads to a simplified training objective which only maximizes the probabilities for masked tokens. We can express this objective in a maximum likelihood estimation fashion

$$(\widehat{\mathbf{W}}, \hat{\theta}) \quad = \quad \arg\max_{\mathbf{W},\theta} \sum_{\mathbf{x}\in\mathcal{D}} \sum_{i\in\mathcal{A}(\mathbf{x})} \log \Pr_i^{\mathbf{W},\theta}(x_i|\bar{\mathbf{x}}) \tag{7.10}$$

or alternatively express it using the cross-entropy loss

$$(\widehat{\mathbf{W}}, \hat{\theta}) \quad = \quad \arg\min_{\mathbf{W},\theta} \sum_{\mathbf{x}\in\mathcal{D}} \sum_{i\in\mathcal{A}(\mathbf{x})} \text{LogCrossEntropy}(\mathbf{p}_i^{\mathbf{W},\theta}, \mathbf{p}_i^{\text{gold}}) \tag{7.11}$$

where $\Pr_k^{\mathbf{W},\theta}(x_k|\bar{\mathbf{x}})$ is the probability of the true token x_k at position k given the corrupted input $\bar{\mathbf{x}}$, and $\mathbf{p}_k^{\mathbf{W},\theta}$ is the probability distribution at position k given the corrupted input $\bar{\mathbf{x}}$. To illustrate, consider the above example where two tokens of the sequence "*the early bird catches the worm*" are masked. For this example, the objective is to maximize the sum of log-scale probabilities

$$\begin{aligned} \text{Loss} \quad = \quad & \log \Pr(x_2 = \textit{early} | \bar{\mathbf{x}} = \text{[CLS] } \textit{The } \underbrace{\text{[MASK]}}_{\bar{x}_2} \textit{ bird catches the } \underbrace{\text{[MASK]}}_{\bar{x}_6}) + \\ & \log \Pr(x_6 = \textit{worm} | \bar{\mathbf{x}} = \text{[CLS] } \textit{The } \underbrace{\text{[MASK]}}_{\bar{x}_2} \textit{ bird catches the } \underbrace{\text{[MASK]}}_{\bar{x}_6}) \end{aligned} \tag{7.12}$$

Once we obtain the optimized parameters $\widehat{\mathbf{W}}$ and $\hat{\theta}$, we can drop $\widehat{\mathbf{W}}$. Then, we can further fine-tune the pre-trained encoder $\text{Encoder}_{\hat{\theta}}(\cdot)$ or directly apply it to downstream tasks.

2. Permuted Language Modeling

While masked language modeling is simple and widely applied, it introduces new issues. One drawback is the use of a special token, [MASK], which is employed only during training but not at test time. This leads to a discrepancy between training and inference. Moreover, the auto-encoding process overlooks the dependencies between masked tokens. For example, in the above example, the prediction of x_2 (i.e., the first masked token) is made independently of x_6 (i.e., the second masked token), though x_6 should be considered in the context of x_2.

These issues can be addressed using the **permuted language modeling** approach to pre-training [Yang et al., 2019]. Similar to causal language modeling, permuted language modeling involves making sequential predictions of tokens. However, unlike causal modeling where

predictions follow the natural sequence of the text (like left-to-right or right-to-left), permuted language modeling allows for predictions in any order. The approach is straightforward: we determine an order for token predictions and then train the model in a standard language modeling manner, as described in Section 7.2.1. Note that in this approach, the actual order of tokens in the text remains unchanged, and only the order in which we predict these tokens differs from standard language modeling. For example, consider a sequence of 5 tokens $x_0x_1x_2x_3x_4$. Let $\mathbf{e}_i$ represent the embedding of x_i (i.e., combination of the token embedding and positional embedding). In standard language modeling, we would generate this sequence in the order of $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4$. The probability of the sequence can be modeled via a generation process.

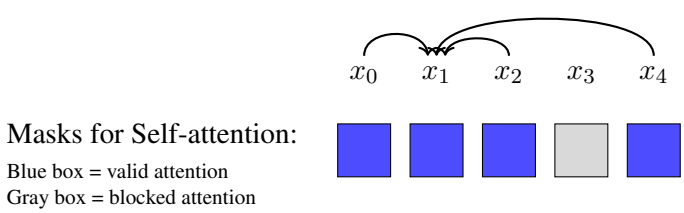
$$\begin{aligned}\Pr(\mathbf{x}) &= \Pr(x_0)\cdot\Pr(x_1|x_0)\cdot\Pr(x_2|x_0,x_1)\cdot\Pr(x_3|x_0,x_1,x_2)\cdot \\ &\quad \Pr(x_4|x_0,x_1,x_2,x_3) \\ &= \Pr(x_0)\cdot\Pr(x_1|\mathbf{e}_0)\cdot\Pr(x_2|\mathbf{e}_0,\mathbf{e}_1)\cdot\Pr(x_3|\mathbf{e}_0,\mathbf{e}_1,\mathbf{e}_2)\cdot \\ &\quad \Pr(x_4|\mathbf{e}_0,\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3) \end{aligned} \tag{7.13}$$

Now, let us consider a different order for token prediction: $x_0 \rightarrow x_4 \rightarrow x_2 \rightarrow x_1 \rightarrow x_3$. The sequence generation process can then be expressed as follows:

$$\begin{aligned}\Pr(\mathbf{x}) &= \Pr(x_0)\cdot\Pr(x_4|\mathbf{e}_0)\cdot\Pr(x_2|\mathbf{e}_0,\mathbf{e}_4)\cdot\Pr(x_1|\mathbf{e}_0,\mathbf{e}_4,\mathbf{e}_2)\cdot \\ &\quad \Pr(x_3|\mathbf{e}_0,\mathbf{e}_4,\mathbf{e}_2,\mathbf{e}_1) \end{aligned} \tag{7.14}$$

This new prediction order allows for the generation of some tokens to be conditioned on a broader context, rather than being limited to just the preceding tokens as in standard language models. For example, in generating x_3, the model considers both its left-context (i.e., $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2$) and right-context (i.e., $\mathbf{e}_4$). The embeddings $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_4$ incorporate the positional information of x_0, x_1, x_2, x_4, preserving the original order of the tokens. As a result, this approach is somewhat akin to masked language modeling: we mask out x_3 and use its surrounding tokens x_0, x_1, x_2, x_4 to predict this token.

The implementation of permuted language models is relatively easy for Transformers. Because the self-attention model is insensitive to the order of inputs, we do not need to explicitly reorder the sequence to have a factorization like Eq. (7.14). Instead, permutation can be done by setting appropriate masks for self-attention. For example, consider the case of computing $\Pr(x_1|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2)$. We can place x_0, x_1, x_2, x_3, x_4 in order and block the attention from x_3 to x_1 in self-attention, as illustrated below



For a more illustrative example, we compare the self-attention masking results of causal language modeling, masked language modeling and permuted language modeling in Figure 7.3.

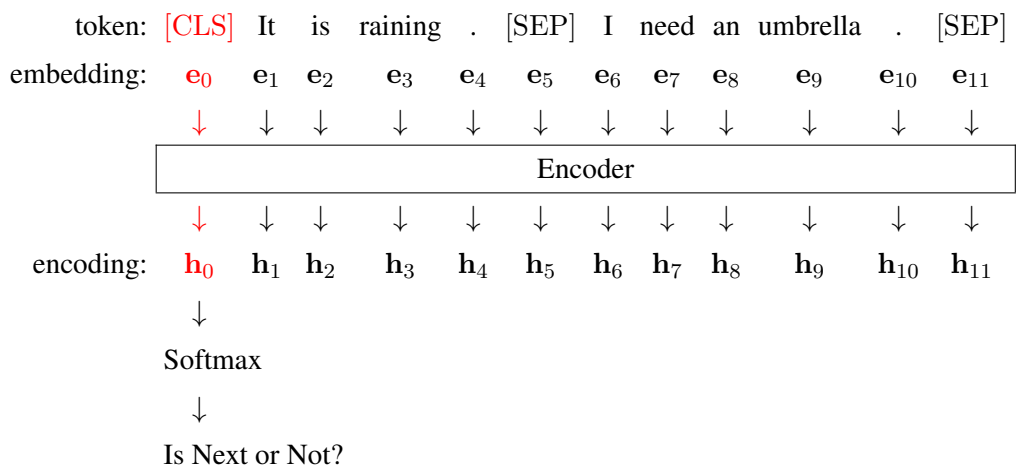
3. Pre-training Encoders as Classifiers

Another commonly-used idea to train an encoder is to consider classification tasks. In self-supervised learning, this is typically done by creating new classification challenges from the unlabeled text. There are many different ways to design the classification tasks. Here we present two popular tasks.

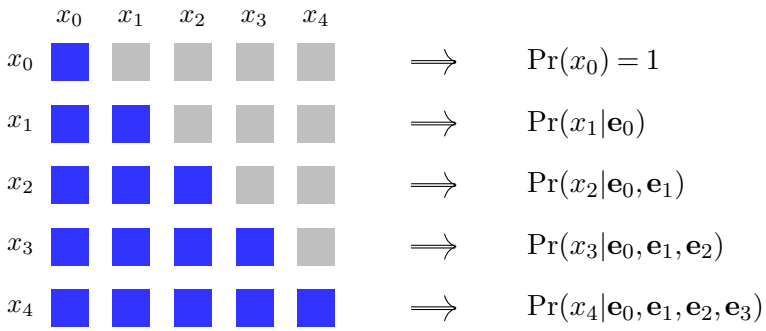
A simple method, called **next sentence prediction** (**NSP**), is presented in BERT's original paper [Devlin et al., 2019]. The assumption of NSP is that a good text encoder should capture the relationship between two sentences. To model such a relationship, in NSP we can use the output of encoding two consecutive sentences Sent_A and Sent_B to determine whether Sent_B is the next sentence following Sent_A. For example, suppose $\text{Sent}_A =$ *'It is raining .'* and $\text{Sent}_B =$ *'I need an umbrella .'*. The input sequence of the encoder could be

$$\text{[CLS] It is raining . [SEP] I need an umbrella . [SEP]}$$

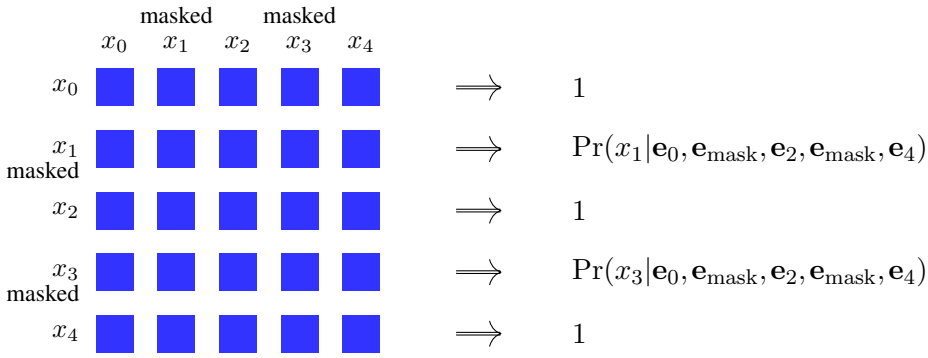
where [CLS] is the start symbol (i.e., x_0) which is commonly used in encoder pre-training, and [SEP] is a separator that separates the two sentences. The processing of this sequence follows a standard procedure of Transformer encoding: we first represent each token x_i as its corresponding embedding $\mathbf{e}_i$, and then feed the embedding sequence $\{\mathbf{e}_0, ..., \mathbf{e}_m\}$ into the encoder to obtain the output sequence $\{\mathbf{h}_0, ..., \mathbf{h}_m\}$. Since $\mathbf{h}_0$ is generally considered as the representation of the entire sequence, we add a Softmax layer on top of it to construct a binary classification system. This process is illustrated as follows



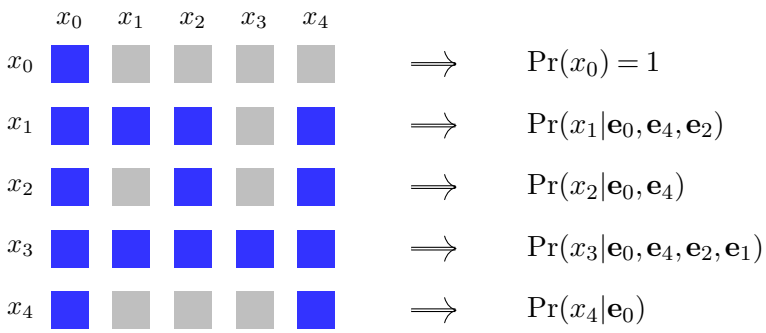
In order to generate training samples, we need two sentences each time, one for Sent_A and the other for Sent_B. A simple way to do this is to utilize the natural sequence of two consecutive sentences in the text. For example, we obtain a positive sample by using actual consecutive



(a) Causal Language Modeling (order: $x_0 \to x_1 \to x_2 \to x_3 \to x_4$)



(b) Masked Language Modeling (order: $x_0, [\text{MASK}], x_2, [\text{MASK}], x_4 \to x_1, x_3$)

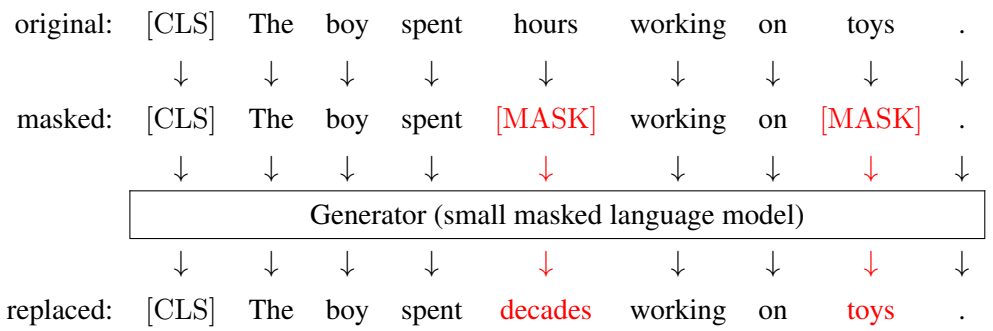


(c) Permuted Language Modeling (order: $x_0 \to x_4 \to x_2 \to x_1 \to x_3$)

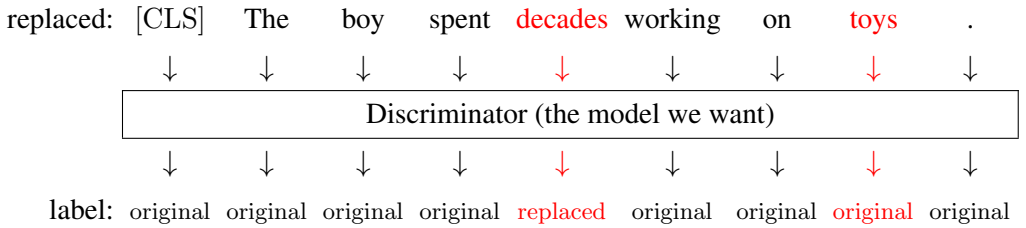
Figure 7.3: Comparison of self-attention masking results of causal language modeling, masked language modeling and permuted language modeling. The gray cell denotes the token at position j does not attend to the token at position i. The blue cell (i, j) denotes that the token at position j attends to the token at position i. $\mathbf{e}_{\text{mask}}$ represents the embedding of the symbol $[\text{MASK}]$, which is a combination of the token embedding and the positional embedding.

sentences, and a negative sample by using randomly sampled sentences. Consequently, training this model is the same as training a classifier. Typically, NSP is used as an additional training loss function for pre-training based on masked language modeling.

A second example of training Transformer encoders as classifiers is to apply classification-based supervision signals to each output of an encoder. For example, Clark et al. [2019] in their ELECTRA model, propose training a Transformer encoder to identify whether each input token is identical to the original input or has been altered in some manner. The first step of this method is to generate a new sequence from a given sequence of tokens, where some of the tokens are altered. To do this, a small masked language model (call it the generator) is applied: we randomly mask some of the tokens, and train this model to predict the masked tokens. For each training sample, this masked language model outputs a token at each masked position, which might be different from the original token. At the same time, we train another Transformer encoder (call it the discriminator) to determine whether each predicted token is the same as the original token or altered. More specifically, we use the generator to generate a sequence where some of the tokens are replaced. Below is an illustration.



Then, we use the discriminator to label each of these tokens as original or replaced, as follows



For training, the generator is optimized as a masked language model with maximum likelihood estimation, and the discriminator is optimized as a classifier using a classification-based loss. In ELECTRA, the maximum likelihood-based loss and the classification-based loss are combined for jointly training both the generator and discriminator. An alternative approach is to use generative adversarial networks (GANs), that is, the generator is trained to fool the discriminator, and the discriminator is trained to distinguish the output of the generator from the true distribution. However, GAN-style training complicates the training task and is more

difficult to scale up. Nevertheless, once training is complete, the generator is discarded, and the encoding part of the discriminator is applied as the pre-trained model for downstream tasks.

7.2.3 Encoder-Decoder Pre-training

In NLP, encoder-decoder architectures are often used to model sequence-to-sequence problems, such as machine translation and question answering. In addition to these typical sequence-to-sequence problems in NLP, encoder-decoder models can be extended to deal with many other problems. A simple idea is to consider text as both the input and output of a problem, and so we can directly apply encoder-decoder models. For example, given a text, we can ask a model to output a text describing the sentiment of the input text, such as *positive*, *negative*, and *neutral*.

Such an idea allows us to develop a single text-to-text system to address any NLP problem. We can formulate different problems into the same text-to-text format. We first train an encoder-decoder model to gain general-purpose knowledge of language via self-supervision. This model is then fine-tuned for specific downstream tasks using targeted text-to-text data.

1. Masked Encoder-Decoder Pre-training

In Raffel et al. [2020]'s **T5** model, many different tasks are framed as the same text-to-text task. Each sample in T5 follows the format

$$\text{Source Text} \quad \rightarrow \quad \text{Target Text}$$

Here $\rightarrow$ separates the source text, which consists of a task description or instruction and the input given to the system, from the target text, which is the response to the input task. As an example, consider a task of translating from Chinese to English. A training sample can be expressed as

$$[\text{CLS}]\ \text{Translate from Chinese to English: 你好！} \quad \rightarrow \quad \langle s \rangle\ \text{Hello!}$$

where $[\text{CLS}]$ and $\langle s \rangle$ are the start symbols on the source and target sides, respectively[5].

[5]We could use the same start symbol for different sequences. Here we use different symbols to distinguish the sequences on the encoder and decoder-sides.

Likewise, we can express other tasks in the same way. For example

[CLS] Answer: when was Albert Einstein born?
→ ⟨s⟩ He was born on March 14, 1879.

[CLS] Simplify: the professor, who has published numerous papers in his field, will be giving a lecture on the topic next week.
→ ⟨s⟩ The experienced professor will give a lecture next week.

[CLS] Text: John bought a new car. Hypothesis: John has a car.
→ ⟨s⟩ Entailment

[CLS] Score the translation from English to Chinese. English: when in Rome, do as the Romans do. Chinese: 人 在 罗马 就 像 罗马 人 一样 做事 。
→ ⟨s⟩ 0.81

where instructions are highlighted in gray. An interesting case is that in the last example we reframe the scoring problem as the text generation problem. Our goal is to generate a text representing the number 0.81, rather than outputting it as a numerical value.

The approach described above provides a new framework of universal language understanding and generation. Both the task instructions and the problem inputs are provided to the system in text form. The system then follows the instructions to complete the task. This method puts different problems together, with the benefit of training a single model that can perform many tasks simultaneously.

In general, fine-tuning is necessary for adapting the pre-trained model to a specific downstream task. In this process, one can use different ways to instruct the model for the task, such as using a short name of the task as the prefix to the actual input sequence or providing a detailed description of the task. Since the task instructions are expressed in text form and involved as part of the input, the general knowledge of instruction can be gained through learning the language understanding models in the pre-training phase. This may help enable zero-shot learning. For example, pre-trained models can generalize to address new problems where the task instructions have never been encountered.

There have been several powerful methods of self-supervised learning for either Transformer encoders or decoders. Applying these methods to pre-train encoder-decoder models is relatively straightforward. One common choice is to train encoder-decoder models as language models. For example, the encoder receives a sequence prefix, while the decoder generates the remaining sequence. However, this differs from standard causal language modeling, where the entire sequence is autoregressively generated from the first token. In our case, the encoder processes the prefix at once, and then the decoder predicts subsequent tokens in the manner of causal language modeling. Put more precisely, this is a **prefix language modeling** problem: a

language model predicts the subsequent sequence given a prefix, which serves as the context for prediction.

Consider the following example

$$\underbrace{[\text{CLS}]\ \text{The puppies are frolicking}}_{\text{Prefix}} \rightarrow \underbrace{\langle s \rangle\ \text{outside the house .}}_{\text{Subsequent Sequence}}$$

We can directly train an encoder-decoder model using examples like this. Then, the encoder learns to understand the prefix, and the decoder learns to continue writing based on this understanding. For large-scale pre-training, it is easy to create a large number of training examples from unlabeled text.

It is worth noting that for pre-trained encoder-decoder models to be effective in multi-lingual and cross-lingual tasks, such as machine translation, they should be trained with multi-lingual data. This typically requires that the vocabulary includes tokens from all the languages. By doing so, the models can learn shared representations across different languages, thereby enabling capabilities in both language understanding and generation in a multi-lingual and cross-lingual context.

A second approach to pre-training encoder-decoder models is masked language modeling. In this approach, as discussed in Section 7.2.2, tokens in a sequence are randomly replaced with a mask symbol, and the model is then trained to predict these masked tokens based on the entire masked sequence.

As an illustration, consider the task of masking and reconstructing the sentence

$$\text{The puppies are frolicking outside the house .}$$

By masking two tokens (say, *frolicking* and *the*), we have the BERT-style input and output of the model, as follows

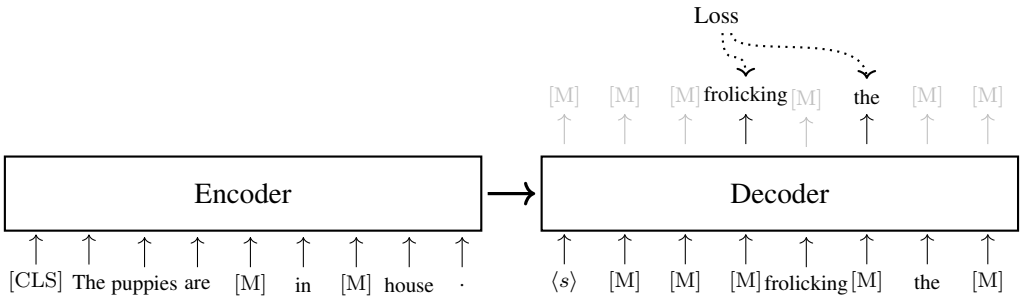
$$\begin{aligned} & [\text{CLS}]\ \text{The puppies are}\ [\text{MASK}]\ \text{outside}\ [\text{MASK}]\ \text{house .} \\ \rightarrow\ & \langle s \rangle\ \_\_\ \_\_\ \_\_\ \text{frolicking}\ \_\_\ \text{the}\ \_\_\ \_\_ \end{aligned}$$

Here __ denotes the masked position at which we do not make token predictions. By varying the percentage of the tokens in the text, this approach can be generalized towards either BERT-style training or language modeling-style training [Song et al., 2019]. For example, if we mask out all the tokens, then the model is trained to generate the entire sequence

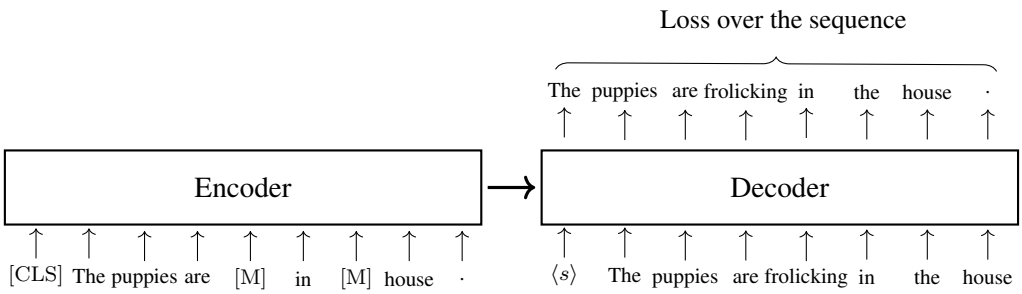
$$\begin{aligned} & [\text{CLS}]\ [\text{MASK}]\ [\text{MASK}]\ [\text{MASK}]\ [\text{MASK}]\ [\text{MASK}]\ [\text{MASK}]\ [\text{MASK}]\ [\text{MASK}] \\ \rightarrow\ & \langle s \rangle\ \text{The puppies are frolicking outside the house .} \end{aligned}$$

In this case, we train the decoder as a language model.

Note that, in the context of the encoder-decoder architecture, we can use the encoder to read the masked sequence, and use the decoder to predict the original sequence. With this objective, we essentially have a denoising autoencoder: the encoder transforms a corrupted



(a) Training an encoder-decoder model with BERT-style masked language modeling



(b) Training an encoder-decoder model with denoising autoencoding

Figure 7.4: Training an encoder-decoder model using BERT-style and denoising autoencoding methods. In both methods, the input to the encoder is a corrupted token sequence where some tokens are masked and replaced with [MASK] (or [M] for short). The decoder predicts these masked tokens, but in different ways. In BERT-style training, the decoder only needs to compute the loss for the masked tokens, while the remaining tokens in the sequence can be simply treated as [MASK] tokens. In denoising autoencoding, the decoder predicts the sequence of all tokens in an autoregressive manner. As a result, the loss is obtained by accumulating the losses of all these tokens, as in standard language modeling.

input into some hidden representation, and the decoder reconstructs the uncorrupted input from this hidden representation. Here is an example of input and output for denoising training.

$$\begin{aligned} & \text{[CLS] The puppies are [MASK] outside [MASK] house .} \\ \rightarrow\ & \langle s \rangle \text{ The puppies are frolicking outside the house .} \end{aligned}$$

By learning to map from this corrupted sequence to its uncorrupted counterpart, the model gains the ability to understand on the encoder side and to generate on the decoder side. See Figure 7.4 for an illustration of how an encoder-decoder model is trained with BERT-style and denoising autoencoding objectives.

As we randomly select tokens for masking, we can certainly mask consecutive tokens

[Joshi et al., 2020]. Here is an example.

$$\begin{aligned} & [\text{CLS}] \text{ The puppies are } [\text{MASK}] \text{ outside } [\text{MASK}] \ [\text{MASK}] \ . \\ \rightarrow \ & \langle s \rangle \text{ The puppies are frolicking outside the house .} \end{aligned}$$

Another way to consider consecutive masked tokens is to represent them as spans. Here we follow Raffel et al. [2020]'s work, and use [X], [Y] and [Z] to denote sentinel tokens that cover one or more consecutive masked tokens. Using this notation, we can re-express the above training example as

$$\begin{aligned} & [\text{CLS}] \text{ The puppies are } [\text{X}] \text{ outside } [\text{Y}] \ . \\ \rightarrow \ & \langle s \rangle \ [\text{X}] \text{ frolicking } [\text{Y}] \text{ the house } [\text{Z}] \end{aligned}$$

The idea is that we represent the corrupted sequence as a sequence containing placeholder slots. The training task is to fill these slots with the correct tokens using the surrounding context. An advantage of this approach is that the sequences used in training would be shorter, making the training more efficient. Note that masked language modeling provides a very general framework for training encoder-decoder models. Various settings can be adjusted to have different training versions, such as altering the percentage of tokens masked and the maximum length of the masked spans.

2. Denoising Training

If we view the problem of training encoder-decoder models as a problem of training denoising autoencoders, there will typically be many different methods for introducing input corruption and reconstructing the input. For instance, beyond randomly masking tokens, we can also alter some of them or rearrange their order.

Suppose we have an encoder-decoder model that can map an input sequence $\mathbf{x}$ to an output sequence $\mathbf{y}$

$$\begin{aligned} \mathbf{y} &= \text{Decode}_{\omega}(\text{Encode}_{\theta}(\mathbf{x})) \\ &= \text{Model}_{\theta,\omega}(\mathbf{x}) \end{aligned} \tag{7.15}$$

where θ and ω are the parameters of the encoder and the decoder, respectively. In denoising autoencoding problems, we add some noise to $\mathbf{x}$ to obtain a noisy, corrupted input $\mathbf{x}_{\text{noise}}$. By feeding $\mathbf{x}_{\text{noise}}$ into the encoder, we wish the decoder to output the original input. The training objective can be defined as

$$(\hat{\theta}, \hat{\omega}) = \arg\min_{\theta,\omega} \text{Loss}(\text{Model}_{\theta,\omega}(\mathbf{x}_{\text{noise}}), \mathbf{x}) \tag{7.16}$$

Here the loss function $\text{Loss}(\text{Model}_{\theta,\omega}(\mathbf{x}_{\text{noise}}), \mathbf{x})$ evaluates how well the model $\text{Model}_{\theta,\omega}(\mathbf{x}_{\text{noise}})$ reconstructs the original input $\mathbf{x}$. We can choose the cross-entropy loss as usual.

As the model architecture and the training approach have been developed, the remaining

issue is the corruption of the input. Lewis et al. [2020], in their **BART** model, propose corrupting the input sequence in several different ways.

- **Token Masking**. This is the same masking method that we used in masked language modeling. The tokens in the input sequence are randomly selected and masked.
- **Token Deletion**. This method is similar to token masking. However, rather than replacing the selected tokens with a special symbol [MASK], these tokens are removed from the sequence. See the following example for a comparison of the token masking and token deletion methods.

  Original ($\mathbf{x}$): The puppies are frolicking outside the house .
  Token Masking ($\mathbf{x}_{\text{noise}}$): The puppies are [MASK] outside [MASK] house .
  Token Deletion ($\mathbf{x}_{\text{noise}}$): The puppies are ~~frolicking~~ outside ~~the~~ house .

  where the underlined tokens in the original sequence are masked or deleted.
- **Span Masking**. Non-overlapping spans are randomly sampled over the sequence. Each span is masked by [MASK]. We also consider spans of length 0, and, in such cases, [MASK] is simply inserted at a position in the sequence. For example, we can use span masking to corrupt the above sequence as

  Original ($\mathbf{x}$): The 0 puppies are frolicking outside the house .
  Span Masking ($\mathbf{x}_{\text{noise}}$): The [MASK] puppies are [MASK] house .

  Here the span *frolicking outside the* is replaced with a single [MASK]. 0 indicates a length-0 span, and so we insert an [MASK] between *The* and *puppies*. Span masking introduces new prediction challenges in which the model needs to know how many tokens are generated from a span. This problem is very similar to fertility modeling in machine translation [Brown et al., 1993].

If we consider a sequence consisting of multiple sentences, additional methods of corruption can be applied. In the BART model, there are two such methods.

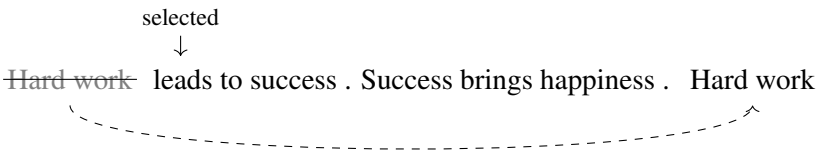
- **Sentence Reordering**. This method randomly permutes the sentences so that the model can learn to reorder sentences in a document. Consider, for example, two consecutive sentences

  Hard work leads to success . Success brings happiness .

  We can reorder the two sentences to have a corrupted input sequence

  Success brings happiness . Hard work leads to success .

- **Document Rotation**. The goal of this task is to identify the start token of the sequence. First, a token is randomly selected from the sequence. Then, the sequence is rotated so that the selected token is the first token. For example, suppose we select the token *leads* from the above sequence. The rotated sequence is



where the subsequence *Hard work* before *leads* is appended to the end of the sequence.

For pre-training, we can apply multiple corruption methods to learn robust models, for example, we randomly choose one of them for each training sample. In practice, the outcome of encoder-decoder pre-training depends heavily on the input corruption methods used, and so we typically need to choose appropriate training objectives through careful experimentation.

7.2.4 Comparison of Pre-training Tasks

So far, we have discussed a number of pre-training tasks. Since the same training objective can apply to different architectures (e.g., using masked language modeling for both encoder-only and encoder-decoder pre-training), categorizing pre-training tasks based solely on model architecture does not seem ideal. Instead, we summarize these tasks based on the training objectives.

- **Language Modeling**. Typically, this approach refers to an auto-regressive generation procedure of sequences. At one time, it predicts the next token based on its previous context.
- **Masked Language Modeling**. Masked Language Modeling belongs to a general mask-predict framework. It randomly masks tokens in a sequence and predicts these tokens using the entire masked sequence.
- **Permuted Language Modeling**. Permuted language modeling follows a similar idea to masked language modeling, but considers the order of (masked) token prediction. It reorders the input sequence and predicts the tokens sequentially. Each prediction is based on some context tokens that are randomly selected.
- **Discriminative Training**. In discriminative training, supervision signals are created from classification tasks. Models for pre-training are integrated into classifiers and trained together with the remaining parts of the classifiers to enhance their classification performance.
- **Denoising Autoencoding**. This approach is applied to the pre-training of encoder-decoder models. The input is a corrupted sequence and the encoder-decoder models are trained to reconstruct the original sequence.

Table 7.1 illustrates these methods and their variants using examples. The use of these examples does not distinguish between models, but we mark the model architectures where the pre-training tasks can be applied. In each example, the input consists of a token sequence, and the output is either a token sequence or some probabilities. For generation tasks, such as language modeling, superscripts are used to indicate the generation order on the target side. If the superscripts are omitted, it indicates that the output sequence can be generated

Method	Enc	Dec	E-D	Input	Output
Causal LM		•	•		The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7}
Prefix LM		•	•	[C] The kitten is	chasing^1 the^2 ball^3 .^{4}
Masked LM	•		•	[C] The kitten [M] chasing the [M] .	_ _ is _ _ ball _
MASS-style	•		•	[C] The kitten [M] [M] [M] ball .	_ _ is chasing the _ _
BERT-style	•		•	[C] The kitten [M] playing the [M] .	_ kitten is chasing _ ball _
Permuted LM	•			[C] The kitten is chasing the ball .	The^5 kitten^7 is^6 chasing^1 the^4 ball^2 .^{3}
Next Sentence Prediction	•			[C] The kitten is chasing the ball . Birds eat worms .	Pr(IsNext | representation-of-[C])
Sentence Comparison	•			Encode a sentence as $\mathbf{h}_a$ and another sentence as $\mathbf{h}_b$	Score($\mathbf{h}_a, \mathbf{h}_b$)
Token Classification	•			[C] The kitten is chasing the ball .	Pr(·|The) Pr(·|kitten) ... Pr(·|.)
Token Reordering			•	[C] . kitten the chasing The is ball	The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7}
Token Deletion			•	[C] The kitten is ~~chasing~~ the ball .	The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7}
Span Masking			•	[C] The kitten [M] is [M] .	The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7}
Sentinel Masking			•	[C] The kitten [X] the [Y]	[X]^1 is^2 chasing^3 [Y]^4 ball^5 .^{6}
Sentence Reordering			•	[C] The ball rolls away swiftly . The kitten is chasing the ball .	The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7} The^8 ball^9 rolls^{10} away^{11} swiftly^{12} .^{13}
Document Rotation			•	[C] chasing the ball . The ball rolls away swiftly . The kitten is	The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7} The^8 ball^9 rolls^{10} away^{11} swiftly^{12} .^{13}

Table 7.1: Comparison of pre-training tasks, including language modeling, masked language modeling, permuted language modeling, discriminative training, and denoising autoencoding. [C] = [CLS], [M] = [MASK], [X], [Y] = sentinel tokens. Enc, Dec and E-D indicate whether the approach can be applied to encoder-only, decoder-only, encoder-decoder models, respectively. For generation tasks, superscripts are used to represent the order of the tokens.

either autoregressively or simultaneously. On the source side, we assume that the sequence undergoes a standard Transformer encoding process, meaning that each token can see the entire sequence in self-attention. The only exception is in permuted language modeling, where an autoregressive generation process is implemented by setting attention masks on the encoder side. To simplify the discussion, we remove the token $\langle s \rangle$ from the target-side of each example.

While these pre-training tasks are different, it is possible to compare them in the same framework and experimental setup [Dong et al., 2019; Raffel et al., 2020; Lewis et al., 2020]. Note that we cannot list all the pre-training tasks here as there are many of them. For more discussions on pre-training tasks, the interested reader may refer to some surveys on this topic [Qiu et al., 2020; Han et al., 2021].

7.3 Example: BERT

In this section, we introduce BERT models, which are among the most popular and widely used pre-trained sequence encoding models in NLP.

7.3.1 The Standard Model

The standard BERT model, which is proposed in Devlin et al. [2019]'s work, is a Transformer encoder trained using both masked language modeling and next sentence prediction tasks. The loss used in training this model is a sum of the loss of the two tasks.

$$\text{Loss}_{\text{BERT}} = \text{Loss}_{\text{MLM}} + \text{Loss}_{\text{NSP}} \tag{7.17}$$

As is regular in training deep neural networks, we optimize the model parameters by minimizing this loss. To do this, a number of training samples are collected. During training, a batch of training samples is randomly selected from this collection at a time, and $\text{Loss}_{\text{BERT}}$ is accumulated over these training samples. Then, the model parameters are updated via gradient descent or its variants. This process is repeated many times until some stopping criterion is satisfied, such as when the training loss converges.

1. Loss Functions

In general, BERT models are used to represent a single sentence or a pair of sentences, and thus can handle various downstream language understanding problems. In this section we assume that the input representation is a sequence containing two sentences Sent_A and Sent_B, expressed as

$$[\text{CLS}]\ \ \text{Sent}_A\ \ [\text{SEP}]\ \ \text{Sent}_B\ \ [\text{SEP}]$$

Here we follow the notation in BERT's paper and use $[\text{SEP}]$ to denote the separator.

Given this sequence, we can obtain Loss_{MLM} and Loss_{NSP} separately. For masked language modeling, we predict a subset of the tokens in the sequence. Typically, a certain percentage of the tokens are randomly selected, for example, in the standard BERT model, 15% of the tokens in each sequence are selected. Then the sequence is modified in three ways

- **Token Masking**. 80% of the selected tokens are masked and replaced with the symbol $[\text{MASK}]$. For example

Original:	[CLS] It is raining . [SEP] I need an umbrella . [SEP]
Masked:	[CLS] It is [MASK] . [SEP] I need [MASK] umbrella . [SEP]

  where the selected tokens are underlined. Predicting masked tokens makes the model learn to represent tokens from their surrounding context.
- **Random Replacement**. 10% of the selected tokens are changed to a random token. For

example

Original:	[CLS] It is raining . [SEP] I need an umbrella . [SEP]
Random Token:	[CLS] It is raining . [SEP] I need an hat . [SEP]

This helps the model learn to recover a token from a noisy input.

- **Unchanged**. 10% of the selected tokens are kept unchanged. For example,

Original:	[CLS] It is raining . [SEP] I need an umbrella . [SEP]
Unchanged Token:	[CLS] It is raining . [SEP] I need an umbrella . [SEP]

This is not a difficult prediction task, but can guide the model to use easier evidence for prediction.

Let $\mathcal{A}(\mathbf{x})$ be the set of selected positions of a given token sequence $\mathbf{x}$, and $\bar{\mathbf{x}}$ be the modified sequence of $\mathbf{x}$. The loss function of masked language modeling can be defined as

$$\text{Loss}_{\text{MLM}} = -\sum_{i \in \mathcal{A}(\mathbf{x})} \log \Pr_i(x_i|\bar{\mathbf{x}}) \tag{7.18}$$

where $\Pr_i(x_i|\bar{\mathbf{x}})$ is the probability of predicting x_i at the position i given $\bar{\mathbf{x}}$. Figure 7.5 shows a running example of computing Loss_{MLM}.

For next sentence prediction, we follow the method described in Section 7.2.2. Each training sample is classified into a label set $\{\text{IsNext}, \text{NotNext}\}$, for example,

Sequence:	[CLS] It is raining . [SEP] I need an umbrella . [SEP]
Label:	IsNext

Sequence:	[CLS] The cat sleeps on the windowsill . [SEP] Apples grow on trees . [SEP]
Label:	NotNext

The output vector of the encoder for the first token [CLS] is viewed as the sequence representation, denoted by $\mathbf{h}_{\text{cls}}$ (or $\mathbf{h}_0$). A classifier is built on top of $\mathbf{h}_{\text{cls}}$. Then, we can compute the probability of a label c given $\mathbf{h}_{\text{cls}}$, i.e., $\Pr(c|\mathbf{h}_{\text{cls}})$. There are many loss functions one can choose for classification problems. For example, in maximum likelihood training, we can define Loss_{NSP} as

$$\text{Loss}_{\text{NSP}} = -\log \Pr(c_{\text{gold}}|\mathbf{h}_{\text{cls}}) \tag{7.19}$$

where c_{gold} is the correct label for this sample.

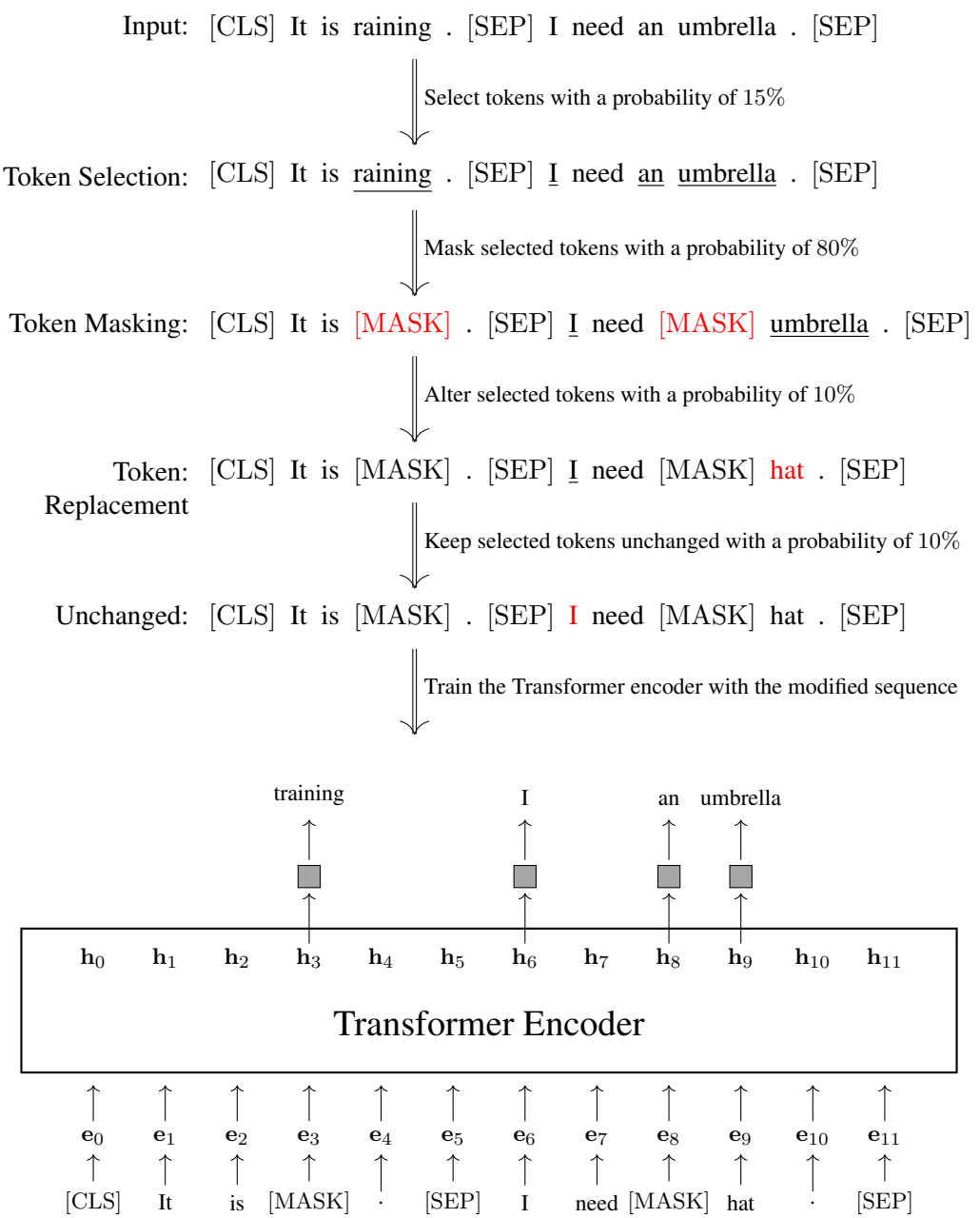


Figure 7.5: A running example of BERT-style masked language modeling. First, 15% of the tokens are randomly selected. These selected tokens are then processed in one of three ways: replaced with a [MASK] token (80% of the time), replaced with a random token (10% of the time), or kept unchanged (10% of the time). The model is trained to predict these selected tokens based on the modified sequence. $\mathbf{e}_i$ represents the embedding of the token at the position i. Gray boxes represent the Softmax layers.

2. Model Setup

As shown in Figure **??**, BERT models are based on the standard Transformer encoder architecture. The input is a sequence of embeddings, each being the sum of the token embedding, the

positional embedding, and the segment embedding.

$$\mathbf{e} \quad = \quad \mathbf{x} + \mathbf{e}_{\text{pos}} + \mathbf{e}_{\text{seg}} \tag{7.20}$$

Both the token embedding ($\mathbf{x}$) and positional embedding ($\mathbf{e}_{\text{pos}}$) are regular, as in Transformer models. The segment embedding ($\mathbf{e}_{\text{seg}}$) is a new type of embedding that indicates whether a token belongs to Sent_A or Sent_B. This can be illustrated by the following example.

Token	[CLS]	It	is	raining	.	[SEP]	I	need	an	umbrella	.	[SEP]
$\mathbf{x}$	$\mathbf{x}_0$	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	$\mathbf{x}_6$	$\mathbf{x}_7$	$\mathbf{x}_8$	$\mathbf{x}_9$	$\mathbf{x}_{10}$	$\mathbf{x}_{11}$
$\mathbf{e}_{\text{pos}}$	PE(0)	PE(1)	PE(2)	PE(3)	PE(4)	PE(5)	PE(6)	PE(7)	PE(8)	PE(9)	PE(10)	PE(11)
$\mathbf{e}_{\text{seg}}$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$

The main part of BERT models is a multi-layer Transformer network. A Transformer layer consists of a self-attention sub-layer and an FFN sub-layer. Both of them follow the post-norm architecture: $\text{output} = \text{LNorm}(F(\text{input}) + \text{input})$, where $F(\cdot)$ is the core function of the sub-layer (either a self-attention model or an FFN), and $\text{LNorm}(\cdot)$ is the layer normalization unit. Typically, a number of Transformer layers are stacked to form a deep network. At each position of the sequence, the output representation is a real-valued vector which is produced by the last layer of the network.

There are several aspects one may consider in developing BERT models.

- **Vocabulary Size** ($|V|$). In Transformers, each input token is represented as an entry in a vocabulary V. Large vocabularies can cover more surface form variants of words, but may lead to increased storage requirements.
- **Embedding Size** (d_e). Every token is represented as a d_e-dimensional real-valued vector. As presented above, this vector is the sum of the token embedding, positional embedding, and segment embedding, all of which are also d_e-dimensional real-valued vectors.
- **Hidden Size** (d). The input and output of a sub-layer are of d dimensions. Besides, most of the hidden states of a sub-layer are d-dimensional vectors. In general, d can be roughly viewed as the width of the network.
- **Number of Heads** (n_{head}). In self-attention sub-layers, one needs to specify the number of heads used in multi-head self-attention. The larger this number is, the more sub-spaces in which attention is performed. In practical systems, we often set $n_{\text{head}} \geq 4$.
- **FFN Hidden Size** (d_{ffn}). The size of the hidden layer of the FFNs used in Transformers is typically larger than d. For example, a typical setting is $d_{\text{ffn}} = 4d$. For larger Transformers, such as recent large models, d_{ffn} may be set to a very large value.
- **Model Depth** (L). Using deep networks is an effective way to improve the expressive power of Transformers. For BERT models, L is typically set to 12 or 24. However, networks with even greater depth are also feasible and can be applied for further enhancements.

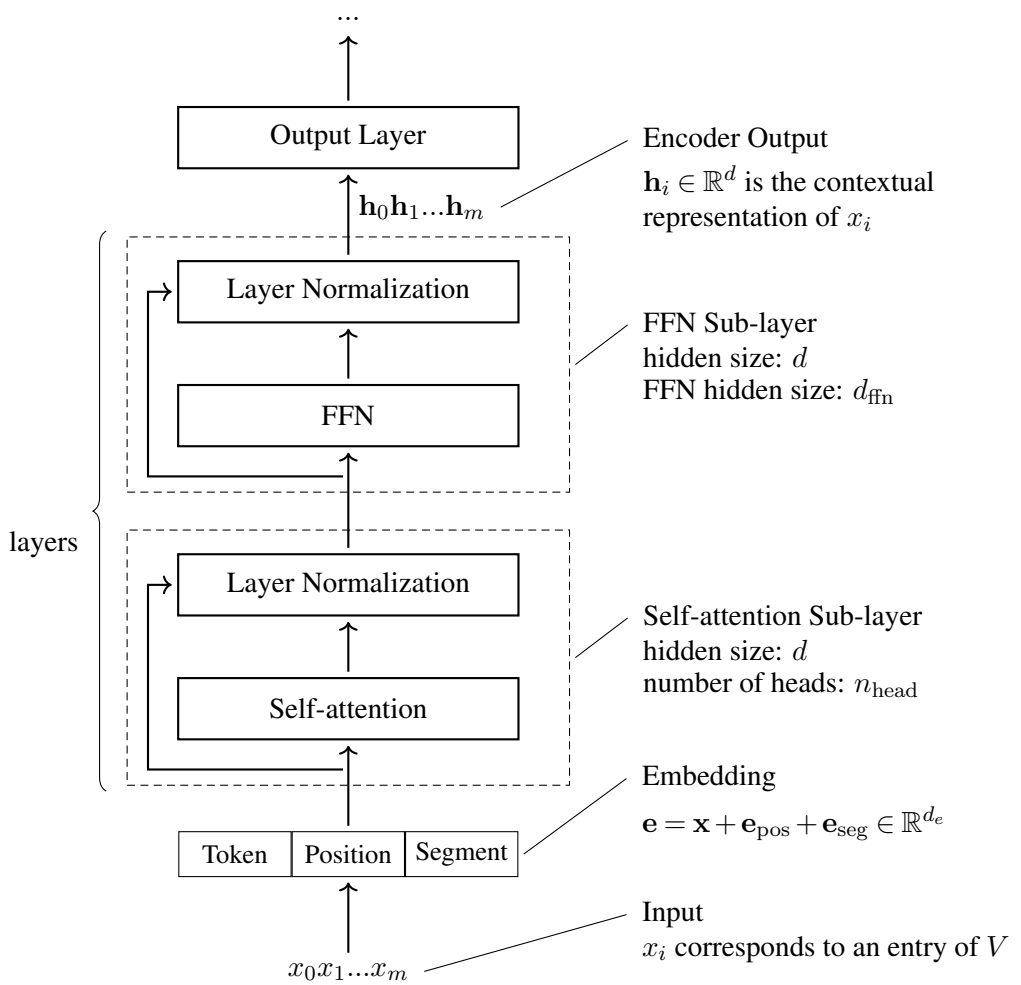


Figure 7.6: The model architecture of BERT (Transformer encoder). The input tokens are first represented as embeddings, each of which is the sum of the corresponding token embedding, positional embedding and segment embedding. Then, the embedding sequence is processed by a stack of Transformer layers. Each layer in this stack includes a self-attention sub-layer and a FFN sub-layer. The output of the BERT model is a sequence of vectors produced by the final Transformer layer.

Different settings of these hyper-parameters lead to different model sizes. There are two widely-used BERT models.

- $\text{BERT}_{\text{base}}$: $d = 768$, $L = 12$, $n_{\text{head}} = 12$, total number of parameters $= 110$M.
- $\text{BERT}_{\text{large}}$: $d = 1,024$, $L = 24$, $n_{\text{head}} = 16$, total number of parameters $= 340$M.

Training BERT models follows the standard training process of Transformers. Training larger models such as $\text{BERT}_{\text{large}}$ requires more training effort and time. This is a common problem for pre-training, especially when a model is trained on a very large amount of data. In practice, there are often considerations of training efficiency. For example, a practice is to first train a BERT model on relatively short sequences for a large number of training steps, and

then continue training it on full-length sequences for the remaining training steps.

7.3.2 More Training and Larger Models

BERT is a milestone model in NLP, sparking many subsequent efforts to improve it. One direction is to scale up the model itself, including increasing training data and developing larger models.

RoBERTa, an extension of the standard BERT model, is an example of such efforts [Liu et al., 2019]. It introduces two major improvements. First, simply using more training data and more compute can improve BERT models without need of changing the model architectures. Second, removing the NSP loss does not decrease the performance on downstream tasks if the training is scaled up. These findings suggest exploring a general direction of pre-training: we can continue to improve pre-training by scaling it up on simple pre-training tasks.

A second approach to improving BERT models is to increase the number of model parameters. For example, in He et al. [2021]'s work, a 1.5 billion-parameter BERT-like model is built by increasing both the model depth and hidden size. However, scaling up BERT and various other pre-trained models introduces new challenges in training, for example, training very large models often becomes unstable and difficult to converge. This makes the problem more complicated, and requires careful consideration of various aspects, including model architecture, parallel computation, parameter initialization, and so on. In another example, Shoeybi et al. [2019] successfully trained a 3.9 billion-parameter BERT-like model, where hundreds of GPUs were used to manage the increased computational demands.

7.3.3 More Efficient Models

Compared to its predecessors, BERT is a relatively large model for the time it was proposed. This increase in model size results in larger memory requirements and a consequent slowdown in system performance. Developing smaller and faster BERT models is part of the broader challenge of building efficient Transformers, which has been extensively discussed in Chapter 6. However, a deeper discussion of this general topic is beyond the scope of our current discussion. Here we instead consider a few efficient variants of BERT.

Several threads of research are of interest to NLP researchers in developing efficient BERT models. First, work on knowledge distillation, such as training student models with the output of well-trained teacher models, shows that smaller BERT models can be obtained by transferring knowledge from larger BERT models. Given that BERT models are multi-layer networks with several different types of layers, knowledge distillation can be applied at different levels of representation. For example, beyond distilling knowledge from the output layers, it is also possible to incorporate training loss that measures the difference in output of hidden layers between teacher models and student models [Sun et al., 2020; Jiao et al., 2020]. Indeed, knowledge distillation has been one of the most widely-used techniques for learning small pre-trained models.

Second, conventional model compression methods can be directly applied to compress BERT models. One common approach is to use general-purpose pruning methods to prune the Transformer encoding networks [Gale et al., 2019]. This generally involves removing entire

layers [Fan et al., 2019] or a certain percentage of parameters in the networks [Sanh et al., 2020; Chen et al., 2020]. Pruning is also applicable to multi-head attention models. For example, Michel et al. [2019] show that removing some of the heads does not significantly decrease the performance of BERT models, but speeds up the inference of these models. Another approach to compressing BERT models is quantization [Shen et al., 2020]. By representing model parameters as low-precision numbers, the models can be greatly compressed. While this method is not specific to BERT models, it proves effective for large Transformer-based architectures.

Third, considering that BERT models are relatively deep and large networks, another thread of research uses dynamic networks to adapt these models for efficient inference. An idea in this paradigm is to dynamically choose the layers for processing a token, for example, in depth-adaptive models we exit at some optimal depth and thus skip the rest of the layers in the layer stack [Xin et al., 2020; Zhou et al., 2020]. Similarly, we can develop length-adaptive models in which the length of the input sequence is dynamically adjusted. For example, we can skip some of the tokens in the input sequence so that the model can reduce computational load on less important tokens, enhancing overall efficiency.

Fourth, it is also possible to share parameters across layers to reduce the size of BERT models. A simple way to do this is to share the parameters of a whole Transformer layer across the layer stack [Dehghani et al., 2018; Lan et al., 2020]. In addition to the reduced number of parameters, this enables reuse of the same layer in a multi-layer Transformer network, leading to savings of memory footprint at test time.

7.3.4 Multi-lingual Models

The initial BERT model was primarily focused on English. Soon after this model was proposed, it was extended to many languages. One simple way to do this is to develop a separate model for each language. Another approach, which has become more popular in recent work on large language models, is to train multi-lingual models directly on data from all the languages. In response, **multi-lingual BERT** (**mBERT**) models were developed by training them on text from 104 languages [6]. The primary difference from monolingual BERT models is that mBERT models use larger vocabularies to cover tokens from multiple languages. As a result, the representations of tokens from different languages are mapped into the same space, allowing for the sharing of knowledge across languages via this universal representation model.

One important application of multi-lingual pre-trained models is cross-lingual learning. In the cross-lingual setting, we learn a model on tasks in one language, and apply it to the same tasks in another language. In cross-lingual text classification, for example, we fine-tune a multi-lingual pre-trained model on English annotated documents. Then, we use the fine-tuned model to classify Chinese documents.

An improvement to multi-lingual pre-trained models like mBERT is to introduce bilingual data into pre-training. Rather than training solely on monolingual data from multiple languages, bilingual training explicitly models the relationship between tokens in two languages. The

[6] https://github.com/google-research/bert/

resulting model will have innate cross-lingual transfer abilities, and thus can be easily adapted to different languages. Lample and Conneau [2019] propose an approach to pre-training **cross-lingual language models** (**XLMs**). In their work, a cross-lingual language model can be trained in either the causal language modeling or masked language modeling manner. For masked language modeling pre-training, the model is treated as an encoder. The training objective is the same as BERT: we maximize the probabilities of some randomly selected tokens which are either masked, replaced with random tokens, or kept unchanged in the input. If we consider bilingual data in pre-training, we sample a pair of aligned sentences each time. Then, the two sentences are packed together to form a single sequence used for training. For example, consider an English-Chinese sentence pair

鲸鱼 是 哺乳 动物 。 ↔ Whales are mammals .

We can pack them to obtain a sequence, like this

[CLS] 鲸鱼 是 哺乳 动物 。 [SEP] Whales are mammals . [SEP]

We then select a certain percentage of the tokens and replace them with [MASK].

[CLS] [MASK] 是 [MASK] 动物 。 [SEP] Whales [MASK] [MASK] . [SEP]

The goal of pre-training is to maximize the product of the probabilities of the masked tokens given the above sequence. By performing training in this way, the model can learn to represent both the English and Chinese sequences, as well as to capture the correspondences between tokens in the two languages. For example, predicting the Chinese token 鲸鱼 may require the information from the English token *Whales*. Aligning the representations of the two languages essentially transforms the model into a "translation" model. So this training objective is also called **translation language modeling**. Figure 7.7 shows an illustration of this approach.

A benefit of multi-lingual pre-trained models is their inherent capability of handling code-switching. In NLP and linguistics, code-switching refers to switching among languages in a text. For example, the following is a mixed language text containing both Chinese and English:

周末 我们 打算 去 做 hiking ， 你 想 一起 来 吗 ？
(We plan to go hiking this weekend, would you like to join us?)

For multi-lingual pre-trained models, we do not need to identify whether a token is Chinese or English. Instead, every token is just an entry of the shared vocabulary. This can be imagined as creating a "new" language that encompasses all the languages we want to process.

The result of multi-lingual pre-training is influenced by several factors. Given that the model architecture is fixed, one needs to specify the size of the shared vocabulary, the number (or percentage) of samples in each language, the size of the model, and so on. Conneau et al.

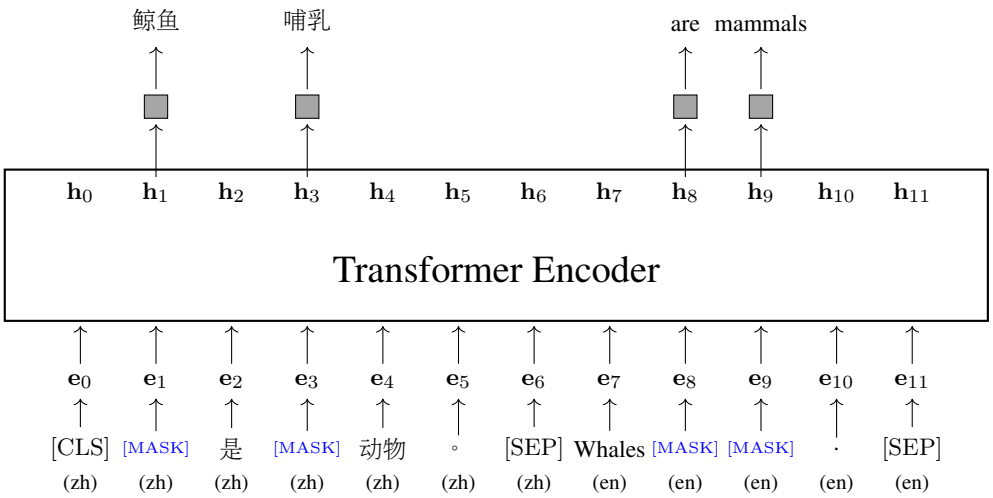


Figure 7.7: An illustration of translation language modeling. For ease of understanding, we present a simple example where all the selected tokens are masked. The model is trained to predict these masked tokens. As the sequence contains tokens in two languages, predicting a token in one language allows access to tokens in the other language, thereby enabling cross-lingual modeling. In Lample and Conneau [2019]'s work, an input embedding (i.e., $\mathbf{e}_i$) is the sum of the token embedding, positional embedding, and language embedding. This requires that each token is assigned with a language label. Thus we can distinguish tokens in different languages. In multi-lingual pre-training, particularly in work using shared vocabularies, specifying the language to which a token belongs is not necessary. The use of language embeddings in turn makes it difficult to handle code-switching. Therefore, we assume here that all token representations are language-independent.

[2020] point out several interesting issues regarding large-scale multi-lingual pre-training for XLM-like models. First, as the number of supported languages increases, a larger model is needed to handle these languages. Second, a larger shared vocabulary is helpful for modeling the increased diversity in languages. Third, low-resource languages more easily benefit from cross-lingual transfer from high-resource languages, particularly when similar high-resource languages are involved in pre-training. However, **interference** may occur if the model is trained for an extended period, meaning the overall performance of the pre-trained model starts decreasing at a certain point during pre-training. Thus, in practical systems, one may need to stop the pre-training early to prevent interference.

7.4 Applying BERT Models

Once a BERT model is pre-trained, it can then be used to solve NLP problems. But BERT models are not immediately ready for performing specific downstream tasks. In general, additional fine-tuning work is required to make them adapt. As a first step, we need a predictor to align the output of the model with the problem of interest. Let $\mathrm{BERT}_{\hat{\theta}}(\cdot)$ be a BERT model with pre-trained parameters $\hat{\theta}$, and $\mathrm{Predict}_{\omega}(\cdot)$ be a prediction network with parameters ω. By

integrating the prediction network with the output of the BERT model, we develop a model to tackle the downstream tasks. This model can be expressed as

$$\mathbf{y} \quad = \quad \mathrm{Predict}_{\omega}(\mathrm{BERT}_{\hat{\theta}}(\mathbf{x})) \tag{7.21}$$

where $\mathbf{x}$ is the input and $\mathbf{y}$ is the output that fits the problem. For example, in classification problems, the model outputs a probability distribution over labels.

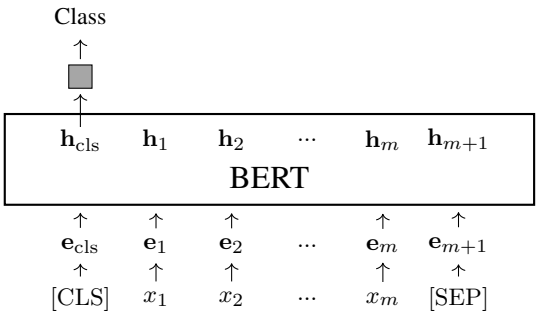
Then, we collect a set of labeled samples $\mathcal{D}$, and fine-tune the model by

$$(\tilde{\omega}, \tilde{\theta}) \quad = \quad \underset{\omega, \hat{\theta}^{+}}{\arg\min} \sum_{(\mathbf{x}, \mathbf{y}_{\mathrm{gold}}) \in \mathcal{D}} \mathrm{Loss}(\mathbf{y}_{\omega, \hat{\theta}^{+}}, \mathbf{y}_{\mathrm{gold}}) \tag{7.22}$$

where $(\mathbf{x}, \mathbf{y}_{\mathrm{gold}})$ represents a tuple of an input and its corresponding output. The notation of this equation seems a bit complicated, but the training/tuning process is standard. We optimize the model by minimizing the loss over the tuning samples. The outcome is the optimized parameters $\tilde{\omega}$ and $\tilde{\theta}$. The optimization starts with the pre-trained parameters $\hat{\theta}$. Here we use $\hat{\theta}^{+}$ to indicate that the parameters are initialized with $\hat{\theta}$, and use $\mathbf{y}_{\omega, \hat{\theta}^{+}}$ to denote the model output computed using the parameters ω and $\hat{\theta}^{+}$.

With the fine-tuned parameters $\tilde{\omega}$ and $\tilde{\theta}$, we can apply the model $\mathrm{Predict}_{\tilde{\omega}}(\mathrm{BERT}_{\tilde{\theta}}(\cdot))$ to new data of the same tasks for which the model was fine-tuned. The form of the downstream tasks determines the input and output formats of the model, as well as the architecture of the prediction network. In the following we list some tasks to which BERT models are generally suited.

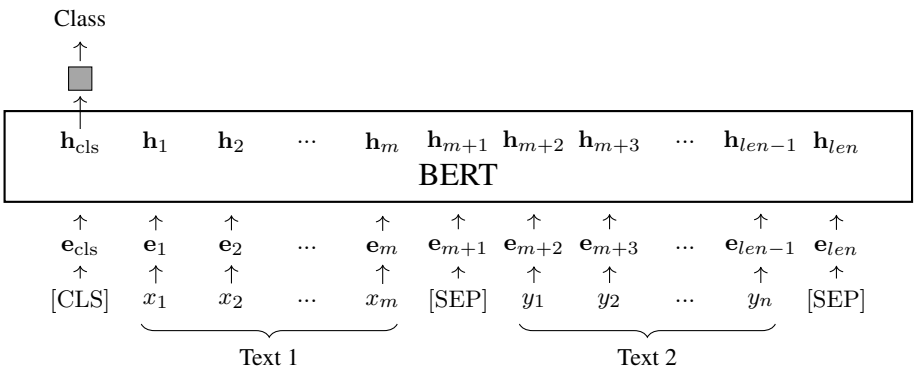
- **Classification** (Single Text). One of the most widely-used applications of BERT models is text classification. In this task, a BERT model receives a sequence of tokens and encodes it as a sequence of vectors. The first output vector $\mathbf{h}_{\mathrm{cls}}$ (or $\mathbf{h}_0$) is typically used as the representation of the entire text. The prediction network takes $\mathbf{h}_{\mathrm{cls}}$ as input to produce a distribution of labels. Let $[\mathrm{CLS}]x_1x_2...x_m$ be an input text. See below for an illustration of BERT-based text classification.



  Here the gray box denotes the prediction network. Many NLP problems can be categorized as text classification tasks, and there have been several text classification benchmarks for evaluating pre-trained models. For example, we can classify texts by their grammatical correctness (grammaticality) or emotional tone (sentiment) [Socher

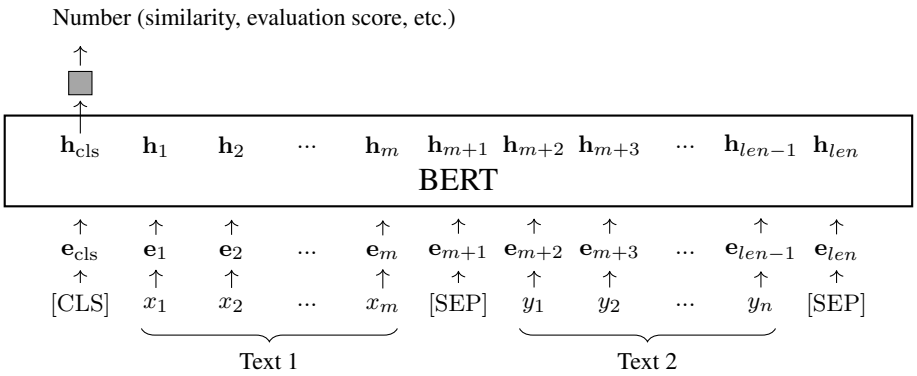
et al., 2013; Warstadt et al., 2019]. Note that the prediction network could be any classification model, such as a deep neural network or a more traditional classification model. The entire model can then be trained or fine-tuned in the manner of a standard classification model. For example, the prediction network can be simply a Softmax layer and the model parameters can be optimized by maximizing the probabilities of the correct labels.

- **Classification** (Pair of Texts). Classification can also be performed on a pair of texts. Suppose we have two texts, $x_1 ... x_m$ and $y_1 ... y_n$. We can concatenate these texts to form a single sequence with a length len. Then, we predict a label for this combined text sequence based on the $\mathbf{h}_{\text{cls}}$ vector, as follows



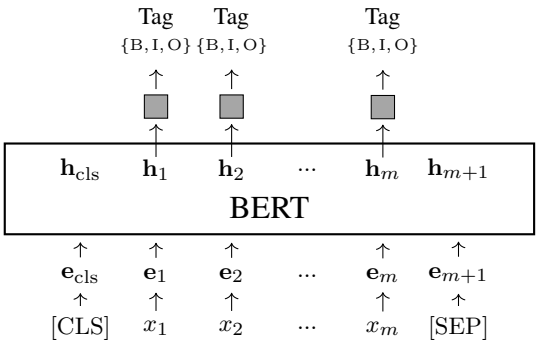
where $len = n + m + 2$. Text pair classification covers several problems, including semantic equivalence judgement (determine whether two texts are semantically equivalent) [Dolan and Brockett, 2005], text entailment judgement (determine whether a hypothesis can be logically inferred or entailed from a premise) [Bentivogli and Giampiccolo, 2011; Williams et al., 2018], grounded commonsense inference (determine whether an event is likely to happen given its context) [Zellers et al., 2018], and question-answering inference (determine whether an answer corresponds to a given question).

- **Regression**. Instead of generating a label distribution, we can have the prediction network output a real-valued score. For example, by adding a Sigmoid layer to the prediction network, the system can be employed to compute the similarity between two given sentences. The architecture is the same as that of BERT-based classification systems, with only the change of the output layer.



For training or fine-tuning, we can minimize the regression loss of the model output as usual.

- **Sequence Labeling**. Sequence labeling is a machine learning approach applicable to a wide range of NLP problems. This approach assigns a label to each token in an input sequence, and some linguistic annotations can then be derived from this sequence of labels. An example of sequence labeling in NLP is part-of-speech (POS) tagging. We label each word in a sentence with its corresponding POS tag. Another example is named entity recognition (NER) in which we label each word with an NER tag, and named entities are identified using these tags. See below for an illustration of the model architecture for NER.



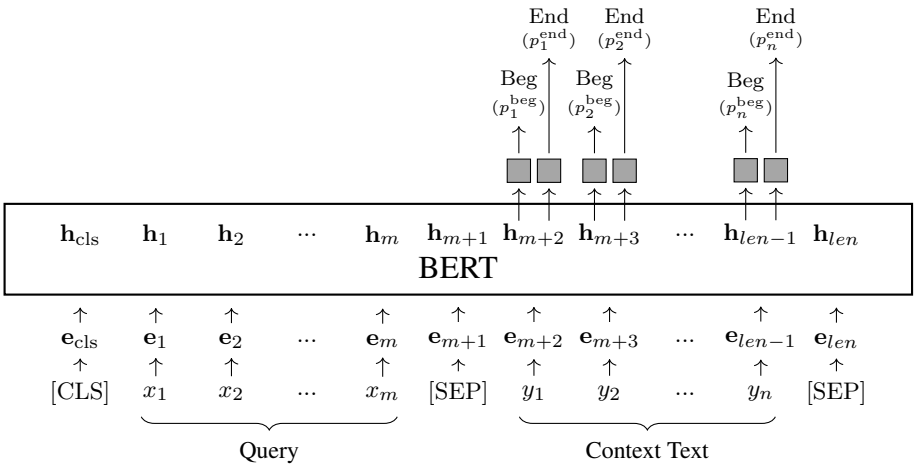
Here $\{\mathrm{B}, \mathrm{I}, \mathrm{O}\}$ is the tag set of NER. For example, B-ORG means the beginning of an organization, I-ORG means the word is inside an organization, and O means the word does not belong to any named entity. This NER model can output a distribution over the tag set at each position, denoted as $\mathbf{p}_i$. The training or fine-tuning of the model can be performed over these distributions $\{\mathbf{p}_1, ..., \mathbf{p}_m\}$. For example, suppose $p_i(\text{tag}_i)$ is the probability of the correct tag at position i. The training loss can be defined to be the negative likelihood

$$\text{Loss} \quad = \quad -\frac{1}{m}\sum_{i=1}^{m} \log p_i(\text{tag}_i) \tag{7.23}$$

Finding the best label sequence given a trained NER model is a well-studied issue in

NLP. This is often achieved via dynamic programming, which, in the context of path finding over a lattice, has linear complexity [Huang, 2009].

- **Span Prediction**. Some NLP tasks require predicting a span in a text. A common example is reading comprehension. In this task, we are given a query $x_1 ... x_m$ and a context text $y_1 ... y_n$. The goal is to identify a continuous span in $y_1 ... y_n$ that best answers the query. This problem can be framed as a sequence labeling-like task in which we predict a label for each y_j to indicate the beginning or ending of the span. Following Seo et al. [2017], we add two networks on top of the BERT output for y_j: one for generating the probability of y_j being the beginning of the span (denoted by p_j^{beg}), and one for generating the probability of y_j being the ending of the span (denoted by p_j^{end}). The resulting model architecture is shown as follows



We pack the query and context text together to obtain the input sequence. The prediction networks are only applied to outputs for the context text, generating the probabilities p_j^{beg} and p_j^{end} at each position. The loss can be computed by summing the negative log likelihoods of the two models across the entire context text.

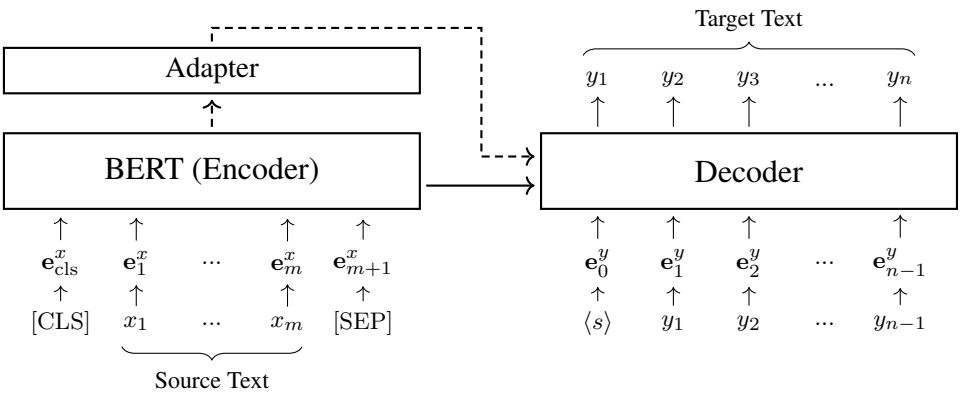
$$\text{Loss} \quad = \quad -\frac{1}{n}\sum_{j=1}^{n}\left(\log p_j^{\text{beg}} + \log p_j^{\text{end}}\right) \tag{7.24}$$

At test time, we search for the best span by

$$(\hat{j}_1, \hat{j}_2) \quad = \quad \underset{1 \le j_1 \le j_2 \le n}{\arg\max} \left(\log p_{j_1}^{\text{beg}} + \log p_{j_2}^{\text{end}}\right) \tag{7.25}$$

- **Encoding for Encoder-Decoder Models**. While our focus in this section has been primarily on language understanding problems, it is worth noting that BERT models can be applied to a broader range of NLP tasks. In fact, BERT models can be used in all the scenarios where we need to encode a piece of text. One application that we have not mentioned is text generation which includes a range of tasks such as machine translation, summarization, question answering, and dialogue generation. These tasks

can be formulated as sequence-to-sequence problems: we use an encoder to represent the source text, and a decoder to generate the corresponding target text. A straightforward method to apply BERT models is to consider them as encoders. Before fine-tuning, we can initialize the parameters of the encoder with those from a pre-trained BERT model. Then, the encoder-decoder model can be fine-tuned on pairs of texts as usual. The following shows the architecture of a neural machine translation system where a BERT model is applied on the source side.



Here $x_1 ... x_m$ denotes the source sequence, $y_1 ... y_n$ denotes the target sequence, $\mathbf{e}^x_1 ... \mathbf{e}^x_m$ denotes the embedding sequence of $x_1 ... x_m$, and $\mathbf{e}^y_1 ... \mathbf{e}^y_n$ denotes the embedding sequence of $y_1 ... y_n$. The adapter, which is optional, maps the output of the BERT model to the form that is better suited to the decoder.

Fine-tuning BERT models is a complicated engineering problem, influenced by many factors, such as the amount of fine-tuning data, the model size, and the optimizer used in fine-tuning. In general, we wish to fine-tune these models sufficiently so that they can perform well in the downstream tasks. However, fine-tuning BERT models for specific tasks may lead to overfitting, which in turn reduces their ability to generalize to other tasks. For example, suppose we have a BERT model that performs well on a particular task. If we then fine-tune it for new tasks, this may decrease its performance on the original task. This problem is related to the **catastrophic forgetting** problem in continual training, where a neural network forgets previously learned information when updated on new samples. In practical applications, a common way to alleviate catastrophic forgetting is to add some old data into fine-tuning and train the model with more diverse data. Also, one may use methods specialized to catastrophic forgetting, such as experience replay [Rolnick et al., 2019] and elastic weight consolidation [Kirkpatrick et al., 2017]. The interested reader can refer to some surveys for more detailed discussions of this issue in continual learning [Parisi et al., 2019; Wang et al., 2023a;b].

7.5 Summary

In this chapter we have discussed the general idea of pre-training in NLP. In particular, we have discussed self-supervised pre-training and its application to encoder-only, decoder-only, and

encoder-decoder architectures. Moreover, we have presented and compared a variety of pre-training tasks for these architectures. As an example, BERT is used to illustrate how sequence models are pre-trained via masked language modeling and applied to different downstream tasks.

Recent years have shown remarkable progress in NLP, led by the large-scale use of self-supervised pre-training. And sweeping advances are being made across many tasks, not only in NLP but also in computer vision and other areas of AI. One idea behind these advances is that a significant amount of knowledge about the world can be learned by simply training these AI systems on huge amounts of unlabeled data. For example, a language model can learn some general knowledge of a language by repeatedly predicting masked words in large-scale text. As a result, this pre-trained language model can serve as a foundation model, which can be easily adapted to address specific downstream NLP tasks. This paradigm shift in NLP has enabled the development of incredibly powerful systems for language understanding, generation, and reasoning [Manning, 2022]. However, it is important to recognize that we are still in the early stages of creating truly intelligent systems, and there is a long way to go. Nevertheless, large-scale pre-training has opened a door to intelligent systems that researchers have long aspired to develop, though several key research areas remain open for exploration, such as learning intelligence efficiently using reasonably small-sized data and acquiring complex reasoning and planning abilities.

Note that this chapter is mostly introductory and cannot cover all aspects of pre-training. For example, there are many methods to fine-tune a pre-trained model, offering different ways to better adapt the model to diverse situations. Moreover, large language models, which are considered one of the most significant achievements in AI in recent years, are skipped in this section. We leave the discussion of these topics to the following chapters.

Bibliography

[Bengio et al., 2006] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.

[Bentivogli and Giampiccolo, 2011] Luisa Bentivogli and Danilo Giampiccolo. Pascal recognizing textual entailment challenge (rte-7) at tac 2011. https://tac.nist.gov/2011/RTE/, 2011.

[Blum and Mitchell, 1998] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100, 1998.

[Brown et al., 1993] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993.

[Brown et al., 2020] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[Bubeck et al., 2023] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[Chen et al., 2020] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. *Advances in neural information processing systems*, 33:15834–15846, 2020.

[Clark et al., 2019] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. In *Proceedings of International Conference on Learning Representations*, 2019.

[Conneau et al., 2020] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Édouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451, 2020.

[Dehghani et al., 2018] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

[Devlin et al., 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert:

Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[Dolan and Brockett, 2005] Bill Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of Third International Workshop on Paraphrasing (IWP2005)*, 2005.

[Dong et al., 2019] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. *Advances in neural information processing systems*, 32, 2019.

[Erhan et al., 2010] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208, 2010.

[Fan et al., 2019] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In *Proceedings of International Conference on Learning Representations*, 2019.

[Gale et al., 2019] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[Han et al., 2021] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Liang Zhang, Wentao Han, Minlie Huang, Qin Jin, Yanyan Lan, Yang Liu, Zhiyuan Liu, Zhiwu Lu, Xipeng Qiu, Ruihua Song, Jie Tang, Ji-Rong Wen, Jinhui Yuan, Wayne Xin Zhao, and Jun Zhu. Pre-trained models: Past, present and future. *AI Open*, 2:225–250, 2021.

[He et al., 2019] Kaiming He, Ross Girshick, and Piotr Dollár. Rethinking imagenet pre-training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4918–4927, 2019.

[He et al., 2021] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention. In *Proceedings of International Conference on Learning Representations*, 2021.

[Huang, 2009] Liang Huang. Dynamic programming-based search algorithms in NLP. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, 2009.

[Jiao et al., 2020] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, 2020.

[Joshi et al., 2020] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the association for computational linguistics*, 8:64–77, 2020.

[Kirkpatrick et al., 2017] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114 (13):3521–3526, 2017.

[Lample and Conneau, 2019] Guillaume Lample and Alexis Conneau. Cross-lingual language model

pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

[Lan et al., 2020] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *Proceedings of International Conference on Learning Representations*, 2020.

[Lewis et al., 2020] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, 2020.

[Liu et al., 2019] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[Manning, 2022] Christopher D Manning. Human language understanding & reasoning. *Daedalus*, 151(2):127–138, 2022.

[Michel et al., 2019] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Advances in neural information processing systems*, 32, 2019.

[Mikolov et al., 2013] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, pages 3111–3119, 2013.

[Parisi et al., 2019] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural networks*, 113:54–71, 2019.

[Pennington et al., 2014] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[Qiu et al., 2020] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897, 2020.

[Radford et al., 2018] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI*, 2018.

[Raffel et al., 2020] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[Rolnick et al., 2019] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[Sanh et al., 2020] Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in Neural Information Processing Systems*, 33:20378–20389, 2020.

[Schmidhuber, 2015] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[Seo et al., 2017] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. In *Proceedings of International Conference on*

Learning Representations, 2017.

[Shen et al., 2020] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821, 2020.

[Shoeybi et al., 2019] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[Socher et al., 2013] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[Song et al., 2019] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. In *International Conference on Machine Learning*, pages 5926–5936. PMLR, 2019.

[Sun et al., 2020] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170, 2020.

[Wang et al., 2023] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *arXiv preprint arXiv:2302.00487*, 2023a.

[Wang et al., 2023] Zhenyi Wang, Enneng Yang, Li Shen, and Heng Huang. A comprehensive survey of forgetting in deep learning beyond continual learning. *arXiv preprint arXiv:2307.09218*, 2023b.

[Warstadt et al., 2019] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 2019.

[Williams et al., 2018] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122, 2018.

[Xin et al., 2020] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating bert inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2246–2251, 2020.

[Yang et al., 2019] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[Yarowsky, 1995] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the 33rd annual meeting of the association for computational linguistics*, pages 189–196, 1995.

[Zellers et al., 2018] Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. Swag: A large-scale adversarial dataset for grounded commonsense inference. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 93–104, 2018.

[Zhou et al., 2020] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei.

Bert loses patience: Fast and robust inference with early exit. *Advances in Neural Information Processing Systems*, 33:18330–18341, 2020.

[Zoph et al., 2020] Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin Dogus Cubuk, and Quoc Le. Rethinking pre-training and self-training. *Advances in neural information processing systems*, 33:3833–3845, 2020.