forms, for simplicity, we describe them using a single model defined as follows:

$$\begin{aligned} \mathbf{o} &= g(x_0, x_1, ..., x_m; \theta) \\ &= g_\theta(x_0, x_1, ..., x_m) \end{aligned} \tag{1.1}$$

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.

1.1.1 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 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

[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.

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.

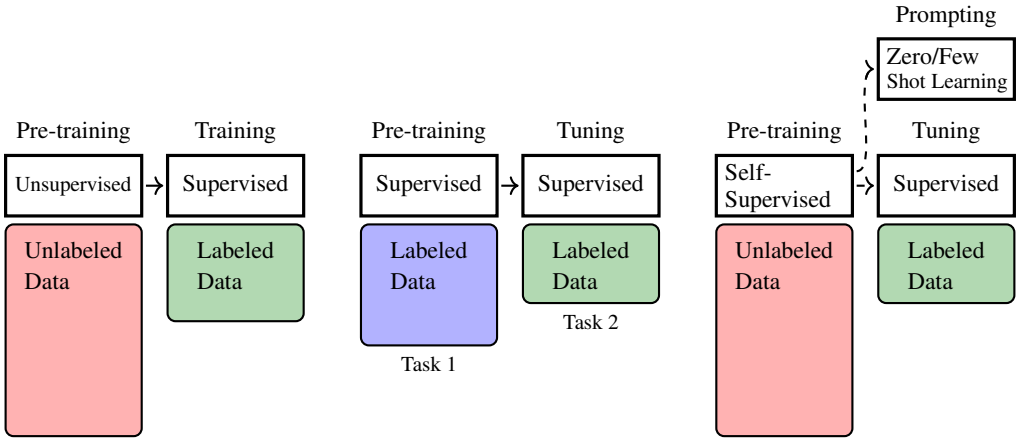
A third approach to pre-training is **self-supervised learning**. In this approach, a neural network is trained using the supervision signals generated by itself, rather than those provided by humans. This is generally done by constructing its own training tasks directly from unlabeled data, such as 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 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 1.1 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.

1.1.2 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.



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

Fig. 1.1: 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.

- **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.

1.1.2.1 Fine-tuning of Pre-trained Models

For sequence encoding pre-training, a common method of adapting pre-trained models is fine-tuning. Let $\text{Encode}_\theta(\cdot)$ denote an encoder with parameters θ, for example, $\text{Encode}_\theta(\cdot)$ 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

$$\mathbf{H} \quad = \quad \text{Encode}_{\hat{\theta}}(\mathbf{x}) \tag{1.2}$$

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,

[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.

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

$$\begin{aligned} \mathrm{Pr}_{\omega,\hat{\theta}}(\cdot|\mathbf{x}) &= \mathrm{Classify}_{\omega}(\mathbf{H}) \\ &= \mathrm{Classify}_{\omega}(\mathrm{Encode}_{\hat{\theta}}(\mathbf{x})) \end{aligned} \tag{1.3}$$

Here $\mathrm{Pr}_{\omega,\hat{\theta}}(\cdot|\mathbf{x})$ 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!

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

$$F_{\tilde{\omega},\tilde{\theta}}(\mathbf{x}_{\mathrm{new}}) = \begin{bmatrix} \mathrm{Pr}(\mathrm{positive}|\mathbf{x}_{\mathrm{new}}) & \mathrm{Pr}(\mathrm{negative}|\mathbf{x}_{\mathrm{new}}) & \mathrm{Pr}(\mathrm{neutral}|\mathbf{x}_{\mathrm{new}}) \end{bmatrix} \tag{1.4}$$

And we select the label of the entry with the maximum value as output. In this example it is $\mathrm{positive}$.

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 adaption 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 1.4.

1.1.2.2 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

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

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.

> 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 3. 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 4.

1.2 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.

1.2.1 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

$$\begin{aligned} \text{Loss}_\theta(x_0, ..., x_m) &= \sum_{i=0}^{m-1} \mathcal{L}(\mathbf{p}_{i+1}^\theta, \mathbf{p}_{i+1}^{\text{gold}}) \\ &= \sum_{i=0}^{m-1} \text{LogCrossEntropy}(\mathbf{p}_{i+1}^\theta, \mathbf{p}_{i+1}^{\text{gold}}) \end{aligned} \tag{1.5}$$

where $\text{LogCrossEntropy}(\cdot)$ is the log-scale cross-entropy, and $\mathbf{p}_{i+1}^{\text{gold}}$ 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 $\mathcal{D}$

$$\hat{\theta} = \arg\min_\theta \sum_{\mathbf{x} \in \mathcal{D}} \text{Loss}_\theta(\mathbf{x}) \tag{1.6}$$

Note that this objective is mathematically equivalent to maximum likelihood estimation, and can be re-expressed as

$$\begin{aligned} \hat{\theta} &= \arg\max_\theta \sum_{\mathbf{x} \in \mathcal{D}} \log \Pr_\theta(\mathbf{x}) \\ &= \arg\max_\theta \sum_{\mathbf{x} \in \mathcal{D}} \sum_{i=0}^{i-1} \log \Pr_\theta(x_{i+1} | x_0, ..., x_i) \end{aligned} \tag{1.7}$$

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.

1.2.2 Encoder-only Pre-training

As defined in Section 1.1.2.1, 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 1.2 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

$$\begin{bmatrix} \mathbf{p}_1^{\mathbf{W},\theta} \\ \vdots \\ \mathbf{p}_m^{\mathbf{W},\theta} \end{bmatrix} = \text{Softmax}_{\mathbf{W}}(\text{Encoder}_\theta(\mathbf{x})) \tag{1.9}$$

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

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_0 \\ \vdots \\ \mathbf{h}_m \end{bmatrix} \tag{1.8}$$