used in recent LLMs such as PaLM and Llama. xPos, built on RoPE, enhances the translation invariance and length extrapolation of Transformers by adding a special exponential decay to each dimension of the rotation degree vector, stabilizing training over increased distances. ALiBi improves Transformer extrapolation by biasing attention scores with a distance-based penalty between keys and queries without trainable parameters. It has demonstrated superior extrapolation performance and training stability compared to other position embedding methods, including sinusoidal PE, RoPE, and T5 bias.

#### 2.3.10.5 Attention Mechanism

The original Transformer utilizes full attention, conducting attention pairwise and considering all token pairs in a sequence. It employs scaled dot-product attention and multi-head attention, where queries, keys, and values are projected differently in each head, with the concatenated output of each head forming the final output. Sparse attention addresses the quadratic computational complexity challenge of full attention, especially with long sequences.

**! Practical Tips**

Efficient Transformer variants, like locally banded sparse attention (e.g., Factorized Attention in GPT-3), allow each query to attend to a subset of tokens based on positions, reducing complexity. Multi-query attention, where different heads share the same linear transformation matrices on keys and values, offers computational savings with minimal impact on model quality. Models such as PaLM and StarCoder utilize multi-query attention. FlashAttention optimizes the speed and memory consumption of attention modules on GPUs without compromising model quality. It organizes input into blocks and introduces recomputation to utilize fast memory (SRAM) on GPUs efficiently. Integrated into platforms such as PyTorch, DeepSpeed, and Megatron-LM, FlashAttention optimizes attention modules from an IO-aware perspective. For optimal generalization and training stability, pre-RMSNorm is recommended for layer normalization, with SwiGLU or GeGLU as the activation function. It is advised not to use layer normalization immediately after embedding layers to avoid performance degradation. Some methods, such as Realformer and Predictive Attention Transformer, reuse attention distributions from previous blocks to guide the current block, creating more direct paths through the network. Transparent Attention eases optimization using a weighted sum of encoder representations from all layers in cross-attention modules. Adaptive Computation Time (ACT) has been introduced to tailor computation time based on input difficulty, leading to strategies such as Universal Transformer and Conditional Computation Transformer, which either refine representations iteratively or utilize gating mechanisms to optimize computational resources.

Table 2.1: In our network configurations, *Sublayer* refers to either a feed-forward neural network (FFN) or a self-attention module within a Transformer layer. The symbol $d$ represents the size of the hidden states in the network. The position embedding at a specific position $i$ is denoted by $pi$. In the attention mechanism, $A_{ij}$ signifies the attention score computed between a given query and its corresponding key. The difference in positions between the query and the key is represented by $r_{i-j}$, a learnable scalar value. Finally, the term $R_{\theta,t}$ refers to a rotary matrix, which rotates by an angle determined by multiplying $t$ by $\theta$.

| **Configuration** | **Method** | **Equation** |
|---|---|---|
| Normalization | Post Norm [1] | $\text{Norm}(\mathbf{x} + \text{Sublayer}(\mathbf{x}))$ |
| position | Pre Norm [2] | $\mathbf{x} + \text{Sublayer}(\text{Norm}(\mathbf{x}))$ |
| | Sandwich Norm [3] | $\mathbf{x} + \text{Norm}(\text{Sublayer}(\text{Norm}(\mathbf{x})))$ |
| Normalization | LayerNorm [4] | $\frac{\mathbf{x}-\mu}{\sqrt{\sigma}} \cdot \gamma + \beta,\ \mu = \frac{1}{d}\sum_{i=1}^{d} \mathbf{x}_i,\ \sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(\mathbf{x}_i - \mu)^2}$ |
| method | RMSNorm [5] | $\frac{\mathbf{x}}{\text{RMS}(\mathbf{x})} \cdot \gamma,\ \text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} \mathbf{x}_i^2}$ |
| | DeepNorm [6] | $\text{LayerNorm}(\alpha \cdot \mathbf{x} + \text{Sublayer}(\mathbf{x}))$ |
| Activation | ReLU [7] | $\text{ReLU}(\mathbf{x}) = \max(0, \mathbf{x})$ |
| function | GeLU [8] | $\text{GeLU}(\mathbf{x}) = 0.5\mathbf{x} \bigotimes \left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right)$ |
| | Swish [9] | $f(x) = x \cdot \frac{1}{1+e^{-x}}$ |
| | SwiGLU [10] | $f(x) = x \odot \sigma(Wx + b)$ |
| | GeGLU [10] | Similar to SwiGLU with GeLU |
| Positional | Absolute [1] | $x_i = x_i + p_i$ |
| embeddings | Relative [11] | $A_{ij} = W_q x_i x_j^T W_k + r_{i-j}$ |
| | RoPE [12] | $A_{ij} = W_q x_i R_{\theta,i-j} x_j^T W_k$ |
| | Alibi [13] | $A_{ij} = W_q x_i x_j^T W_k - m(i-j)$ |

Key: [1] (Vaswani et al., 2017), [2] (Radford et al., 2019), [3] (Ding et al., 2021), [4] (Ba et al., 2016), [5] (Zhang and Sennrich, 2019), [6] (Wang et al., 2022), [7] (Nair and Hinton, 2010), [8] (Wang et al., 2019), [9] (Ramachandran et al., 2017), [10] (Shazeer, 2020), [11] (Raffel et al., 2020), [12] (Su et al., 2021), [13] (Press et al., 2021)

#### 2.3.10.6 Structural Modifications

To address the computational demands of the Transformer, various high-level modifications have been proposed. The *Lite Transformer* introduces a two-branch structure, combining attention for long-range contexts and convolution for local dependencies, making it suitable for mobile devices. Meanwhile, *Funnel Transformer* and *DeLighT* introduce funnel-like encoder architectures and replace standard Transformer blocks with specialized modules, respectively, aiming to achieve efficiency in terms of FLOPs, memory, and model size. Transformers face challenges in handling long sequences due to their quadratic complexity. Divide-and-conquer strategies, such as recurrent and hierarchical Transformers, have emerged to address this issue. *Recurrent Transformers* utilize cache memory to store historical information, with techniques such as *Transformer-XL* extending context lengths. At the same time, hierarchical Transformers break down inputs into smaller pieces, first processing low-level features and then aggregating them for higher-level processing, aiding in handling long inputs and generating richer representations.

## 2.4 Data

Thus far, in this chapter, we have primarily discussed the technical concepts behind LLMs. In addition to the architectural elements of the models themselves, the data used to train them are equally essential to understanding how they work. This section will provide a view of the types of training data commonly utilized and their effects on the capabilities of LLMs.

### 2.4.1 Language Model Pre-Training Datasets

Transfer learning has dominated all areas of NLP since 2018. In that year, three significant language models were released: ULMFiT, followed by GPT and BERT. Each of these models varied substantially in their architectures, but they all shared a common theme: using only a self-supervised language modeling objective for pre-training and then fine-tuning on task-specific labeled training data. This approach can leverage massive bodies of text for general language understanding without requiring the data to be labeled, which is highly beneficial since labeled data are often difficult to obtain. This section describes the most commonly used data sources for language model pre-training.

The objective during pre-training is to condition the LLM with general language understanding and world knowledge. As such, the selected training data should cover a broad range of topics and use an extensive vocabulary while also capturing a representative distribution of the patterns found in written language. In addition, of course, it also needs to be available in vast quantities. Effective sources include the following:

- **Web-scraping:** Web pages are collected in an automated fashion by following the links within a given page, then following the links in all of those pages, etc. This type of data offers an extensive range of language, but its quality can be suspect. The internet contains slang, typos, and other non-standard language that can increase the robustness of a model. However, by the same token, much of the text may be indecipherable or counterfactual, leading to detrimental effects if not cleaned adequately. The Common Crawl data is the most notable publicly available web scrape.
- **Wikipedia:** Training on Wikipedia data has several benefits. First, it provides a wealth of factual information. It is generally well edited and consistently formatted, making it less prone to the data quality issues of the wider web. As a

bonus, Wikipedia has articles in many languages, allowing for expansion beyond English.

- **Books:** Novels are an excellent narrative source about how humans think and interact with each other and their environments. This type of language is not found in a knowledge base such as Wikipedia, which contains only third-person accounts of events. Most books are also great at modeling long-term dependencies. The obvious downside is that much of the information in story books is fictional.
- **Code:** As generative models have become increasingly powerful, code generation has become a popular application. Data from GitHub and StackExchange are frequently used to train models capable of producing code. Interestingly, training on code may also enhance LLM capabilities on other logical reasoning tasks (Fu and Khot, 2022).

Early Transformer models were trained on a scale at which it was typical to choose one or two of the data sources described above. At the scale of modern LLMs, it is now more common to combine all of these (and more) to realize the unique benefits that each can provide. The Pile (Gao et al., 2020) introduced a corpus spanning 22 sources, such as legal and medical texts, academic research papers, and code from GitHub. They demonstrated that these sources improved downstream performance over models trained on less diverse corpora such as Common Crawl. Taking this idea further, the ROOTS corpus (Laurençon et al., 2023) incorporates 46 natural and 13 programming languages from hundreds of sources.

Table 2.2: Descriptions of various corpora widely adopted for pre-training LLMs.

| **Corpus** | **Source** |
|---|---|
| BookCorpus | Books |
| Wikitext103 | Wikipedia |
| Common Crawl | Internet |
| OpenWebText | Internet |
| The Pile | Internet, Academic Research, Books, Dialog, Code |
| ROOTS | High and Low Resource Languages, Internet, Code |

### 2.4.1.1 Multilingual and Parallel Corpora

Many LLMs are trained exclusively or primarily in a single language, but models that can interpret and translate between many different languages require data spanning all of the desired languages. These data fall broadly into two categories:

- In a parallel corpus, each text example has a corresponding translation in a second language. These language pairs are then used with a training objective wherein one language is the input and the other is the target. The model predictions are then scored based on how closely they match the target.

- A multilingual corpus contains data in multiple languages without any explicit translation between languages. These corpora are useful for language modeling objectives, not the machine translation objective used with parallel corpora.

In recent years, modern LLMs have reached a scale that allows them to perform well on translation tasks in a few-shot setting without specific training on parallel data (Workshop et al., 2023). Translation capabilities emerge from the model's joint conditioning on multiple languages rather than learning from explicit language pairs.

### 2.4.2 Data Pre-Processing

Since the corpora used for pre-training are far too large to be manually reviewed, various methods exist to filter out data that might hinder the model's performance or cause unintended effects. Any text that falls too far outside the language distribution, as well as text that is offensive or contains sensitive personal information, should be removed.

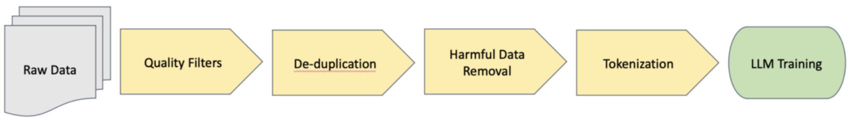

Fig. 2.7: A general sequence of steps to prepare a large corpus for use in LLM pre-training.

#### 2.4.2.1 Low-Quality Data

As shown in Fig. 2.7, the first pre-processing stage is focused on overall data quality. Since the raw corpora tend to be substantially large, one can usually afford to remove sizable portions of data that show any signs of being unsuitable for training. As such, this stage of pre-processing can be somewhat coarse-grained.

One typical quality issue that may arise in large corpora is languages that fall outside the model's intended use. If the model is being trained specifically for Spanish applications, for instance, then the presence of any languages other than Spanish will decrease training efficiency. These data can be filtered out with a language classification model or a more rule-based approach.

Another helpful pre-processing step is statistical filtering based on unusual text patterns. Some examples include a high frequency of strings much longer than a typical word, a high density of punctuation characters, and a prevalence of very long or short sentences. Any of these patterns indicate that the document or set of documents will be less generalizable and, therefore, less valuable for the model during training.

#### 2.4.2.2 Duplicate Data

There has been considerable discussion about the effects of duplicate training data. Hernandez et al. (2022) observed several potential negative consequences from training on repeated data. As a counterpoint, analysis by Biderman et al. (2023) indicated that training on duplicated data neither benefits nor hurts the model. At any rate, training on duplicated data appears to be a suboptimal use of compute cycles, even in the best-case scenario. It is, therefore, a standard practice to remove repeated text wherever possible during the pre-processing stage.

#### 2.4.2.3 Harmful Data

The above issues are primarily about optimizing training cycles using only the most applicable data. A further concern is that certain information may be undesirable for the model to capture. For example, it could be problematic if real people's names and email addresses appear in LLM-generated outputs after being scraped from the web. Toxicity and bias present in the training data are also significant areas of concern. Combating these elements is a more complex matter that will be discussed in later chapters, but removing offensive language in the pre-processing stage is worthwhile wherever possible.

#### 2.4.2.4 Text Normalization

Some data may suffer from less severe issues that need to be cleaned up but don't warrant complete removal of the text. For example, data scraped from the web will naturally contain remnants of HTML tags that should be stripped out. Another common step is Unicode normalization, which addresses the fact that equivalent strings can be represented with multiple possible encodings. Rather than forcing the model to try to learn these equivalencies, it is usually preferable to standardize the representation as much as possible using one of several methods. Similarly, if desired, one can optionally choose to lowercase all text so that the model will not treat capital letters as distinct characters.

#### 2.4.2.5 Tokenization

Upon completion of pre-processing, the data are then used to train a tokenizer such as those described in Sect. 2.3.3. Naturally, this must be done before the actual LLM can be trained since the tokenized output is the input to the model. A frequent practice is to use an existing tokenizer rather than training one from scratch, but this is only an option if similar data sources are used. First and foremost, the tokenization must reflect the languages (or programming languages) included in the training data. Additionally, conversational data might gravitate toward shorthand tokens such as "thx" or "omg", while the academic literature might have a rather different distribution of tokens representing technical terminology.

The data are fed through the tokenizer in chunks of text, each of which is mapped to a sequence of tokens. For efficiency, the tokens are represented as vectors of integers with length $l$ given by the number of subwords. The first layer of the model, also called the embedding layer, has dimensions $nxm$, where $n$ corresponds to the total number of tokens learned by the tokenizer and $m$ is a predetermined embedding size. Thus, the tokenized output is a list of index lookups to retrieve vectors of size $m$ for every token identified in the original input. The text has now been converted into a $lxm$ matrix of floating point values that can be passed through the model to initiate the learning process.

### 2.4.3 Effects of Data on LLMs

As discussed previously, many data sources are available for training LLMs. The results produced by Gopher Rae et al. (2022) demonstrated that varying the percentages of data from each source had notable effects on the overall performance of the LLM for an assortment of downstream tasks. In general, data diversity consistently results in better performance across many tasks; however, it is also essential to consider the intended applications of the model. In building a chatbot, one would likely want a substantial portion of the training data to be conversational. Conversely, unless the chatbot dispenses legal advice, including many legal documents would not be sensible.

The amount of data seen by the model during pre-training has a substantial effect. This became abundantly clear with the release of Chinchilla Hoffmann et al. (2022), which demonstrated that previous LLMs had been undertrained. In pursuing the powerful capabilities that emerge with increasing model size, the effects of data size have been miscalculated. Through empirical trials, the Chinchilla researchers sought to establish a formula for determining the optimal number of parameters *and* training tokens for a given compute budget. They found that model size and data size should increase roughly in proportion, a stark contrast to previous work that emphasized the increase in parameters. This was a significant result, showing that highly capable LLMs could be smaller than previously thought. Following these guidelines, the pre-training budget is used more efficiently, and fine-tuning and inference are less expensive.

### 2.4.4 Task-Specific Datasets

For research purposes, NLP “tasks” are often used as a general measure to approximate how well a given model will perform in various real-world settings. Most task-specific datasets are carefully curated and labeled for supervised training and evaluation. As a result, they tend to be much smaller than the very large unlabeled datasets used for LLM pre-training.

Task-specific datasets are generally pre-split into train and test sets to ensure that all researchers train and test on the same examples. Evaluating the performance on these standardized datasets allows direct comparisons between different architectures and training strategies. Importantly, LLMs can often achieve favorable evaluation metrics on a test set without seeing examples from the corresponding training data; this is called zero-shot learning.

## 2.5 Pre-trained LLM Design Choices

This section explores the multifaceted design elements that set apart various LLMs (Zhao et al., 2023). Specifically, we will discuss the nuances of pre-training tasks, delve into different pre-training objectives, examine the intricacies of Transformer architectural choices, and shed light on various decoding strategies.

### 2.5.1 Pre-Training Methods

Understanding the diverse methodologies for pre-training is critical for effectively deploying language models in various domains. Each method has benefits and challenges and suits particular tasks and data types. This section will explore five main pre-training methods, providing a clear overview of how each works, where it is used, and its pros and cons (Kalyan et al., 2021).

#### 2.5.1.1 Pre-training from Scratch

Pre-training from scratch (PTS) involves training Transformer models from the ground on extensive volumes of unlabeled text. This foundational method is crucial for initializing Transformer-based pre-trained language models, which typically comprise an embedding layer followed by multiple Transformer layers. PTS is beneficial because it does not rely on prior knowledge, making it a versatile starting point for various applications. However, this approach requires substantial computational resources and time, especially when dealing with large models and datasets. Models like BERT, RoBERTa, ELECTRA, and T5 are pre-trained from scratch on large volumes of unlabeled text.

#### 2.5.1.2 Continual Pre-training

Continual pre-training (CPT) is a subsequent step following PTS, where the model undergoes further training on a domain-specific corpus. This method is helpful for tasks requiring specialized knowledge, enhancing the model's performance in specific domains. For instance, BioBERT is a variant of BERT that has undergone CPT on biomedical texts, making it adept at tasks related to the biomedical and clinical domains. The drawback of CPT is that it might lead the model to overfit the domain-specific corpus, potentially losing its generalizability.

#### 2.5.1.3 Simultaneous Pre-training

Simultaneous pre-training (SPT) is a method in which models are simultaneously pre-trained on a combination of domain-specific and general-domain corpora. This approach aims to strike a balance, allowing the model to acquire general and domain-specific knowledge concurrently. An example of SPT is ClinicalBERT, which is pre-trained on a mixed corpus of clinical notes and general-domain text. While SPT offers a balanced knowledge base, the challenge lies in effectively selecting and combining corpora to avoid bias toward either domain.

#### 2.5.1.4 Task Adaptive Pre-training

Task adaptive pre-training (TAPT) is a technique for pre-training on a small, task-related corpus. This method is less resource intensive than other methods and is particularly useful when the available data for a specific task are limited. TAPT can complement other pre-training approaches, as it can further refine models that have undergone PTS or CPT, enhancing their performance on specific tasks. However, the effectiveness of TAPT relies heavily on the relevance and quality of the task-related corpus used for pre-taining.

#### 2.5.1.5 Knowledge Inherited Pre-training

Knowledge inherited pre-training (KIPT) is a novel method that utilizes self-supervised learning and inherits knowledge from existing pre-trained models. This approach is inspired by the human learning process, which involves learning from knowledgeable individuals in addition to self-learning. KIPT is efficient because it reduces the time and resources required for pre-training from scratch. However, the success of KIPT depends on the quality and relevance of the knowledge inherited from existing models, and it might not always be straightforward to combine or transfer this knowledge effectively.

### 2.5.2 Pre-training Tasks

Supervised learning has been pivotal in AI advancement, necessitating extensive human-annotated data for practical model training. While proficient in specific tasks, these models often require substantial amounts of labeled data, making the process costly and time intensive, especially in specialized fields like medicine and law, where such data is scarce. Furthermore, supervised learning models lack generalization capabilities, often learning only from provided data, leading to generalization errors and unintended correlations. Recognizing these limitations, researchers are exploring alternative paradigms such as self-supervised learning (SSL). SSL is a learning paradigm in which labels are automatically generated based on data attributes and the definition of pre-training tasks. It helps models learn universal knowledge through pseudo-supervision provided by pre-training tasks. The primary objectives of SSL are to learn universal language representations and improve generalization ability by utilizing a large amount of freely available unlabeled data.

The loss function for SSL is given by:

$$\mathcal{L}_{\text{SSL}} = \lambda_1 \mathcal{L}_{\text{PT1}} + \lambda_2 \mathcal{L}_{\text{PT2}} + \ldots + \lambda_m \mathcal{L}_{\text{PTm}} \tag{2.20}$$

where:

- $\mathcal{L}_{\text{SSL}}$ is the total loss function for SSL.
- $\mathcal{L}_{\text{PT1}}, \mathcal{L}_{\text{PT2}}, \ldots, \mathcal{L}_{\text{PTm}}$ are the loss functions associated with each pre-training task.
- $\lambda_1, \lambda_2, \ldots, \lambda_m$ are the weights assigned to each pre-training task's loss, controlling their contribution to the total loss.

Numerous self-supervised pre-training tasks have been established to train various LLMs (Kalyan et al., 2021). The following section will explore some of the prevalent pre-training tasks employed in LLMs.

#### 2.5.2.1 Causal Language Model

Causal language modeling (CLM) is utilized for predicting the next word in a sequence based on the context, which can be either left-to-right or right-to-left. For a given sequence $x = \{x_1, x_2, x_3, \ldots, x_{|x|}\}$, where $|x|$ represents the number of tokens in the sequence, the loss function for CLM is defined as:

$$\mathcal{L}^{(x)}_{\text{CLM}} = -\frac{1}{|x|}\sum_{i=1}^{|x|} \log P(x_i|x_{<i}) \tag{2.21}$$

where $x_{<i}$ represents the tokens preceding $x_i$ in the sequence.

#### 2.5.2.2 Masked Language Model

Masked language modeling (MLM) is used in the pre-training phase, where selected tokens are masked in the input sequence, and the model is trained to predict these masked tokens. Let $x_{\backslash M_x}$ represent the masked version of $x$, and $M_x$ represent the set of masked token positions in $x$. The loss function for MLM is defined as:

$$\mathcal{L}^{(x)}_{\text{MLM}} = -\frac{1}{|M_x|}\sum_{i\in M_x} \log P(x_i/x_{\backslash M_x}) \tag{2.22}$$

The model aims to minimize this loss by learning to predict the masked tokens accurately, thereby gaining a deeper understanding of the language structure. BERT, a prominent model in natural language processing, employs MLM as a pre-training task, selecting tokens to be masked with a probability of 0.15.

#### 2.5.2.3 Replaced Token Detection

Replaced token detection (RTD) mitigates the drawbacks of MLM by enhancing the training signals and minimizing the discrepancy between the pre-training and fine-tuning phases. Unlike MLM, which uses special mask tokens for corruption, RTD corrupts sentences with tokens generated by a model pre-trained with the MLM objective. This approach transforms the task into a binary classification at the token level, where each token is classified as either replaced or not. The procedure involves two steps: first, training a generator model with the MLM objective, and second, training a discriminator model (initialized from the generator) with the RTD objective. The loss function for RTD is expressed as:

$$\mathcal{L}^{(x)}_{\text{RTD}} = -\frac{1}{|\widehat{x}|}\sum_{i=1}^{|\widehat{x}|} \log P(d/\widehat{x}_i) \tag{2.23}$$

where $d \in \{0, 1\}$ denotes whether a token is replaced (1) or not (0), $\widehat{x}$ is the corrupted sentence, and $P(d/\widehat{x}_i)$ represents the probability of a token being replaced or not.

#### 2.5.2.4 Shuffled Token Detection

Shuffled token detection (STD) is designed to improve the model's understanding of coherent sentence structures, ultimately enhancing its performance across various tasks. In this task, tokens within a sequence are shuffled with a probability of 0.15. The loss function associated with STD is given by:

$$\mathcal{L}_{\text{STD}}^{(x)} = -\frac{1}{|\widehat{x}|}\sum_{i=1}^{|\widehat{x}|} \log P(d/\widehat{x}_i) \tag{2.24}$$

In this equation, $d \in \{0, 1\}$ denotes whether a token is replaced (1) or not (0), and $\widehat{x}$ is the corrupted sentence. The model aims to minimize this loss by learning to identify and comprehend the shuffled tokens within the sequence context effectively.

#### 2.5.2.5 Random Token Substitution

Random token substitution (RTS) is a method introduced by Liello et al. (2021) for identifying tokens that have been randomly substituted in a sequence. In this technique, 15% of the tokens in a given sequence are randomly replaced with other tokens from the vocabulary. This approach is efficient because it does not require a separate generator model to corrupt the input sequence. The loss function for RTS is articulated as:

$$\mathcal{L}_{\text{RTS}}^{(x)} = -\frac{1}{|\widehat{x}|}\sum_{i=1}^{|\widehat{x}|} \log P(d/\widehat{x}_i) \tag{2.25}$$

where $d \in \{0, 1\}$ signifies whether a token has been randomly substituted (1) or not (0), and $\widehat{x}$ is the sequence obtained by randomly substituting 15% of the tokens in the original sequence $x$.

#### 2.5.2.6 Swapped Language Modeling

Swapped language modeling (SLM) addresses the discrepancy in the MLM pre-training task caused by using a special mask token. This discrepancy occurs between the pre-training and fine-tuning stages. SLM mitigates this by corrupting the input sequence with random tokens selected from the vocabulary with a probability of 0.15. Although SLM is akin to MLM in predicting the corrupted tokens, it differs by replacing tokens with random ones instead of mask tokens. Although SLM and RTS both employ random tokens for corruption, SLM is not as sample-efficient as RTS. This inefficiency arises because SLM involves only 15% of input tokens, whereas RTS engages every token in the input sequence. The loss function for SLM is defined as:

$$\mathcal{L}_{\text{SLM}}^{(x)} = -\frac{1}{|R_x|} \sum_{i \in R_x} \log P(x_i / x_{\backslash R_x}) \tag{2.26}$$

where $R_x$ represents the set of positions of randomly substituted tokens, and $x_{\backslash R_x}$ represents the corrupted version of $x$.

#### 2.5.2.7 Translation Language Modeling

Translation language modeling (TLM) is designed for pre-training multilingual models. Given a pair of sentences in different languages, TLM masks some tokens in both sentences and trains the model to predict the masked tokens. The loss function for TLM is defined as:

$$\mathcal{L}_{\text{TLM}}^{(x)} = -\frac{1}{|M_x|} \sum_{i \in M_x} \log P(x_i / x_{\backslash M_x}, y_{\backslash M_y}) - \frac{1}{|M_y|} \sum_{i \in M_y} \log P(y_i / x_{\backslash M_x}, y_{\backslash M_y}) \tag{2.27}$$

In this context, $M_x$ and $M_y$ denote the sets of masked positions within sentences $x$ and $y$, while $x_{\backslash M_x}$ and $y_{\backslash M_y}$ signify the masked versions of $x$ and $y$ respectively.

#### 2.5.2.8 Alternate Language Modeling

Alternate language modeling (ALM) is used for cross-lingual model pre-training. It involves alternating the language of each sentence in the input sequence. Given a pair of parallel sentences $(x, y)$, a code-switched sentence is created by randomly replacing some phrases in $x$ with their translations from $y$. ALM follows the same masking procedure as the standard MLM for selecting tokens to be masked. By pre-training the model on these code-switched sentences, the model can learn relationships between languages more effectively.

$$\mathcal{L}_{\text{ALM}}^{(z(x,y))} = -\frac{1}{|M|} \sum_{i \in M} \log P(z_i / z_{\backslash M}) \tag{2.28}$$

In this context, $z$ represents the code-switched sentence generated from $x$ and $y$, $z_{\backslash M}$ denotes the masked version of $z$, and $M$ is the set of masked token positions within $z_{\backslash M}$.

#### 2.5.2.9 Sentence Boundary Objective

Sentence boundary objective (SBO) involves predicting masked tokens based on span boundary tokens and position embeddings. The loss function for SBO is defined as:

$$\mathcal{L}_{\mathrm{SBO}}^{(x)} = -\frac{1}{|S|} \sum_{i \in S} \log P(x_i / f(x_{s-1}, x_{e+1}, p_{s-e+1})) \tag{2.29}$$

where $f()$ is a two-layered feed-forward neural network, $S$ represents the positions of tokens in the contiguous span, $s$ and $e$ represent the start and end positions of the span, respectively, and $p$ represents the position embedding.

#### 2.5.2.10 Next Sentence Prediction

Next sentence prediction (NSP) is a binary sentence pair classification task. The loss function for NSP is defined as:

$$\mathcal{L}_{\mathrm{NSP}}^{(x,y)} = -\log P(d/x, y) \tag{2.30}$$

where $d$ is a binary variable representing whether the sentences $(x, y)$are consecutive (1) or not (0).

#### 2.5.2.11 Sentence Order Prediction

Sentence order prediction (SOP) focuses on sentence coherence, unlike NSP, which also includes topic prediction. SOP, introduced by ALBERT, involves determining whether sentences are in the correct order or swapped. The training instances are balanced with 50% swapped. The SOP loss is defined as:

$$\mathcal{L}_{\mathrm{SOP}}^{(x,y)} = -\log P(d/x, y) \tag{2.31}$$

where $d \in \{1, 0\}$ indicates whether the sentences are swapped.

#### 2.5.2.12 Sequence-to-Sequence Language Modeling

Sequence-to-Sequence Language Modeling (Seq2Seq) is an extension of MLM used for pre-training encoder-decoder-based models. The loss function for Seq2Seq is defined as:

$$\mathcal{L}_{\mathrm{Seq2Seq}}^{(x)} = -\frac{1}{l_s} \sum_{s=i}^{j} \log P(x_s/\widehat{x}, x_{i:s-1}) \tag{2.32}$$

where $\widehat{x}$ is the masked version of $x$ and $l_s$ represents the length of the masked n-gram span.

#### 2.5.2.13 Denoising Autoencoder

The denoising autoencoder (DAE) involves reconstructing the original text from the corrupted text. The loss function for DAE is defined as:

$$\mathcal{L}_{\text{DAE}} = -\frac{1}{|x|} \sum_{i=1}^{|x|} \log P(x_i / \widehat{x}, x_{<i}) \tag{2.33}$$

where $\widehat{x}$ is the corrupted version of $x$.

### 2.5.3 Architectures

Initially proposed by Vaswani et al. (2017), Transformers are composed of stacks of encoder and decoder layers. A Transformer-based language model can be pre-trained using a stack of encoders, decoders, or both, thus resulting in various architectures, as shown in Fig. 2.8.

#### 2.5.3.1 Encoder-Decoder

The encoder-decoder architecture is a two-part structure in which the encoder processes the input sequence, and the decoder generates the output. The encoder transforms the input into a continuous representation that holds all the learned information of the input. The decoder then uses this representation to generate the output sequence. This architecture is beneficial for sequence-to-sequence tasks such as machine translation and text summarization. For instance, in a machine translation task, the encoder processes the input sentence in the source language, and the decoder generates the translation in the target language. The attention mechanism in this architecture allows the model to focus on different parts of the input sequence while generating the output, providing a dynamic computation of context.

#### 2.5.3.2 Causal Decoder

The causal decoder architecture is designed for autoregressive tasks where the model generates the output token by token. This architecture employs a unidirectional attention mechanism, meaning that each token can only attend to previous tokens and itself during the generation process. This is particularly useful for text generation tasks where the model needs to generate coherent and contextually appropriate text. For example, in text completion tasks, the model predicts the next token based on the previous ones, ensuring that the generated text is coherent and contextually relevant.

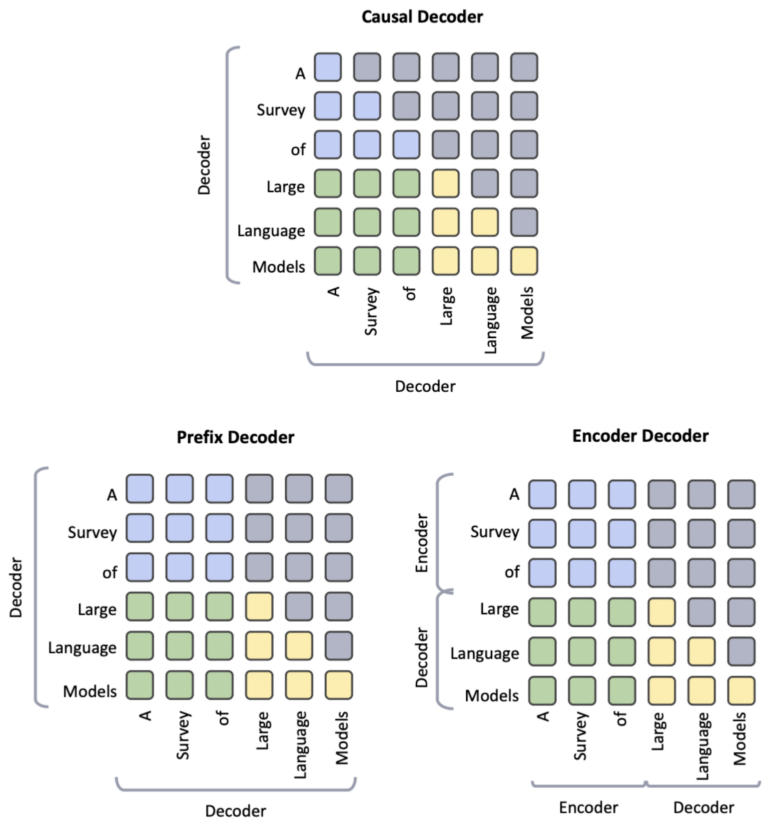

Fig. 2.8: Analysis of attention patterns across three primary architectures. In this context, the blue, green, yellow, and gray rounded shapes represent attention within prefix tokens, attention between prefix and target tokens, attention among target tokens, and masked attention, respectively.

### 2.5.3.3 Prefix Decoder

The prefix decoder architecture is a variation of the causal decoder where the model can attend bi-directionally to a prefix of tokens while maintaining unidirectional attention for the rest. This hybrid attention mechanism allows the model to have a broader context while generating each token, making it effective for tasks that require understanding both previous and subsequent tokens in a sequence. For instance, the model can attend to the dialog history and the partially generated response in a dialog system while generating the next token.

#### 2.5.3.4 Encoder

The encoder is designed to efficiently process and understand the contextual information embedded within input sequences, making it a preferred choice for certain NLP tasks. Each encoder layer within the architecture generates a robust contextual representation of the input sequence. The final output from the last encoder layer is utilized as the contextual representation, serving as a valuable input for diverse downstream tasks. The encoder architecture is particularly advantageous for tasks requiring a deep understanding of token context without requiring sequence generation, such as classification tasks.

#### 2.5.3.5 Mixture-of-Experts

The Mixture-of-Experts (MoE) architecture is a variant of Transformer models that incorporates MoE layers, replacing the standard feed-forward blocks as shown in Fig. 2.9. These layers contain multiple parallel units called "experts", each with unique parameters. A router directs input tokens to specific experts based on their capabilities. Experts, which are feed-forward layers following the attention block, process tokens independently. Unlike traditional models where capacity increases lead to higher computational costs, the MoE architecture simultaneously activates only a few experts. This sparse activation allows the architecture to support larger model sizes without a proportional increase in computational demand, maintaining efficient performance.

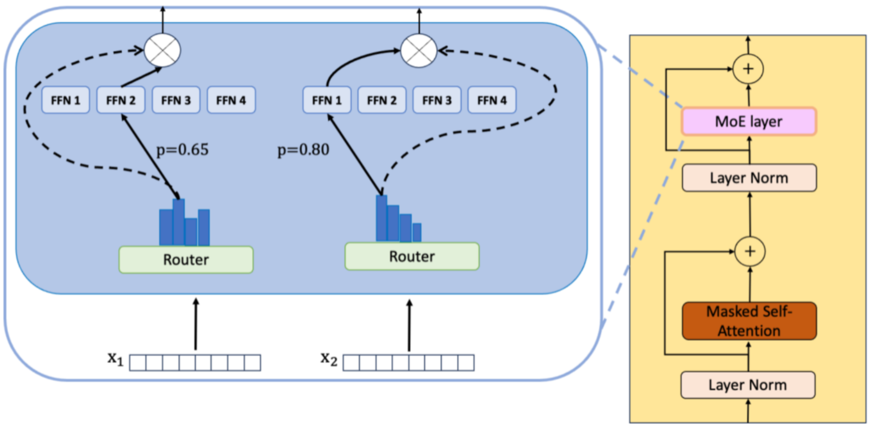

Fig. 2.9: Mixture-of-experts variant of the Transformer architecture.

### 2.5.4 LLM Pre-training Tips and Strategies

This section will explore the key configurations, methods, and strategies for training LLMs.

#### 2.5.4.1 Training Methods

- **Learning Rate** Most LLMs follow a similar learning rate schedule with warm-up and decay phases during pre-training. Initially, the learning rate is gradually increased for approximately 0.1% to 0.5% of the training steps, typically ranging from $5 \times 10^{-5}$ to $1 \times 10^{-4}$. After this phase, the learning rate is progressively reduced using a cosine decay strategy.
- **Batch Size** During language model pre-training, it is common to use large batch sizes, often with 2,048 examples or 4M tokens, to enhance stability and efficiency. Models such as GPT-3 and PaLM employ a dynamic approach, adjusting the batch size throughout training, with GPT-3's batch size, for instance, expanding from 32K to 3.2M tokens. This adaptive batch sizing has been shown to stabilize LLM training effectively.
- **Optimizers** For training LLMs such as GPT-3, the Adam and AdamW optimizers are commonly used. These optimizers adapt based on gradient estimations with typical hyper-parameters: $\beta_1 = 0.9, \beta_2 = 0.95$, and $\epsilon = 10^{-8}$. Additionally, the Adafactor optimizer, a memory-efficient variant of Adam, is employed for models such as PaLM and T5. Its hyper-parameters are $\beta_1 = 0.9$ and $\beta_2$ adjusted based on the number of training steps.

#### 2.5.4.2 Decoding Strategies

**Greedy Search**
This autoregressive decoding mechanism is one of the techniques utilizing decoder-only architectures. A most common decoding method herein is the *greedy search*. This method predicts the most probable token at each generation step, conditioned on the previously generated tokens. The mathematical formulation of this process is as follows:

$$x_i = \arg\max_x P(x|x_{<i}),$$

where $x_i$ denotes the token predicted at the $i$-th step, which is the most probable token given the context $x_{<i}$. Consider a partial sentence, "The sky is so", for illustration. The greedy search method might predict "blue" as the next token, given its high likelihood of completing the sentence appropriately. This approach is efficient in text generation tasks such as machine translation and text summarization, where there is a strong dependency between the input and the expected output.

The greedy search offers reliable results by leveraging probability and context in scenarios where the output must align closely with the input. This decoding strategy is not limited to decoder-only architectures and can be applied to encoder-decoder and prefix-decoder models. Many improvements to greedy search have been proposed, and we will discuss some of them here. Beam search is a notable strategy, holding onto the top-$n$ probable sentences during each decoding step and ultimately choosing the one with the highest probability.

**! Practical Tips**

Typically, a beam size between 3 to 6 is adequate, though increasing it may reduce performance. Length penalty, or length normalization, is another improvement that compensates for for beam search's tendency to prefer shorter sentences. This method modifies sentence probability about its length, applying an exponential power as a divisor. Penalties for generating previously used tokens have been introduced to mitigate the issue of generating repetitive tokens or $n$-grams. Additionally, diverse beam search offers a valuable improvement, yielding a variety of outputs from a single input.

**Random Search**

Sampling-based methods offer an alternative decoding strategy, introducing a probabilistic approach to token selection to foster diversity and randomness in text generation. This strategy is beneficial when the goal is to generate both varied and engaging text. For instance, given the context sentence, "I am thirsty. I would like a cup of", the probability distribution of the next token might favor words such as "tea," "coffee," or "water." However, sampling-based methods still allow the selection of words with lower probabilities, albeit at a reduced likelihood. While "tea" has the highest probability, words such as "coffee," "water," and "juice" still have a chance of being selected, introducing diversity to the responses. This approach applies to various architectures, including decoder-only, encoder-decoder, and prefix decoder models, offering flexibility for different language generation tasks.

Improvements to random sampling have been developed to enhance the quality of generated text by mitigating the selection of words with extremely low probabilities. One such improvement is temperature sampling, which adjusts the softmax function's temperature coefficient when calculating each token's probability over the vocabulary. This is given by:

$$P(x_j|x_{<i}) = \frac{\exp(l_j/t)}{\sum_{j'} \exp(l_{j'}/t)}$$

where $l'_j$ denotes the logits of each word and $t$ is the temperature coefficient. By reducing the temperature, words with higher probabilities are more likely to be selected, while those with lower probabilities are less likely. For instance, with a temperature of 1, the method defaults to random sampling. As the temperature approaches 0, it becomes akin to a greedy search, and as it increases indefinitely, it transitions to uniform sampling.

**! Practical Tips**

Another improvement is Top-$k$ sampling. This approach involves truncating tokens with lower probabilities and only sampling from those with the top $k$ highest probabilities. Top-$p$ sampling, or nucleus sampling, is another strategy. It samples from the smallest set of tokens whose cumulative probability is greater than or equal to a specified value $p$. This set is constructed by progressively adding tokens (sorted by descending generative probability) until the cumulative probability surpasses $p$. For example, if the tokens are sorted and added until their cumulative probability exceeds 0.8, only those tokens are considered for sampling.

#### 2.5.4.3 3D Parallelism

3D parallelism integrates three key parallel training techniques–data, pipeline, and tensor parallelism–for efficiently training LLMs.

**Data Parallelism**
This method enhances training speed by distributing model parameters and the training dataset across multiple GPUs. Each GPU processes its data and calculates gradients, and then these gradients are combined and used to update the model on each GPU.

**! Practical Tips**

The ZeRO technique, introduced by the DeepSpeed library, addresses memory redundancy in data parallelism. Typically, data parallelism forces every GPU to store an identical copy of an LLM, encompassing model parameters, gradients, and optimizer parameters (Rajbhandari et al., 2020). However, this redundancy leads to extra memory usage. ZeRO's solution is to keep only a portion of the data on each GPU, fetching the rest from other GPUs as needed. Three strategies based on data storage are proposed: optimizer state partitioning, gradient partitioning, and parameter partitioning. Tests show that the first two do not add to communication costs, while the third increases communication by approximately 50% but conserves memory based on the GPU count. PyTorch has also introduced a technique akin to ZeRO, named FSDP.

**Pipeline Parallelism**
Here, different layers of an LLM are spread across several GPUs. Sequential layers are assigned to the same GPU to minimize the data transfer costs. While basic imple-

mentations might under-utilize GPUs, advanced methods like GPipe and PipeDream enhance efficiency by processing multiple data batches simultaneously and updating gradients asynchronously (Harlap et al., 2018; Huang et al., 2019).

**Tensor Parallelism**
This technique divides LLMs' tensors or parameter matrices for distribution across multiple GPUs. For instance, the parameter matrix can be split column-wise and processed on different GPUs during matrix multiplication. The results from each GPU are then merged. Libraries such as Megatron-LM support tensor parallelism, which can be applied to more complex tensors (Shoeybi et al., 2019).

## 2.6 Commonly Used Pre-trained LLMs

This section delves into three prominent LLM architectures, examining them from the perspectives of the datasets employed, their alignment with the Transformer architecture, essential insights, and their diverse variants.

### 2.6.1 BERT (Encoder)

The *Bidirectional Encoder Representation from Transformer* (BERT) is a pre-trained model that employs an attention mechanism to better comprehend linguistic context (Devlin et al., 2019). BERT consists of multiple encoder segments, each contributing to its robustness. Upon its introduction, BERT set new benchmarks for a range of NLP tasks, such as question answering on the SQuAD v1.1 dataset and natural language inference on the MNLI dataset. Unlike traditional language models that process text sequences in a unidirectional manner, BERT's bidirectional training approach offers a more comprehensive understanding of linguistic context and sequence flow.

#### 2.6.1.1 Dataset

BERT's training data primarily comprise Wikipedia, accounting for approximately 2.5 billion words, and the BooksCorpus, which contains approximately 800 million words.

#### 2.6.1.2 Architecture

BERT is an encoder-only Transformer and offers various pre-trained models differentiated by their architectural scale. Two examples include:

- **BERT-BASE** consists of 12 layers, 768 hidden nodes, 12 attention heads, and 110 million parameters.
- **BERT-LARGE** is a more extensive version with 24 layers, 1024 hidden nodes, 16 attention heads, and 340 million parameters.

The training of BERT-BASE utilized four cloud TPUs over four days, while BERT-LARGE required 16 TPUs for the same duration.

#### 2.6.1.3 Training

BERT operates in two phases–pre-training and fine-tuning–as shown in Fig. 2.10. The model learns from unlabeled data across various tasks in the initial pre-training phase. During the fine-tuning phase, the model starts with the parameters acquired from the pre-training and then optimizes these parameters using labeled data specific to the target tasks.

BERT's training methodology combines two objectives: the *masked language model* (MLM) and *next sentence prediction* (NSP). The combined loss function of these techniques is minimized during training. For BERT, each training instance is a pair of sentences that may or may not be sequential in the original document. The special tokens `[CLS]` and `[SEP]` denote the beginning of the sequence and the separation between sentences, respectively. A subset of tokens in the training instance is either masked with a `[MASK]` token or substituted with a random token. Before being input into the BERT model, tokens are transformed into embedding vectors. These vectors are then enhanced with positional encodings, and in BERT's unique approach, segment embeddings are added to indicate whether a token belongs to the first or second sentence.

Once pre-trained, BERT can be adapted for various downstream tasks, whether for individual texts or pairs of texts. General linguistic representations, derived from BERT's 350 million parameters trained on 250 billion tokens, have significantly advanced the state of the art in numerous NLP tasks. During the fine-tuning process, additional layers can be incorporated into BERT. These layers and the pre-trained BERT parameters are updated to align with the training data of specific downstream tasks. The Transformer encoder, essentially a pre-trained BERT, accepts a sequence of text and uses the `[CLS]` representation for predictions. For example, `[CLS]` is replaced with actual classification labels in sentiment analysis or classification tasks. During this fine-tuning phase, the cross-entropy loss between the predictions and actual labels is minimized via gradient-based methods. The additional layers are trained from scratch, and the pre-trained BERT parameters undergo updates.

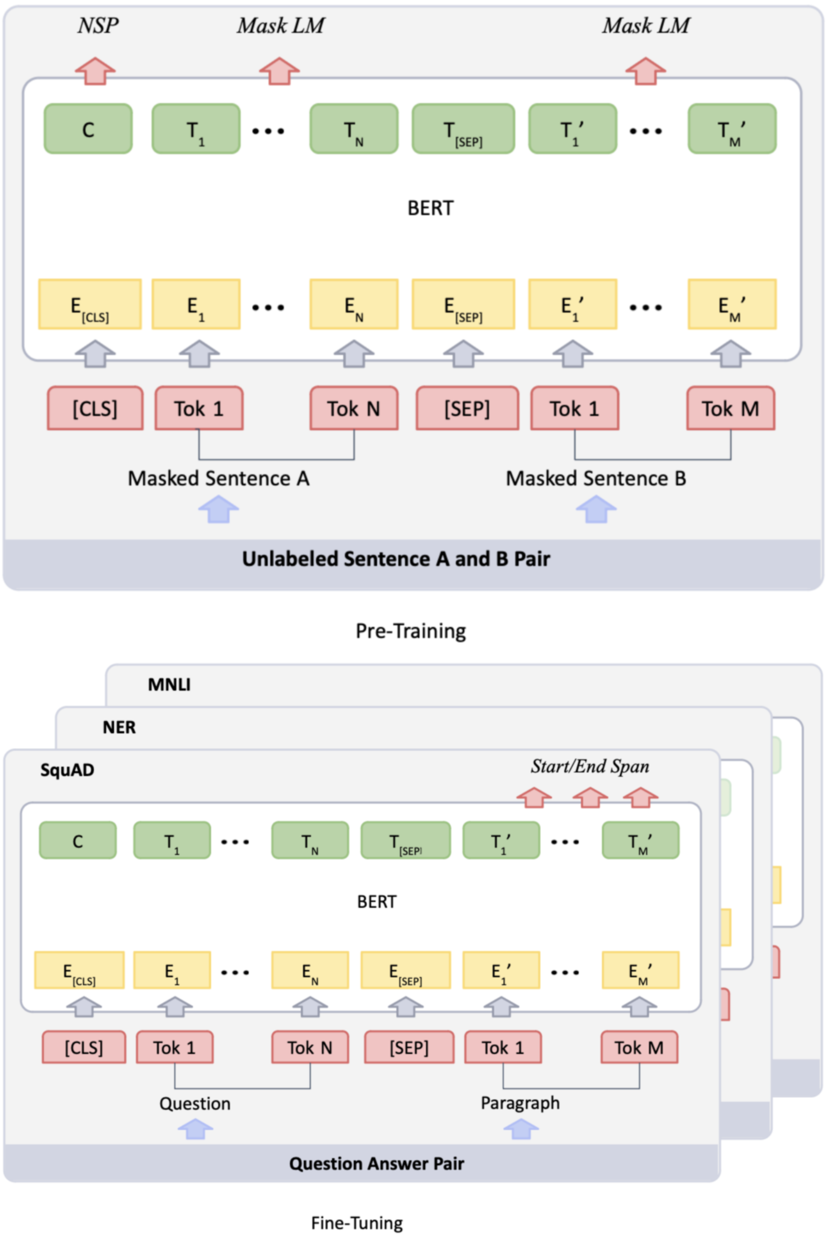

Fig. 2.10: BERT can adapt its pre-training objective to fine-tune on task-specific input data.

#### 2.6.1.4 Key Takeaways

1. The scale of the model is crucial. BERT-LARGE, encompassing 345 million parameters, is the most extensive model in its category. Despite having the same structure, it outperforms BERT-BASE, which contains "merely" 110 million parameters.
2. Given sufficient training data, increasing training steps correlates with enhanced accuracy. For example, in the MNLI task, BERT-BASE's accuracy sees a 1.0% boost when trained for 1 million steps (with a batch size of 128,000 words) instead of 500K steps with an identical batch size.
3. While BERT's bidirectional method (MLM) may converge at a slower rate than unidirectional (left-to-right) methods (given that only 15% of words are predicted in each batch), it surpasses the latter in performance after a limited number of pre-training iterations.

#### 2.6.1.5 Variants

Subsequent developments and variations of BERT have been introduced to enhance model architectures or pre-training objectives (Kamath et al., 2022). Notably:

- **RoBERTa**: A BERT variant of the same size, pre-trained on 200 billion tokens. The loss function used in BERT was found to be less impactful in this context.
- **ALBERT**: Improves efficiency by enforcing parameter sharing.
- **SpanBERT**: Focuses on representing and predicting text spans.
- **DistilBERT**: A lightweight version achieved through knowledge distillation.
- **ELECTRA**: Emphasizes replaced token detection.

### 2.6.2 T5 (Encoder-Decoder)

The *Text-to-Text Transfer Transformer* (T5) model introduces a comprehensive framework that consolidates various NLP transfer learning process elements (Raffel et al., 2020). This includes diverse unlabeled datasets, pre-training goals, benchmarks, and methods for fine-tuning. The framework identifies optimal practices to achieve superior performance by integrating and comparing these components via ablation experiments.