# Chapter 2
# Pre-trained Language Models


**Abstract** This chapter presents the main architecture types of attention-based language models, which describe the distribution of tokens in texts: Autoencoders similar to BERT receive an input text and produce a contextual embedding for each token. Autoregressive language models similar to GPT receive a subsequence of tokens as input. They produce a contextual embedding for each token and predict the next token. In this way, all tokens of a text can successively be generated. Transformer Encoder-Decoders have the task to translate an input sequence to another sequence, e.g. for language translation. First they generate a contextual embedding for each input token by an autoencoder. Then these embeddings are used as input to an autoregressive language model, which sequentially generates the output sequence tokens. These models are usually pre-trained on a large general training set and often fine-tuned for a specific task. Therefore, they are collectively called Pre-trained Language Models (PLM). When the number of parameters of these models gets large, they often can be instructed by prompts and are called Foundation Models. In further sections we described details on optimization and regularization methods used for training. Finally, we analyze the uncertainty of model predictions and how predictions may be explained.

**Keywords** BERT · Language model · GPT-2 · Transformer · Pre-training · Fine-tuning · Sequence-to-sequence model

A model that either computes the joint probability or the conditional probability of natural language texts is called a *language model* as it potentially covers all information about the language. In this chapter, we present the main architecture types of attention-based *language models* (*LMs*), which process texts consisting of sequences of *tokens*, i.e. words, numbers, punctuation, etc.:

- *Autoencoders* (*AE*) receive an input text and produce a contextual embedding for each token. These models are also called *BERT models* and are described in Sect. 2.1.



G. Paaß, S. Giesselbach, *Foundation Models for Natural Language Processing*,
Artificial Intelligence: Foundations, Theory, and Algorithms,
https://doi.org/10.1007/978-3-031-23190-2_2

- *Autoregressive language models* (*AR*) receive a subsequence $v_1, \ldots, v_{t-1}$ of tokens of the input text. They generate contextual embeddings for each token and use them to predict the next token $v_t$. In this way, they can successively predict all tokens of the sequence. These models are also called *GPT models* and are outlined in Sect. 2.2.
- *Transformer Encoder-Decoders* have the task to translate an input sequence in to another sequence, e.g. for language translation. First they generate a contextual embedding for each input token by an autoencoder. Then these embeddings are used as input to an autoregressive language model, which sequentially generates the output sequence tokens. These models are also called *Transformers* and are defined in Sect. 2.3.

In this chapter, we focus on NLP, where we consider sequences of text tokens. Historically, the transformer encoder-decoder was developed in 2017 by Vaswani et al. [141] to perform translation of text into another language. The *autoencoder* [39] and the *autoregressive language model* [118] are the encoder-part and the decoder-part of this transformer encoder-decoder and were proposed later. As they are conceptually simpler, they are introduced in preceding sections. A final section (Sect. 2.4) describes methods for optimizing models during training, determining a model architecture, and estimating the uncertainty of model predictions.

It turned out that the models can first be trained on a large training set of general text documents and are able to acquire the distribution of tokens in correct and fluent language. Subsequently, they can be adapted to a specific task, e.g. by fine-tuning with a small supervised classification task. Therefore, the models are called *Pre-trained Language models*.

As we will see later, all models can be applied to arbitrary sequences, e.g. musical notes, sound, speech, images, or even videos. When the number of parameters of these models gets large, they often can be instructed by prompts and are called *Foundation Models*.

## 2.1 BERT: Self-Attention and Contextual Embeddings

Common words often have a large number of different meanings. For the word *"bank"*, for instance, the lexical database WordNet [94] lists 18 different senses from *"sloping land"* to *"financial institution"*. In a simple embedding of the word *"bank"* introduced in Sect. 1.5 all these meanings are conflated. As a consequence, the interpretation of text based on these embeddings is flawed.

As an alternative, *contextual embeddings* or contextualized embeddings were developed, where the details of a word embedding depend on the word itself as well as on the neighboring words occurring in the specific document. Consequently, each occurrence of the same word in the text has a different embedding depending on the context. Starting with the Transformer [141], a number of approaches have been designed to generate these contextual embeddings, which are generally trained in an unsupervised manner using a large corpus of documents.

**BERT** (Bidirectional Encoder Representations from Transformers) was proposed by Devlin et al. [39] and is the most important approach for generating contextual embeddings. BERT is based on the concept of attention [8] and on prior work by Vaswani et al. [141]. The notion of **attention** is inspired by a brain mechanism that tends to focus on distinctive parts of memory when processing large amounts of information. The details of the computations are explained by Rush [126].

### *2.1.1 BERT Input Embeddings and Self-Attention*

As input BERT takes some text which is converted to tokens, e.g. by the Wordpiece tokenizer (Sect. 1.2) with a vocabulary of a selected size, e.g. 30,000. This means that frequent words like *"dog"* are represented by a token of their own, but more rare words like *"playing"* are split into several tokens, e.g. *"play"* and *"##ing"*, where *"##"* indicates that the token is part of a word. As all characters are retained as tokens, arbitrary words may be represented by a few tokens. In addition, there are special tokens like *[CLS]* at the first position of the input text and two *"[SEP]"* tokens marking the end of text segments. Finally, during training, there are *[MASK]* tokens as explained later. Each token is represented by a *token embedding*, a vector of fixed length $d_{emb}$, e.g. $d_{emb} = 768$. Input sequences of variable length are padded to the maximal length with a special padding token.

Since all token embeddings are processed simultaneously, the tokens need an indication of their position in the input text. Therefore, each position is marked with *position embeddings* of the same length as the token embeddings, which encode the position index. The BERT paper encodes the position number by trainable embeddings, which are added to the input token embeddings [39]. Finally, BERT compares the first and second input segment. Therefore, the algorithm needs the information, which token belongs to the first and second segment. This is also encoded by a trainable segment embedding added to the token and position embedding. The sum of all embeddings is used as *input embedding* for BERT. An example is shown in Fig. 2.1.

#### Self-Attention to Generate Contextual Embeddings

BERT starts with input embeddings $\boldsymbol{x}_t$ of length $d_{emb}$ for each token $v_t$ of the input sequence $v_1, \ldots, v_T$. These embeddings are transformed by linear mappings to so-called *query-vectors* $\boldsymbol{q}_t$, *key-vectors* $\boldsymbol{k}_t$ and *value-vectors* $\boldsymbol{v}_t$. These are computed by multiplying $\boldsymbol{x}_t$ with the matrices $\boldsymbol{W}^{(q)}$, $\boldsymbol{W}^{(k)}$, and $\boldsymbol{W}^{(v)}$ with dimensions $d_{emb} \times d_q$, $d_{emb} \times d_q$ and $d_{emb} \times d_v$ respectively

$$\boldsymbol{q}_t^\top = \boldsymbol{x}_t^\top \boldsymbol{W}^{(q)} \qquad \boldsymbol{k}_t^\top = \boldsymbol{x}_t^\top \boldsymbol{W}^{(k)} \qquad \boldsymbol{v}_t^\top = \boldsymbol{x}_t^\top \boldsymbol{W}^{(v)}. \tag{2.1}$$

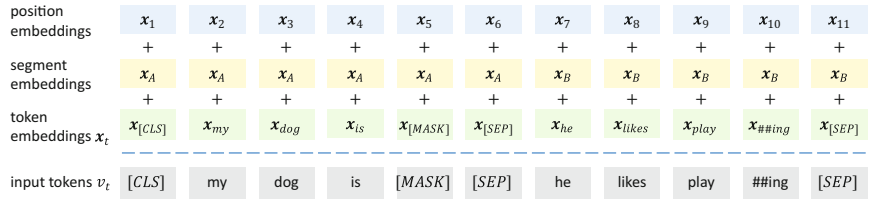


**Fig. 2.1** The input of the BERT model consist of a sequence of embeddings corresponding to the input tokens. Each token is represented by a sum consisting of the embedding of the token text, the embedding of its segment indicator and an embedding of its position [39]

Note that the query- and key-vectors have the same length. Then scalar products $\boldsymbol{q}_r^\intercal \boldsymbol{k}_t$ between the query-vector $\boldsymbol{q}_r$ of a target token $v_r$ and the key-vectors $\boldsymbol{k}_t$ of all tokens of the sequence are computed:

$$(\alpha_{r,1}, \ldots, \alpha_{r,T}) = \text{softmax}\left(\frac{\boldsymbol{q}_r^\intercal \boldsymbol{k}_1}{\sqrt{d_k}}, \ldots, \frac{\boldsymbol{q}_r^\intercal \boldsymbol{k}_T}{\sqrt{d_k}}\right). \tag{2.2}$$

Each scalar product yields a real-valued *association score* $(\boldsymbol{q}_r^\intercal \boldsymbol{k}_t)/\sqrt{d_k}$ between the tokens, which depends on the matrices $\boldsymbol{W}^{(q)}$ and $\boldsymbol{W}^{(k)}$. This association score is called *scaled dot-product attention*. It is normalized to a probability score $\alpha_{r,t}$ by the softmax function. The factor $1/\sqrt{d_k}$ avoids large values, where the softmax function has only tiny gradients. With these weights a weighted average of the value vectors $\boldsymbol{v}_t$ of all sequence elements is formed yielding the new embedding $\breve{\boldsymbol{x}}_r$ of length $d_v$ for the target token $v_r$:

$$\breve{\boldsymbol{x}}_r = \alpha_{r,1} * \boldsymbol{v}_1 + \cdots + \alpha_{r,T} * \boldsymbol{v}_T. \tag{2.3}$$

This algorithm is called *self-attention* and was first proposed by Vaswani et al. [141]. Figure 2.2 shows the computations for the $r$-th token *"mouse"*. Note that the resulting embedding is a *contextual embedding* as it includes information about all words in the input text. A component of $\boldsymbol{v}_t$ gets a high weight whenever the scalar product $\boldsymbol{q}_r^\intercal \boldsymbol{k}_t$ is large. It measures a specific form of a correlation between $\boldsymbol{x}_r$ and $\boldsymbol{x}_t$ and is maximal if the vector $\boldsymbol{x}_r^\intercal \boldsymbol{W}^{(q)}$ points in the same direction as $\boldsymbol{x}_t^\intercal \boldsymbol{W}^{(k)}$.

The self-attention mechanism in general is non-symmetric, as the matrices $\boldsymbol{W}^{(q)}$ and $\boldsymbol{W}^{(k)}$ are different. If token $v_i$ has a high attention to token $v_j$ (i.e. $\boldsymbol{q}_i^\intercal \boldsymbol{k}_j$ is large), this does not necessarily mean that $v_j$ will highly attend to token $v_i$ (i.e. $\boldsymbol{q}_j^\intercal \boldsymbol{k}_i$ also is large). The influence of $v_i$ on the contextual embedding of $v_j$ therefore is different from the influence of $v_j$ on the contextual embedding of $v_i$. Consider the following example text *"Fred gave roses to Mary"*. Here the word *"gave"* has different relations to the remaining words. *"Fred"* is the person who is performing the giving, *"roses"* are the objects been given, and *"Mary"* is the recipient of the given objects. Obviously these semantic role relations are non-symmetric. Therefore, they can be captured with the different matrices $\boldsymbol{W}^{(q)}$ and $\boldsymbol{W}^{(k)}$ and can be encoded in the embeddings.

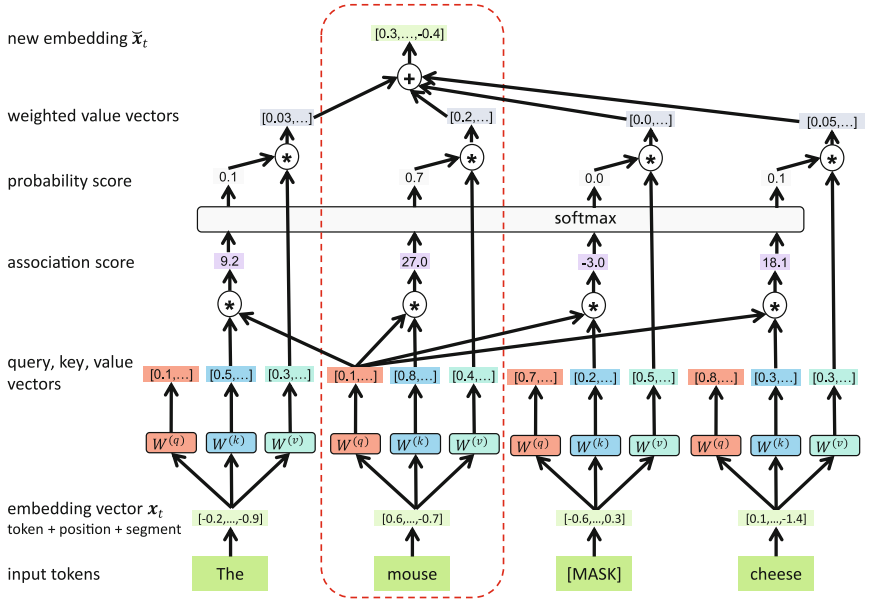


**Fig. 2.2** Computation of a contextual embedding for a single token *"mouse"* by self-attention. By including the embedding of "cheese", the embedding of mouse can be shifted to the meaning of "rodent" and away from "computer pointing device". Such an embedding is computed for every word of the input sequence

Self-attention allows for shorter computation paths and provides direct avenues to compare distant elements in the input sequence, such as a pronoun and its antecedent in a sentence. The multiplicative interaction involved in attention provides a more flexible alternative to the inflexible fixed-weight computation of MLPs and CNNs by dynamically adjusting the computation to the input at hand. This is especially useful for language modeling, where, for instance, the sentence *"She ate the ice-cream with the X"* is processed. While a feed-forward network would always process it in the same way, an attention-based model could adapt its computation to the input and update the contextual embedding of the word *"ate"* if $X$ is *"spoon"*, or update the embedding of *"ice-cream"* if $X$ refers to *"strawberries"* [17].

In practice all query, key, and value vectors are computed in parallel by $\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}^{(q)}$, $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}^{(k)}$, $\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}^{(v)}$, where $\boldsymbol{X}$ is the $T \times d_{emb}$ matrix of input embeddings [141]. The query-vectors $\boldsymbol{q}_t$, key-vectors $\boldsymbol{k}_t$ and value vectors $\boldsymbol{v}_t$ are the rows of $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ respectively. Then the self-attention output matrix ATTL(X) is calculated by one large matrix expression

$$\check{\boldsymbol{X}} = \text{ATTL}(\boldsymbol{X}) = \text{ATTL}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\intercal}{\sqrt{d_k}}\right)\boldsymbol{V}, \tag{2.4}$$

resulting in a $T \times d_v$-matrix $\breve{\boldsymbol{X}}$. Its $r$-th row contains the new embedding $\breve{\boldsymbol{x}}_r$ of the $r$-th token $v_r$.

A number of alternative compatibility measures instead of the scaled dot-product attention (2.2) have been proposed. They are, however, rarely used in PLMs, as described in the surveys [27, 46].

It turns out that a single self-attention module is not sufficient to characterize the tokens. Therefore, in a layer $d_{\text{head}}$ parallel self-attentions are computed with different matrices $\boldsymbol{W}_m^{(q)}$, $\boldsymbol{W}_m^{(k)}$, and $\boldsymbol{W}_m^{(v)}$, $m = 1, \ldots, d_{\text{head}}$, yielding partial new embeddings

$$\breve{\boldsymbol{X}}_m = \text{ATTL}(\boldsymbol{X}\boldsymbol{W}_m^{(q)}, \boldsymbol{X}\boldsymbol{W}_m^{(k)}, \boldsymbol{X}\boldsymbol{W}_m^{(v)}). \tag{2.5}$$

The emerging partial embeddings $\breve{\boldsymbol{x}}_{m,t}$ for a token $v_t$ are able to concentrate on complementary semantic aspects, which develop during training.

The $\text{BERT}_{\text{BASE}}$ model has $d_{\text{head}} = 12$ of these parallel *attention heads*. The lengths of these head embeddings are only a fraction $d_{emb}/d_{\text{head}}$ of the original length $d_{emb}$. The resulting embeddings are concatenated and multiplied with a $(d_{\text{head}} * d_v) \times d_{emb}$-matrix $W^{(o)}$ yielding the matrix of intermediate embeddings

$$\breve{\boldsymbol{X}} = \left[\breve{\boldsymbol{X}}_1, \ldots, \breve{\boldsymbol{X}}_{d_{\text{head}}}\right] \boldsymbol{W}_0, \tag{2.6}$$

where $\boldsymbol{W}_0$ is a parameter matrix. If the length of the input embeddings is $d_{emb}$, the length of the query, key, and value vector is chosen as $d_k = d_v = d_{emb}/d_{\text{head}}$. Therefore, the concatenation again creates a $T \times d_{emb}$ matrix $\breve{\boldsymbol{X}}$. This setup is called *multi-head self-attention*. Because of the reduced dimension of the individual heads, the total computational cost is similar to that of a single-head attention with full dimensionality.

Subsequently, each row of $\breve{\boldsymbol{X}}$, the intermediate embedding vectors $\breve{\boldsymbol{x}}_t^\mathsf{T}$, is converted by a *fully connected layer* FCL with a ReLU activation followed by another linear transformation [141]

$$\tilde{\boldsymbol{x}}_t^\mathsf{T} = \text{FCL}(\breve{\boldsymbol{x}}_t) = ReLU(\breve{\boldsymbol{x}}_t^\mathsf{T} * \boldsymbol{W}_1 + \boldsymbol{b}_1^\mathsf{T}) * \boldsymbol{W}_2 + \boldsymbol{b}_2^\mathsf{T}. \tag{2.7}$$

The matrices $\boldsymbol{W}_0, \boldsymbol{W}_1, \boldsymbol{W}_2$ and the vectors $\boldsymbol{b}_1, \boldsymbol{b}_2$ are parameters. These transformations are the same for each token $v_t$ of the sequence yielding the embedding $\tilde{\boldsymbol{x}}_t$.

To improve training speed, *residual connections* are added as a "bypass", which simply copy the input. They were shown to be extremely helpful for the optimization of multi-layer image classifiers [54]. In addition, *layer normalization* [6] is used for regularization (Sect. 2.4.2), as shown in Fig. 2.3. Together the multi-head self-attention (2.5), the concatenation (2.6), and the fully connected layer (2.7) form an *encoder block*.

This procedure is repeated for a number of $k$ layers with different encoder blocks, using the output embeddings of one block as input embeddings of the next block. This setup is shown in Fig. 2.4. The embeddings $\tilde{\boldsymbol{x}}_{k,t}$ of the last encoder block

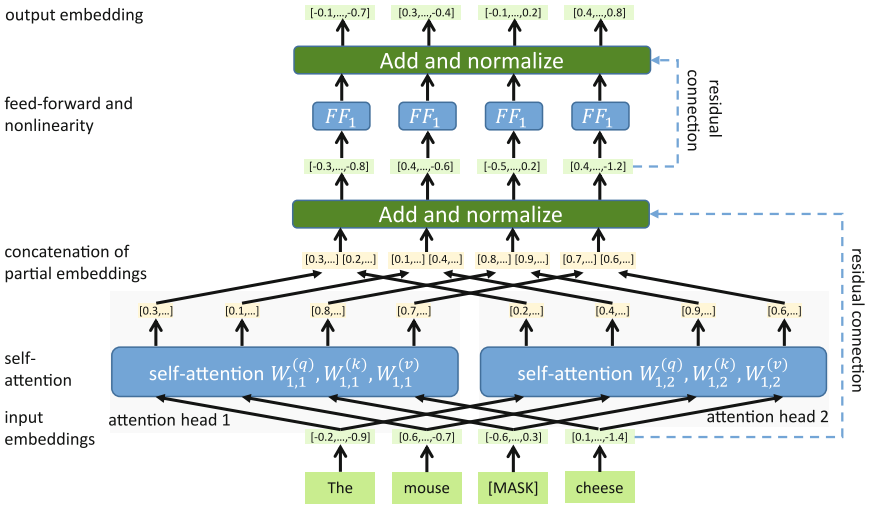


**Fig. 2.3** Multi-head self-attention computes self-attentions for each layer $l$ and head $m$ with different matrices $\boldsymbol{W}_{l,m}^{(q)}$, $\boldsymbol{W}_{l,m}^{(k)}$, and $\boldsymbol{W}_{l,m}^{(v)}$. In this way, different aspects of the association between token pairs, e.g. "mouse" and "cheese", can be computed. The resulting embeddings are concatenated and transformed by a feedforward network. In addition, residual connections and layer normalization improve training convergence [39]

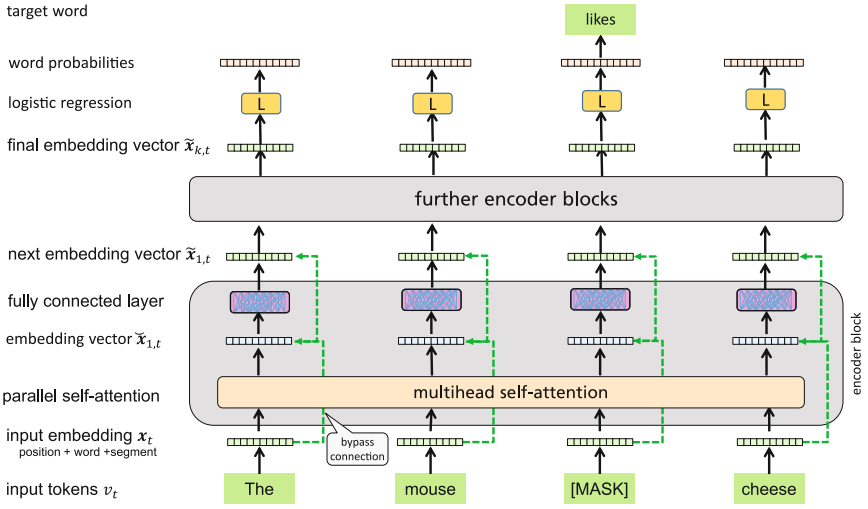


**Fig. 2.4** Parallel computation of contextual embeddings in each encoder block by BERT. The output embeddings of an encoder block are used as input embeddings of the next encoder block. Finally, masked tokens are predicted by a logistic classifier $L$ using the corresponding contextual embedding of the last encoder block as input

provides the desired contextual embeddings. The structure of an encoder block overcomes the limitations of RNNs (namely the sequential nature of RNNs) by allowing each token in the input sequence to directly determine associations with every other token in the sequence. $\text{BERT}_{\text{BASE}}$ has $k = 12$ encoder blocks. It was developed at Google by Devlin et al. [39]. More details on the implementation of self-attention can be found in these papers [38, 41, 126].

### 2.1.2 Training BERT by Predicting Masked Tokens

The BERT model has a large number of unknown parameters. These parameters are trained in a two-step procedure.

- *Pre-training* enables the model to acquire general knowledge about language in an unsupervised way. The model has the task to fill in missing words in a text. As no manual annotation is required, pre-training can use large text corpora.
- *Fine-tuning* adjusts the pre-trained model to a specific task, e.g. sentiment analysis. Here, the model parameters are adapted to solve this task using a smaller labeled training dataset.

The performance on the fine-tuning task is much better than without pre-training because the model can use the knowledge acquired during pre-training through *transfer learning*.

To pre-train the model parameters, a training task is designed: the *masked language model* (*MLM*). Roughly 15% of the input tokens in the training documents are selected for prediction, which is performed by a logistic classifier (Sect. 1.3)

$$p(V_t|v_1, \ldots, v_{t-1}, v_{t+1} \ldots, v_T) = \text{softmax}(A\tilde{\boldsymbol{x}}_{k,t} + \boldsymbol{b}), \tag{2.8}$$

receiving the embedding $\tilde{\boldsymbol{x}}_{k,t}$ of the last layer at position $t$ as input to predict the random variable $V_t$ of possible tokens at position $t$. This approach avoids cycles where words can indirectly "see themselves".

The tokens to be predicted have to be changed, as otherwise the prediction would be trivial. Therefore, a token selected for prediction is replaced by:

- a special *[MASK]* token for 80% of the time (e.g., *"the mouse likes cheese"* becomes *"the mouse [MASK] cheese"*);
- a random token for 10% of the time (e.g., *"the mouse likes cheese"* becomes *"the mouse absent cheese"*);
- the unchanged label token for 10% of the time (e.g., *"the mouse likes cheese"* becomes *"the mouse likes cheese"*).

The second and third variants were introduced, as there is a discrepancy between pre-training and the subsequent fine-tuning, were there is no *[MASK]* token. The authors mitigate this issue by occasionally replacing *[MASK]* with the original token, or by sampling from the vocabulary. Note that in 1.5% of the cases a

random token is inserted. This occasional noise encourages BERT to be less biased towards the masked token (especially when the label token remains unchanged) in its bidirectional context encoding. To predict the masked token, BERT has to concentrate all knowledge about this token in the corresponding output embedding of the last layer, which is the input to the logistic classifier. Therefore, it is often called an *autoencoder*, which generates extremely rich output embeddings.

In addition to predicting the masked tokens, BERT also has to predict, whether the next sentence is a randomly chosen sentence or the actual following sentence (*next sentence prediction*). This requires BERT to consider the relation between two consecutive pieces of text. Again a logistic classifier receiving the embedding of the first *[CLS]* token is used for this classification. However, this task did not have a major impact on BERT's performance, as BERT simply learned if the topics of both sentences are similar [158].

In Fig. 2.4 the task is to predict a high probability of the token *"likes"* for the input text *"The mouse [MASK] cheese"*. At the beginning of the training this probability will be very small ($\approx$ 1/no. of tokens). By backpropagation for each unknown parameter the derivative can be determined, indicating how the parameters should be changed to increase the probability of *"likes"*. The unknown parameters of BERT comprise the input embeddings for each token of the vocabulary, the position embeddings for each position, matrices $\boldsymbol{W}_{l,m}^{(q)}$, $\boldsymbol{W}_{l,m}^{(k)}$, $\boldsymbol{W}_{l,m}^{(v)}$ for each layer $l$ and attention head $m$ (2.4), the parameters of the fully connected layers (2.7) as well as $A, \boldsymbol{b}$ of the logistic classifier (2.8). BERT uses the Adam algorithm [69] for stochastic gradient descent.

The $\text{BERT}_{\text{BASE}}$ model has a hidden size of $d_{emb} = 768$, $k = 12$ encoder blocks each with $d_{\text{head}} = 12$ attention heads, and a total of 110 million parameters. The $\text{BERT}_{\text{LARGE}}$ model has a hidden size of $d_{emb} = 1024$, and $k = 24$ encoder blocks each with $d_{\text{head}} = 16$ attention heads and a total of 340 million parameters [39]. The English Wikipedia and a book corpus with 3.3 billion words were encoded by the WordPiece tokenizer [154] with a vocabulary of 30,000 tokens and used to pre-train BERT. No annotations of the texts by humans were required, so the training is self-supervised. The pre-training took 4 days on 64 TPU chips, which are very fast GPU chips allowing parallel processing. Fine-tuning can be done on a single Graphical Processing Unit (GPU).

To predict the masked tokens, the model has to learn many types of language understanding features: syntax (*[MASK]* is a good position for a verb), semantics (e.g. the mouse prefers cheese), pragmatics, coreference, etc. Note that the computations can be processed in parallel for each token of the input sequence, eliminating the sequential dependency in Recurrent Neural Networks. This parallelism enables BERT and related models to leverage the full power of modern SIMD (single instruction multiple data) hardware accelerators like GPUs/TPUs, thereby facilitating training of NLP models on datasets of unprecedented size. Reconstructing missing tokens in a sentence has long been used in psychology. Therefore, predicting masked tokens is also called a *cloze task* from 'closure' in Gestalt theory (a school of psychology).

It turns out that BERT achieves excellent results for the prediction of the masked tokens, and that additional encoder blocks markedly increase the accuracy. For example, BERT is able to predict the original words (or parts of words) with an accuracy of 45.9%, although in many cases several values are valid at the target position [125]. In contrast to conventional language models, the MLM takes into account the tokens before and after the masked target token. Hence, it is called a *bidirectional encoder*. In addition, self-attention directly provides the relation to distant tokens without recurrent model application. Finally, self-attention is fast, as it can be computed in parallel for all input tokens of an encoder block.

### 2.1.3 Fine-Tuning BERT to Downstream Tasks

Neural networks have already been pre-trained many years ago [16], but the success of pre-training has become more evident in recent years. During pre-training BERT learns general syntactic and semantic properties of the language. This can be exploited for a special training task during subsequent *fine-tuning* with a modified training task. This approach is also called *transfer learning* as the knowledge acquired during pre-training is transferred to a related application. In contrast to other models, BERT requires minimal architecture changes for a wide range of natural language processing tasks. At the time of its publication, BERT improved the SOTA on various natural language processing tasks.

Usually, a fine-tuning task requires a classification, solved by applying a logistic classifier $L$ to the output embedding $\tilde{\boldsymbol{x}}_{k,1}$ of the *[CLS]* token at position 1 of BERT's last encoder block. There are different types of fine-tuning tasks, as shown in Fig. 2.5.

- *Text classification* assigns a sentence to one of two or more classes. Examples are the classification of restaurant reviews as positive/negative or the categorization of sentences as good/bad English. Here the output embedding of the start token *[CLS]* is used as input to $L$ to generate the final classification.
- *Text pair classification* compares two sentences separated by *"[SEP]"*. Examples include classifying whether the second sentence implies, contradicts, or is neutral with respect to the first sentence, or whether the two sentences are semantically equivalent. Again the output embedding of the start token *[CLS]* is used as input to $L$. Sometimes more than one sentence is compared to the root sentence. Then outputs are computed for every sentence pair and jointly normalized to a probability.
- *Word annotation* marks each word or token of the input text with a specific property. An example is *Named Entity Recognition* (*NER*) annotating the tokens with five name classes (e.g. "person", "location", ..., "other"). Here the same logistic model $L$ is applied to every token output embedding $\tilde{\boldsymbol{x}}_{k,t}$ at position $t$ and yields a probability vector of the different entity classes.

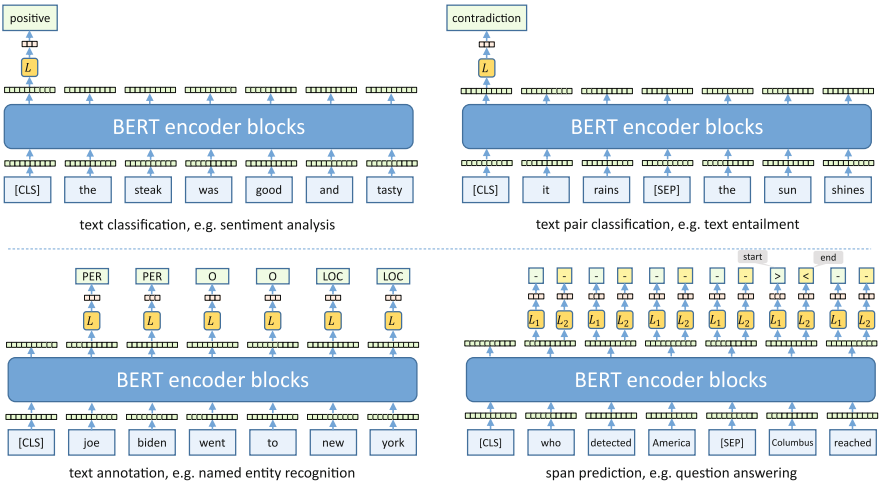


**Fig. 2.5** For fine-tuning, BERT is enhanced with an additional layer containing one or more logistic classifiers $L$ using the embeddings of the last layer as inputs. This setup may be employed for text classification and comparison of texts with the embedding of *[CLS]* as input of the logistic classifier. For sequence tagging, $L$ predicts a class for each sequence token. For span prediction, two logistic classifiers $L_1$ and $L_2$ predict the start and end of the answer phrase [39]

- *Span prediction* tags a short sequence of tokens within a text. An example is *question answering*. The input to BERT consists of a question followed by *"[SEP]"* and a context text, which is assumed to contain the answer. Here two different logistic classifiers $L$ and $\tilde{L}$ are applied to every token output embedding $\tilde{\boldsymbol{x}}_{k,t}$ of the context and generate the probability that the answer to the question starts/ends at the specific position. The valid span (i.e. the end is not before the start) with the highest sum of start/end scores is selected as the answer. An example is the input *"[CLS] When did Caesar die ? [SEP] ... On the Ides of March, 44 BC, Caesar was assassinated by a group of rebellious senators ..."*, where the answer to the question is the span *"Ides$_{start}$ of March, 44 BC$_{end}$"*. Span prediction may be applied to a number of similar tasks.

Therefore, BERT just needs an extra layer with one or more logistic classifiers for fine-tuning. During fine-tuning with a downstream application, parameters of the logistic models are learned from scratch and usually all parameters in the pre-trained BERT model are adapted. The parameters for the logistic classifiers of the masked language model and the next sentence prediction are not used during fine-tuning.

### 2.1.4 Visualizing Attentions and Embeddings

According to Bengio et al. [14], a good representation of language should capture the implicit linguistic rules and common sense knowledge contained in text data, such as lexical meanings, syntactic relations, semantic roles, and the pragmatics of language use. The contextual word embeddings of BERT can be seen as a big step in this direction. They may be used to disambiguate different meanings of the same word.

The self-attention mechanism of BERT computes a large number of "associations" between tokens and merges embeddings according to the strengths of these associations. If $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ are the embeddings of the input tokens $v_1, \ldots, v_T$, the associations $\boldsymbol{q}_r^\intercal \boldsymbol{k}_t$ are determined between the query $\boldsymbol{q}_r^\intercal = \boldsymbol{x}_r^\intercal \boldsymbol{W}^{(q)}$ and the key $\boldsymbol{k}_t^\intercal = \boldsymbol{x}_t^\intercal \boldsymbol{W}^{(k)}$ vectors (2.1). Then a sum of value vectors $\boldsymbol{v}_t^\intercal = \boldsymbol{x}_t^\intercal \boldsymbol{W}^{(v)}$ weighted with the normalized associations is formed yielding the new embeddings (2.3).

This is repeated with different matrices $\boldsymbol{W}_{l,m}^{(q)}, \boldsymbol{W}_{l,m}^{(k)}, \boldsymbol{W}_{l,m}^{(v)}$ in $m$ self-attention heads and $l$ layers. Each layer and head the new embeddings thus captures different aspects of the relations between the embeddings of each layer. For $\text{BERT}_{\text{BASE}}$ we have $l = 12$ layers and $m = 12$ bidirectional self-attention heads in each layer yielding 144 different "associations" or self-attentions. For the input sentence *"The girl and the boy went home. She entered the door."* Figure 2.6 shows on the left side the strength of associations for one of the 144 self-attention heads. Between every pair of tokens of the sentence an attention value is calculated and its strength is symbolized by lines of different widths. We see that the pronoun *"she"* is strongly associated with *"the girl"*. In the subsequent calculations (c.f. Fig. 2.2) the word *"she"* is disambiguated by merging its embedding with the embeddings of *"the"* and *"girl"* generating a new *contextual embedding* of *"she"*, which includes its relation to *"girl"*. On the right side of the figure the input *"The girl and the boy went home. He entered the door."* is processed. Then the model creates an association of *"boy"* with *"he"*.

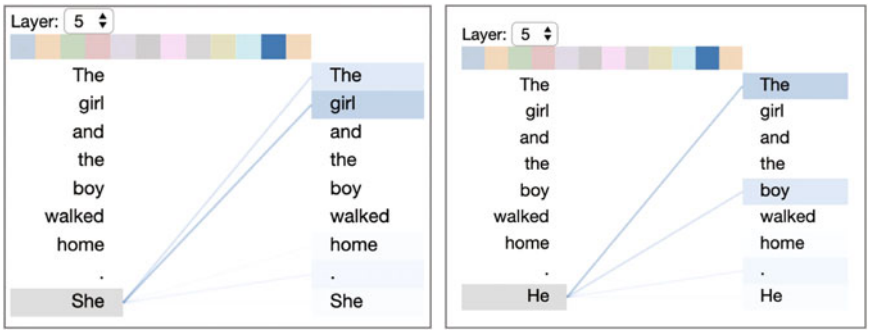


**Fig. 2.6** Visualization of a specific self-attention in the fifth layer of a BERT model with BERTviz [142]. If the next sentence contains the pronoun *"she"* this is associated with *"the girl"*. If this pronoun is changed to *"he"* it is related to *"the boy"*. Image created with BERTviz [142], with kind permission of the author

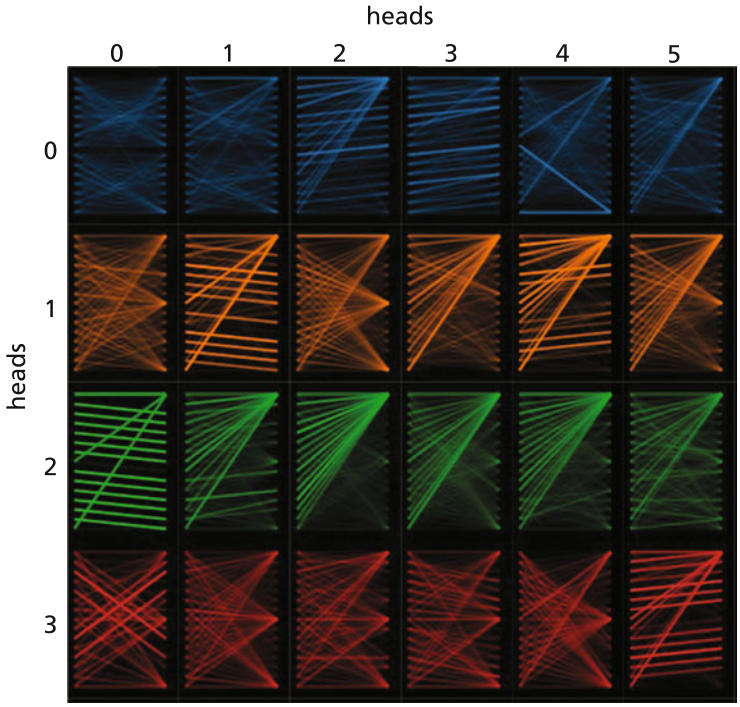


**Fig. 2.7** Visualization of some of the 144 self-attention patterns computed for the sentence *"[CLS] the cat sat on the mat [SEP] the cat lay on the rug[SEP]"* with BERTviz. Image reprinted with kind permission of the author [142]

Figure 2.7 shows a subset of the self-attention patterns for the sentence *"[CLS] the cat sat on the mat [SEP] the cat lay on the rug [SEP]"*. The self-attention patterns are automatically optimized in such a way that they jointly lead to an optimal prediction of the masked tokens. It can be seen that the special tokens *[CLS]* and *[SEP]* often are prominent targets of attentions. They usually function as representatives of the whole sentence [124]. Note, however, that in a multilayer PLM the embeddings generated by different heads are concatenated and transformed by a nonlinear transformation. Therefore, the attention patterns of a single head do not contain the complete information [124]. Whenever the matrices are randomly initialized, the self-attention patterns will be completely different, if the training is restarted with new random parameter values. However, the overall pattern of attentions between tokens will be similar.

Figure 2.10 shows on the left side a plot of six different senses of the token embeddings of *"bank"* in the *Senseval-3 dataset* projected to two dimensions by *T-SNE* [140]. The different senses are identified by different colors and form well-separated clusters of their own. Senses which are difficult to distinguish, like *"bank building"* and *"financial institution"* show a strong overlap [153]. The graphic

demonstrates that BERT embeddings have the ability to distinguish different senses of words which are observed frequently enough.

There is an ongoing discussion on the inner workings of self attention.Tay et al [134] empirically evaluated the importance of the dot product $\boldsymbol{q}_r^\intercal \boldsymbol{k}_s$ on natural language processing tasks and concluded that query-key interaction is "useful but not that important". Consequently they derived alternative formulae, which in some cases worked well and failed in others. A survey of attention approaches is provided by de Santana Correia et al. [37]. There are a number of different attention mechanisms computing the association between embedding vectors [50, 61, 104, 151]. However, most current large-scale models still use the original scaled dot-product attention with minor variations, such as other activation functions and regularizers (c.f. Sect. 3.1.4).

The fully connected layers $\text{FCL}(\check{\boldsymbol{x}}_t)$ in (2.7) contain 2/3 of the parameters of BERT, but their role in the network has hardly been discussed. Geva et al. [49] show that fully connected layers operate as key-value memories, where each key is correlated with text patterns in the training samples, and each value induces a distribution over the output vocabulary. For a key the authors retrieve the training inputs, which yield the highest activation of the key. Experts were able to assign one or more interpretations to each key. Usually lower fully connected layers were associated with shallow patterns often sharing the last word. The upper layers are characterized by more semantic patterns that describe similar contexts. The authors demonstrate that the output of a feed-forward layer is a composition of its memories.

### *2.1.5 Natural Language Understanding by BERT*

An outstanding goal of PLMs is *Natural Language Understanding* (*NLU*). This cannot be evaluated against a single task, but requires a set of benchmarks covering different areas to assess the ability of machines to understand natural language text and acquire linguistic, common sense, and world knowledge. Therefore, PLMs are fine-tuned to corresponding real-world downstream tasks.

**GLUE** [146] is a prominent benchmark for NLU. It is a collection of nine NLU tasks with public training data, and an evaluation server using private test data. Its benchmarks cover a number of different aspects, which can be formulated as classification problems:

- Determine the sentiment (positive/negative) of a sentences (SST-2).
- Classify a sentence as grammatically acceptable or unacceptable (CoLA).
- Check if two sentences are similar or are paraphrases (MPRC, STS-B, QQP).
- Determine if the first sentence entails the second one (MNLI, RTE).
- Check if sentence $B$ contains the answer to question $A$ (QNLI).
- Specify the target of a pronoun from a set of alternatives (WNLI).

Each task can be posed as *text classification* or *text pair classification* problem. The performance of a model is summarized in a single average value, which has the value 87.1 for human annotators [145]. Usually, there is an online leaderboard where the performance of the different models are recorded. A very large repository of leaderboards is on the PapersWithCode website [109]. Table 2.1 describes the tasks by examples and reports the performance of $\text{BERT}_{\text{LARGE}}$. BERT was able to lift the SOTA of average accuracy from 75.2 to 82.1%. This is a remarkable increase, although the value is still far below the human performance of 87.1 with much room for improvement. Recent benchmark results for NLU are described in Sect. 4.1 for the more demanding SuperGLUE and other benchmarks.

### BERT's Performance on Other Fine-Tuning Tasks

The pre-training data is sufficient to adapt the large number of BERT parameters and learn very detailed peculiarities about language. The amount of training data for pre-training usually is much higher than for fine-tuning. Fine-tuning usually only requires two or three passes through the fine-tuning training data. Therefore, the stochastic gradient optimizer changes most parameters only slightly and sticks relatively close to the optimal pre-training parameters. Consequently, the model is usually capable to preserve its information about general language and to combine it with the information about the fine-tuning task.

Because BERT can reuse its general knowledge about language acquired during pre-training, it produces excellent results even with small fine-tuning training data [39].

- **CoNLL 2003** [128] is a benchmark dataset for *Named entity recognition* (*NER*), where each token has to be marked with a named entity tag, e.g. PER (for person), LOC (for location), ..., O (for no name) (Sect. 5.3). The task involves text annotation, where a label is predicted for every input token. BERT increased SOTA from 92.6% to 92.8% F1-value on the test data.
- **SQuAD 1.0** [120] is a collection of 100k triples of questions, contexts, and answers. The task is to mark the span of the answer tokens in the context. An example is the question *"When did Augustus die?"*, where the answer *"14 AD"* has to be marked in the context *"... the death of Augustus in AD 14 ..."* (Sect. 6.2). Using span prediction BERT increased the SOTA of SQuAD from 91.7% to 93.2%, while the human performance was measured as 91.2%.

From these experiments a large body of evidence has been collected demonstrating the strengths and weaknesses of BERT [124]. This is discussed in Sect. 4.2.

In summary, the advent of the BERT model marks a new era of NLP. It combines two pre-training tasks, i.e., predicting masked tokens and determining whether the second sentence matches the first sentence. Transfer learning with unsupervised pre-training and supervised fine-tuning becomes the new standard.

**Table 2.1** GLUE language understanding tasks. $\text{BERT}_{\text{LARGE}}$ was trained for three epochs on the fine-tuning datasets [38]. The performance of the resulting models is printed in the last column yielding an average value of 82.1

| Task | Description | Example | Metric | BERT |
|---|---|---|---|---|
| CoLA | Is the sentence grammatical or ungrammatical? | *"This building is than that one."* → *Ungrammatical* | Matthews correlation | 60.5 |
| SST-2 | Is the movie positive, negative, or neutral? | *"The movie is funny, smart, visually inventive, and most of all, alive."* → *Positive* | Accuracy | 94.9 |
| MRPC | Is the sentence $B$ a paraphrase of sentence $A$? | $A$: *"Today, Taiwan reported 35 new infections."* $B$: *"Taiwan announced another 35 probable cases at noon."* → *Paraphrase* | Accuracy | 89.3 |
| STS-B | How similar are sentences $A$ and $B$? | $A$: *"Elephants are walking down a trail."* $B$: *"A herd of elephants is walking down a trail."* → *Similar* | Pearson/ Spearman correlation | 86.5 |
| QQP | Are the two questions similar? | $A$: *"How can I increase the speed of my Internet connection while using a VPN?"* $B$: *"How can Internet speed be increased by hacking through DNS?"* → *Not Similar* | Accuracy | 72.1 |
| MNLI-mm | Does sentence $A$ entail or contradict sentence $B$? | $A$: *"Tourist information offices can be very helpful."* $B$: *"Tourist information offices are never of any help."* → *Contradiction* | Accuracy | 85.9 |
| QNLI | Does sentence $B$ contain the answer to the question in sentence $A$? | $A$: *"Which collection of minor poems are sometimes attributed to Virgil."* $B$: *"A number of minor poems, collected in the Appendix Vergiliana, are often attributed to him."* → *contains answer* | Accuracy | 92.7 |
| RTE | Does sentence $A$ entail sentence $B$? | $A$: *"Yunus launched the microcredit revolution, funding 50,000 beggars, whom Grameen Bank respectfully calls 'Struggling Members.'"* $B$: *"Yunus supported more than 50,000 Struggling Members."* → *Entailed* | Accuracy | 70.1 |
| WNLI | Sentence $B$ replaces sentence $A$'s pronoun with a noun - is this the correct noun? | $A$: *"Lily spoke to Donna, breaking her concentration."* $B$: *"Lily spoke to Donna, breaking Lily's concentration."* → *Incorrect* | Accuracy | 60.5 |

### 2.1.6 Computational Complexity

It is instructive to illustrate the computational effort required to train PLMs. Its growth determines the time needed to train larger models that can massively improve the quality of language representation. Assume $D$ is the size of the hidden embeddings and the input sequence has length $T$, then the intermediate dimension of the fully connected layer FCL is set to $4D$ and the dimension of the keys and values are set to $D/H$ as in Vaswani et al. [141]. Then according to Lin et al. [81] we get the following computational complexities and parameters counts of self-attention and the position-wise FCL (2.7):

| Module | Complexity | # Parameters |
|---|---|---|
| Self-attention | $O(T^2 * D)$ | $4D^2$ |
| Position-wise FCL | $O(T * D^2)$ | $8D^2$ |

As long as the input sequence length $T$ is small, the hidden dimension $D$ mainly determines the complexity of self-attention and position-wise FCL. The main limiting factor is the FCL. But when the input sequences become longer, the sequence length $T$ gradually dominates the complexity of these modules, so that self-attention becomes the bottleneck of the PLM. Moreover, the computation of self-attention requires that an attention score matrix of size $T \times T$ is stored, which prevents the computation for long input sequences. Therefore, modifications reducing the computational effort for long input sequences are required.

To connect all input embeddings with each other, we could employ different modules. Fully connected layers require $T * T$ networks between the different embeddings. Convolutional layers with a kernel width $K$ do not connect all pairs and therefore need $O(\log_K(T))$ layers in the case of dilated convolutions. RNNs have to apply a network $T$ times. This leads to the following complexities per layer [81, 141]

| Layer type | Complexity per layer | Sequential operations | Maximum path length |
|---|---|---|---|
| Self-attention | $O(T^2 * D)$ | $O(1)$ | $O(1)$ |
| Recurrent | $O(T * D^2)$ | $O(T)$ | $O(T)$ |
| Fully connected | $O(T^2 * D^2)$ | $O(1)$ | $O(1)$ |
| Convolutional | $O(K * T * D^2)$ | $O(1)$ | $O(\log_K(T))$ |
| Restricted self-attention | $O(R * T * D)$ | $O(1)$ | $O(T/R)$ |

The last line describes a restricted self-attention, where self-attention only considers a neighborhood of size $R$ to reduce computational effort. Obviously the computational complexity per layer is a limiting factor. In addition, computation for recurrent layers need to be sequential and cannot be parallelized, as shown in the

column for sequential operations. The last column shows the path length, i.e. the number of computations to communicate information between far-away positions. The shorter these paths between any combination of positions in the input and output sequences, the easier it is to learn long-range dependencies. Here self-attention has a definite advantage compared to all other layer types. Section 3.2 discusses advanced approaches to process input sequences of larger length. In conclusion, BERT requires less computational effort than alternative layer types.

### *2.1.7 Summary*

*BERT* is an autoencoder model whose main task is to derive context-sensitive embeddings for tokens. In a preliminary step, tokens are generated from the words and letters of the training data in such a way that most frequent words are tokens and arbitrary words can be composed of tokens. Each token is encoded by an input embedding. To mark the position of each input token, a position embedding is added to the input embedding.

In each layer of BERT, the lower layer embeddings are transformed by self-attention to a new embedding. Self-attention involves the computation of scalar products between linear transformations of embeddings. In this way, the embeddings in the next layer can adapt to tokens from the context, and the embeddings become context-sensitive. The operation is performed in parallel for several attention heads involving different linear projections. The heads can compute associations in parallel with respect to different semantic features. The resulting partial embeddings are concatenated to a new embedding. In addition to self-attention heads, each encoder block contains a fully connected layer as well as normalization operations.

The original BERT model consists of six encoder blocks and generates a final embedding for each input token. BERT is pre-trained on a very large document collection. The main pre-training task is to predict words from the input sequence, which have been replaced by a [MASK] token. This is done by using the last layer embedding of the token as input to a logistic classifier, which predicts the probabilities of tokens for this position. During pre-training the model parameters are optimized by stochastic gradient descent. This forces the model to collect all available information about that token in the output embedding. The first input token is the [CLS] token. During pre-training, it is used for next sentence prediction, where a logistic classifier with the [CLS]-embedding as input has to decide, if the first and second sentence of the input sequence belong together or not.

Typically, the pre-trained model is fine-tuned for a specific task using a small annotated training dataset. An example is the supervised classification task of whether the input text expresses a positive, negative or neutral sentiment. Again a logistic classifier with the [CLS]-embedding as input has to determine the probability of the three sentiments. During pre-training all parameters of the model are adjusted slightly. It turns out that this transfer learning approach has a much

higher accuracy than supervised training only on the small training dataset, since the model can use knowledge about language acquired during pre-training.

Experiments show that BERT is able to raise the SOTA considerably in many language understanding tasks, e.g. the GLUE benchmark. Other applications are named entity recognition, where names of persons, locations, etc. have to be identified in a text, or question answering, where the answer to a question has to be extracted from a paragraph. An analysis of computational complexity shows that BERT requires less computational effort than alternative layer types. Overall, BERT is the workhorse of natural language processing and is used in different variants to solve language understanding problems. Its encoder blocks are reused in many other models.

Chapter 3 describes ways to improve the performance of BERT models, especially by designing new pre-training tasks (Sect. 3.1.1). In Chap. 4 the knowledge acquired by BERT models is discussed. In the Chaps. 5–7, we describe a number of applications of BERT models such as relation extraction (Sect. 5.4) or document retrieval (Sect. 6.1).

## 2.2 GPT: Autoregressive Language Models

### *2.2.1 The Task of Autoregressive Language Models*

To capture the information in natural language texts the conditional probability of tokens can be described by a language model. These *autoregressive language models* aim to predict the probability of the next token in a text given the previous tokens. If $V_{t+1}$ is a random variable whose values are the possible tokens $v_{t+1}$ at position $t+1$, we have to calculate the conditional probability distribution $p(V_{t+1}|v_1, \ldots, v_t)$. According to the definition of conditional probability the probability of the complete text $v_1, \ldots, v_T$ can be computed as

$$p(V_1 = v_1, \ldots, V_T = v_T) = p(V_T = v_T | v_1, \ldots, v_{T-1}) * \cdots * p(V_1 = v_1). \quad (2.9)$$

Therefore, the conditional probability can represent all information about valid sentences, including adequate and bad usage of language. Qudar et al. [115] provide a recent survey of language models.

In Sect. 1.6, we used RNNs to build language models. However, these had problems determining long-range interactions between tokens. As an alternative, we can employ self-attention to infer contextual embeddings of the past tokens $v_1, \ldots, v_t$ and predict the next token $v_{t+1}$ based on these embeddings.

Consequently, we need to restrict self-attention to the tokens $v_1, \ldots, v_t$. This is the approach taken by the **Generative Pre-trained Transformer** (**GPT**) [116, 118]. Before training, the text is transformed to tokens, e.g. by byte-pair encoding (Sect. 1.2). On input, these tokens are represented by token embeddings and position embeddings (Sect. 2.1.1). During training the GPT-model performs the self-attention computations described in Sect. 2.1.1 in the same way as for BERT. For

predicting the probabilities of different tokens at position $t+1$, the self-attentions are restricted to previous tokens $v_1, \ldots, v_t$ and their embeddings. The probability of the possible next tokens at position $t+1$ is computed by a logistic classifier

$$p(V_{t+1}|v_1, \ldots, v_t) = \text{softmax}(A\tilde{\boldsymbol{x}}_{k,t} + \boldsymbol{b}), \tag{2.10}$$

which takes as input the embedding $\tilde{\boldsymbol{x}}_{k,t}$ of the last layer $k$ at position $t$ to predict the random variable $V_{t+1}$ of possible tokens at position $t+1$ (Fig. 2.8). This approach is called *masked self-attention* or *causal self-attention* because the prediction depends only on past tokens. Since GPT generates the tokens by sequentially applying the same model, it is called an *autoregressive language model*.

### 2.2.2 Training GPT by Predicting the Next Token

The training objective is adapted to the language modeling task of GPT. Figure 2.8 shows the range of computations for two consecutive tokens. By *teacher forcing* the model uses the observed tokens $v_1, \ldots, v_t$ up to position $t$ to compute self-attentions and predict the token probabilities for the next token $v_{t+1}$. This is justified by the factorization (2.9) of the full distribution. Note that the contextual embedding of a token $v_s$, $s < t$, changes each time when a new token $v_{t+1}, v_{t+2}, \ldots$ is taken into account in the masked self-attention. As GPT considers only the tokens before the target token $v_{t+1}$, it is called an *unidirectional encoder*. An intuitive high-level overview over GPT is given by Alammar [3].

During training the model parameters have to be changed by optimization such that the probabilities of observed documents (2.9) get maximal. By this *Maximum Likelihood estimation* (*MLE*) the parameters can be optimized for a large corpus of documents. To avoid numerical problems this is solved by maximizing the *log-likelihood*, sum of logarithms of (2.9)

$$\log p(v_1, \ldots, v_T) = \log p(v_T|v_1, \ldots, v_{T-1}) + \cdots + \log p(v_2|v_1) + \log p(v_1). \tag{2.11}$$

Alternatively we can minimize the negative log-likelihood $-\log p(v_1, \ldots, v_T)$.

GPT-2 can process an input sequence of 1024 tokens with an embedding size of 1024. In its medium version it has 345M parameters and contains 24 layers, each with 12 attention heads. For the training with gradient descent a batch size of 512 was utilized. The model was trained on 40 GB of text crawled from Reddit, a social media platform. Only texts that were well rated by other users were included, resulting in a higher quality data set. The larger model was trained on 256 cloud TPU v3 cores. The training duration was not disclosed, nor the exact details of training.

The quality of a language model may be measured by the probability $p(v_1, \ldots, v_T)$ of a given text collection $v_1, \ldots, v_T$. If we normalize its inverse

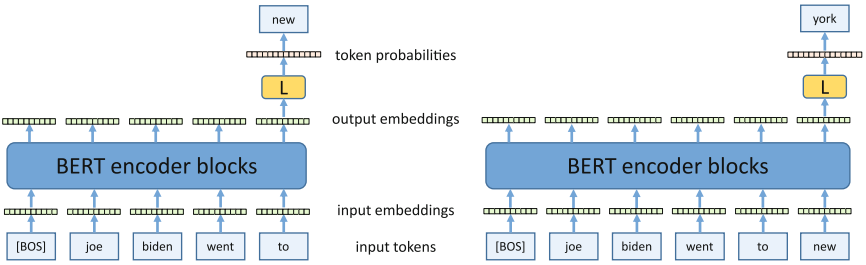


**Fig. 2.8** The input of the GPT model are the embeddings of tokens $v_1, \ldots, v_t$ up to position $t$. GPT computes contextual self-embeddings of these tokens in different layers and uses the output embedding of the last token $v_t$ ="*to*" in the highest layer to predict the probabilities of possible tokens at position $t+1$ with a logistic classifier $L$. This probability should be high for the actually observed token "*new*" (left). Then the observed token $v_{t+1}$ ="*new*" is appended to the input sequence and included in the self-attention computation for predicting the probabilities of possible tokens at position $t+2$, which should be high for "*york*" (right)

by the number $T$ of tokens we get the *perplexity* [28]

$$ppl(v_1, \ldots, v_T) := p(v_1, \ldots, v_T)^{-\frac{1}{T}}. \tag{2.12}$$

A low perplexity indicates a high probability of the text. If we assume that the conditional probabilities $p(v_t|v_1, \ldots, v_{t-1})$ are identical for all $t$, we get $ppl(v_1, \ldots, v_T) = 1/p(v_t|v_1, \ldots, v_{t-1})$, i.e. the inverse probability of the next token. GPT-2 was able to substantially reduce the perplexity on a number of benchmark data sets, e.g. from 46.5 to 35.8 for the *Penn Treebank corpus* [117] meaning that the actual words in the texts were predicted with higher probability.

### Visualizing GPT Embeddings

Kehlbeck et al. [66] investigated the relative location of embeddings in multivariate space for both BERT and GPT-2, each with 12 layers. They calculated 3-D projections using both *principal component analysis (PCA)* [111] and UMAP [89]. The latter can preserve the local structure of neighbors, but—differently to PCA—is unable to correctly maintain the global structure of the data. These 3d-scatterplots can be interactively manipulated on the website [66]. It turns out that GPT-2 forms two separate clusters: There is a small cluster containing just all tokens at position 0, while the embeddings at other positions form ribbon-like structures in the second cluster.

Careful investigations have indicated that most embedding vectors are located in a narrow cone, leading to high cosine similarities between them [25]. The authors identify isolated clusters and low dimensional manifolds in the contextual embedding space. Kehlbeck et al. [66] show that tokens with the same part-of-speech tag form ribbon-like structures in the projections (Fig. 2.9 left). Function words are all located on a tight circular structure, whereas content words like nouns

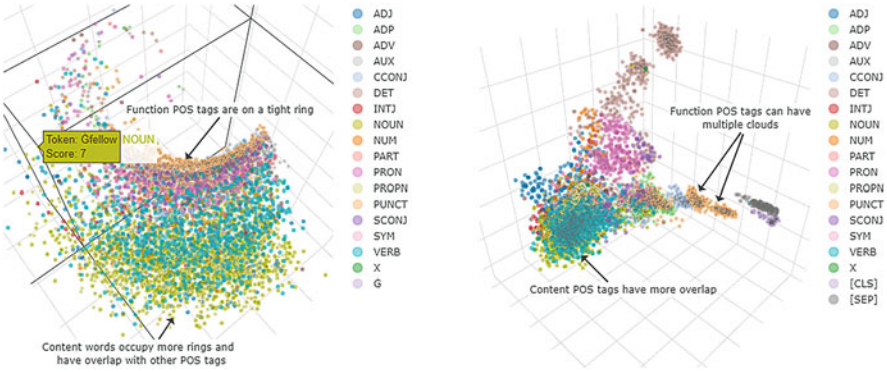


**Fig. 2.9** Visualization of embeddings with PCA together with the corresponding part-of speech tags. On the left side are GPT-2 embeddings of layer 0 of tokens of positions $> 0$ which form ribbon-like structures for the different POS tags, with function words close to the top. On the right side the embeddings of BERT for layer 0 are shown. Image reprinted with kind permission of the author [66]

and verbs are located in other elongated structures and have overlap with other POS-tags. The embeddings generated by BERT form one or more clusters (Fig. 2.9 right). They are quite separated for function words, but show some overlap for content words like nouns, verbs, or adjectives.

The GPT-2 embeddings of content words like *"banks"* and *"material"* at positions $> 0$ form elongated band-structures, as shown in the right part of Fig. 2.10. For higher layers the PCA projections get more diffuse. The user can read the token context by pointing to each dot.

Token-based *self-similarity* is the mean cosine similarity of the same token found in different sentences. In BERT as well as GPT-2, the self-similarity is higher for content than function words [66]. This may indicate that function words have more diverse semantic roles in different contexts. It is interesting to evaluate the 10 nearest neighbors of a token with respect to cosine similarity. In the lower layers, for both models the nearest tokens were in most cases the same tokens, except for a few content words. In the higher layers this changed and different tokens were the nearest tokens. This shows that more and more context is included in the embeddings of higher layers.

The authors also investigated the embeddings generated by a number of other PLM types. They find that their structure is very different as they form different clusters and manifolds. They argue that this structure has to be taken into account for new applications of the models.

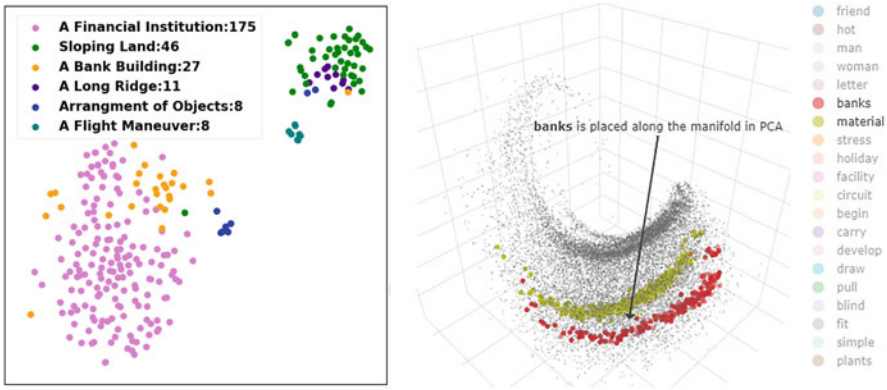


**Fig. 2.10** Plot of BERT-embeddings of different senses of *"bank"* projected to two dimensions by T-SNE (left). The legend contains a short description of the respective WordNet sense and the frequency of occurrence in the training data. Image[153]. The right side shows PCA projections of the embeddings of *"banks"* (lower strip) and *"material"* (middle strip) as well as other words computed for different contexts. Image interactively generated, printed with kind permission of the authors [66]

### *2.2.3 Generating a Sequence of Words*

After training the GPT model can predict the probabilities of the tokens at the next position $t+1$ given the previous tokens $v_1, \ldots, v_t$. To generate a text we have to select a sequence of tokens according to these probabilities.

- *Random sampling* selects the next token according to the predicted probabilities. This approach sometimes can select very improbable tokens such that the probability of the whole sentence gets too low. Although the individual probabilities are tiny, the probability of selecting an element of the group of improbable tokens is quite high. In addition, the estimates of small probability are often affected by errors.
- *Top-k sampling* takes into account only the $k$ tokens with the highest probability to generate the next token. The probability mass is redistributed among them [42] and used for randomly selecting a token.
- *Top-p sampling* considers the smallest set of top candidates with the cumulative probability above a threshold (e.g. $p = 0.95$) and then selects the next token according to the redistributed probabilities [58]. This approach limits the probability mass of rare tokens which are ignored.

There are also strategies which explicitly avoid previously generated tokens by reducing the corresponding scores in the update formula [67]. Both top-$k$ and top-$p$ sampling usually generate plausible token sequences and are actually employed to generate texts.

There are a number of approaches to improve token selection. Meister et al. [90] found that human-produced text tends to have evenly distribution of "surprise". This means that the next token should on average not be too rare and not be too frequent. They propose a number of sampling criteria, e.g. a variance regularizer.

Martins et al. [86] argue that softmax-generated output distributions are unrealistic, as they assign a positive probability to every output token. They propose the *Entmax transformation* which generates a sparse probability distribution from the computed scores, where part of the probabilities are exactly zero. The Entmax transformation can be controlled by a parameter $\alpha \geq 1$. For $\alpha = 1$ we get softmax and $\alpha = \infty$ recovers $\arg\max$. For intermediate values $\infty > \alpha > 1.0$ some tokens get exactly zero probability. Entmax losses are convex and differentiable and therefore may be trained by backpropagation. As in top-$p$ sampling and in opposition to top-$k$ sampling, Entmax sampling considers a varying number of tokens depending on the context. Experiments show that Entmax leads to better perplexities and less repetitions than other approaches. Compared with top-$p$ sampling it has a higher variation in the number of tokens considered.

Khandelwal et al. [68] try to improve the estimated probabilities of the language model by statistics of token $n$-grams. They perform a nearest neighbor search on the last tokens already processed. As distance measure they use the distances of the pre-trained embedding space. From the retrieved nearest neighbors they get additional evidence on the probable next token, which is merged with the token probabilities of the language model. In this way, they are able to improve the perplexity of language models. The approach is particularly helpful in predicting rare patterns, e.g. factual knowledge.

Yang et al. [157] analyze the properties of the softmax function. They find that the standard softmax does not have enough capacity to model natural language, as it restricts the rank of the mapping to probabilities. They propose to predict probabilities by a *Mixture of Softmaxes*, a convex combination of different logistic classifiers, which is more expressive than a single softmax. The authors show that this modification yields better perplexities in language modeling and also improves the performance of other transformer architectures [101].

### *2.2.4 The Advanced Language Model GPT-2*

**GPT-2** [118] is the first language model, which is able to generate documents of grammatically correct and semantically plausible text. Its largest version has 48 encoder blocks with 1.5B parameters and covers sequences of 1600 tokens. Given an initial text the model adapts to the style and content of this text and generates an answer, which often cannot be distinguished from human-generated continuations. Longer generated texts, however, sometimes tend to be repetitive and less coherent.

For GPT-2 top-$k$ truncated sampling was used to generate the example text [117] shown in Fig. 2.11. As can be seen there are no syntax errors and the generated content is plausible. The authors remark that one in two trials were of high quality.

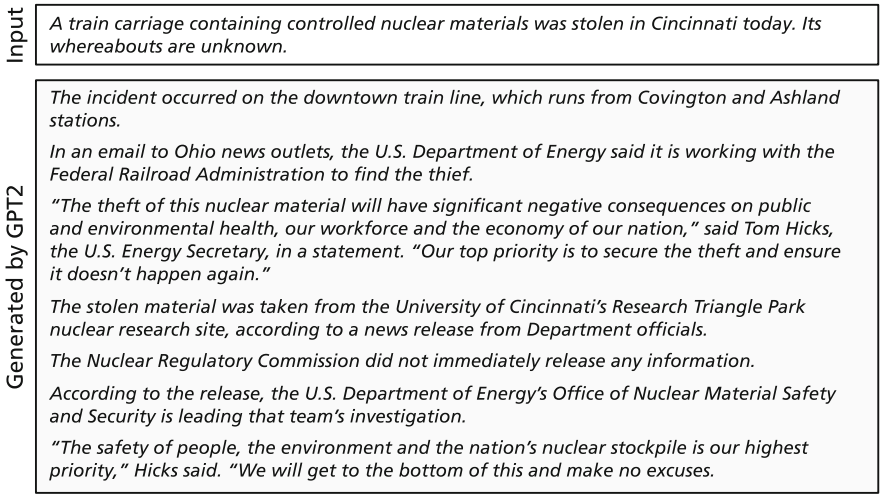


**Fig. 2.11** Given the input text, GPT-2 generates a continuation by top-$k$ sampling [117]. Quoted with kind permission of the authors

The model adapts to the style and content of the input text. This allows the user to generate realistic and coherent continuations about a topic they like. Obviously the topic has to be mentioned in the Reddit training data, which covers a broad spectrum of themes such as news, music, games, sports, science, cooking, and pets.

The model was able to solve many tasks better than previous models without being trained on the specific task. This type of learning is called *zero-shot learning*. For example, GPT-2 had a perplexity of 35.8 on the test set of the Penn Treebank compared to the inferior prior SOTA of 46.5 [117]. This was achieved without training GPT-2 on the *Penn Treebank corpus* [135].

### *2.2.5 Fine-Tuning GPT*

By fine-tuning, GPT-2 may be adapted to new types of text, for example new genres of text. To create song lyrics, for example, St-Amant [4] uses a dataset of 12,500 English rock song lyrics and fine-tunes GPT-2 for 5 epochs. Then the model is able to continue the lyrics of pop songs, which had not been seen by the model during training. The model had a high BLEU score of 68 when applied to song lyrics. Another experiment describes the generation of poetry [19].

Similar to BERT, a pre-trained GPT-2 can also be modified to perform a classification task. An example is fine-tuning to the classification of the sentiment of a document as positive or negative. Radford et al. [116] encode the classification task as a text with specific tokens and a final end token *[END]*. Then the model has to predict the sequence. The embedding of *[END]* in the highest layer is used as

input to a logistic classifier, which is trained to predict the probability of classes. The authors found that including language modeling (2.11) of the fine-tuning data as an auxiliary objective to fine-tuning improved generalization and accelerated convergence. They were able to improve the score on GLUE (Sect. 2.1.5) from 68.9 to 72.8 and achieved SOTA in 7 out of 8 GLUE tasks for natural language understanding. The results show that language models capture relevant information about syntax and semantics.

However, GPT operates from left to right when predicting the next token. In the sentences *"I went to the bank to deposit cash"* and *"I went to the bank to sit down"*, it will create the same context-sensitive embedding for *"bank"* when predicting *"sit"* or *"deposit"*, although the meaning of the token *"bank"* is different in both contexts. In contrast, BERT is bidirectional and takes into account all tokens of the text when predicting masked tokens. This fact explains why BERT for some tasks shows a better performance.

### *2.2.6 Summary*

GPT has an architecture similar to a BERT model that generates the tokens of a sentence one by one. It starts with an input sequence of tokens, which can be empty. Tokens are encoded as a sum of token embeddings and position embeddings. GPT uses the same encoder blocks as BERT, but the computations are masked, i.e. restricted to the already generated tokens. For these tokens the model produces contextual embeddings in several layers. The embedding of the last token in the top layer is entered into a logistic classifier and this calculates the probability of the tokens for the next position. Subsequently, the observed token is appended to the input at the next position and the computations are repeated for the next but one position. Therefore, GPT is called an autoregressive language model.

During training the parameters are changed by stochastic gradient descent in such a way that the model predicts high probabilities of the observed tokens in the training data. The maximum likelihood criterion is used, which optimizes the probability of the input data. When the model has been trained on a large text dataset it can be applied. Conditional to a start text it can sequentially compute the probability of the next token. Then a new token can be selected according to the probabilities.

If all alternative tokens are taken into account, rare tokens are often selected. Usually, the number of eligible tokens is restricted to $k$ high-probability tokens (top-$k$ sampling) or only high-probability tokens are included up to a prescribed probability sum $p$ (top-$p$ sampling). In this way, much better texts are generated. Advanced language models like GPT-2 have billions of parameters and are able to generate plausible stories without syntactic errors.

GPT models can also be fine-tuned. A first type of fine-tuning adapts the model to a specific text genre, e.g. poetry. Alternatively, GPT can be used as a classifier, where the output embedding of the most recently generated token for an input text is input to a logistic classifier. With this approach, GPT-2 was able to improve SOTA for

most natural language understanding task in the GLUE benchmark. This shows that GPT-2 has acquired a comprehensive knowledge about language. However, since self-attention is only aware of past tokens, models like BERT are potentially better as they can take into account all input tokens during computations.

Chapter 3 discusses how to improve the performance of GPT models, in particular by using more parameters (Sect. 3.1.2). These large models with billions of parameters can be instructed to perform a number of tasks without fine-tuning (Sect. 3.6.3). In the Chaps. 5–7, we describe a number of applications of GPT-models such as question-answering (Sect. 6.2.3), story generation (Sect. 6.5), or image generation from text (Sect. 7.2.6).

## 2.3 Transformer: Sequence-to-Sequence Translation

### *2.3.1 The Transformer Architecture*

Translation models based on Recurrent Neural Networks (Sect. 1.6) have a major limitation caused by the sequential nature of RNNs. The number of operations required to determine the relation between tokens $v_s$ and $v_t$ grows with the distance $t - s$ between positions. The model has to store the relations between all tokens simultaneously in a vector, making it difficult to learn complex dependencies between distant positions.

The *Transformer* [141]—similar to RNN-translation models—is based on an encoder and a decoder module (Fig. 2.13). The encoder is very similar to BERT, while the decoder resembles GPT. It is a *sequence-to-sequence model* (*Seq2seq*), which translates a source text of the input language to a target text in the target language. Instead of relating distant tokens by a large number of computation steps, it directly computes the self-attention between these token in parallel in one step.

The *encoder* generates contextual embeddings $\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_{T_{\text{src}}}$ of the source text tokens $v_1, \ldots, v_{T_{\text{src}}}$ with exactly the same architecture as the BERT model (Fig. 2.4). The original transformer [141] uses 6 encoder blocks. The generated embeddings of the last layer are denoted as $\check{\boldsymbol{x}}_1, \ldots, \check{\boldsymbol{x}}_{T_{\text{src}}}$.

The transformer *decoder* step by step computes the probability distributions $p(S_t|s_1, \ldots, s_{t-1}, v_1, \ldots, v_{T_{\text{src}}})$ of target tokens $s_t$ similar to the Recurrent Neural Network. Note that the source tokens $v_i$ as well as observed target tokens $s_j$ are taken as conditions. By the definition of conditional probability this yields the total probability of the output distribution

$$\begin{aligned} &p(S_1 = s_1, \ldots, S_T = s_T | v_1, \ldots, v_{T_{\text{src}}}) \\ &= p(S_T = s_T | s_1, \ldots, s_{T-1}, v_1, \ldots, v_{T_{\text{src}}}) \cdots p(S_1 = s_1 | v_1, \ldots, v_{T_{\text{src}}}), \end{aligned} \tag{2.13}$$

where $S_t$ is a random variable with the possible target tokens $s_t$ at position $t$ as its values. This probability is maximized during training.

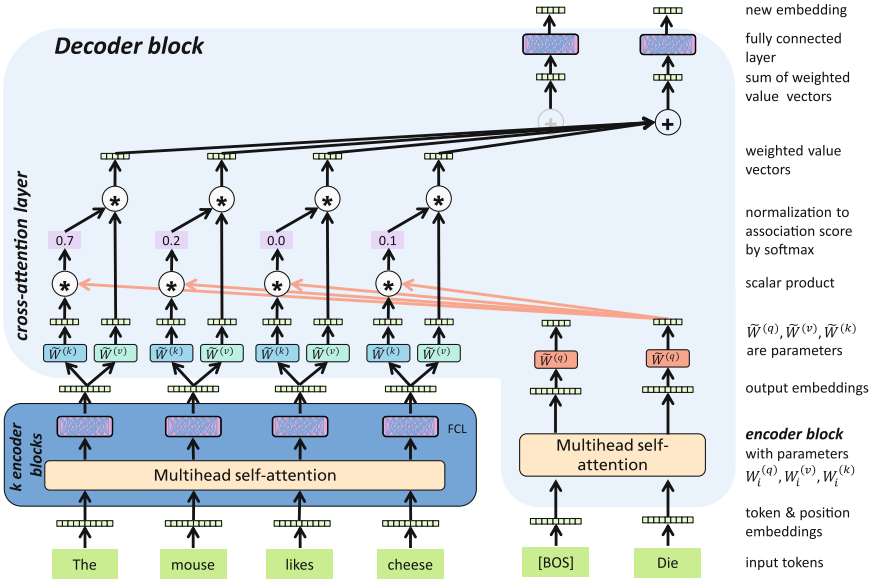


**Fig. 2.12** The transformer [141] uses $k$ encoder blocks with the same architecture as in BERT (Fig. 2.4) to generate contextual embeddings of all tokens of the input text. The decoder block is an autoregressive language model (Fig. 2.8) and sequentially predicts the next token in the target language. Each encoder block contains a multi-head self-attention for the current sequence of output tokens. By cross-attention the information from the input sequence is included. The calculations are repeated for all current input tokens and are very similar to the self-attention computations. The resulting vector is transformed by a fully connected layer yielding the embeddings of that layer

We denote the already translated tokens by $s_0, s_1, \ldots, s_{t-1}$ were $s_0$ is the token *"[BOS]"* indicating the beginning of the output text. The decoder first computes a self-attention for these tokens using the formula (2.4) as for BERT. As only part of the target tokens are covered and the rest is 'masked', this layer is called *masked multi-head self-attention* yielding intermediate contextual embeddings $\tilde{\boldsymbol{s}}_0, \tilde{\boldsymbol{s}}_1, \ldots, \tilde{\boldsymbol{s}}_{t-1}$ for the target tokens $s_0, s_1, \ldots, s_{t-1}$.

### Cross-Attention

Then the decoder performs a *cross-attention* $\text{CATL}(\tilde{\boldsymbol{V}}, \check{\boldsymbol{X}})$ with the input text embeddings of the highest encoder block (Fig. 2.12). Here the query-vectors are computed for the embeddings of the target tokens $\tilde{\boldsymbol{S}}_t = (\tilde{\boldsymbol{s}}_0, \tilde{\boldsymbol{s}}_1, \ldots, \tilde{\boldsymbol{s}}_{t-1})$ provided by the respective decoder block. The key and value vectors are computed for the embeddings $\check{\boldsymbol{X}} = \check{\boldsymbol{x}}_1, \ldots, \check{\boldsymbol{x}}_{T_{\text{src}}}$ of the last encoder block. Note that cross attention employs the same Eq. (2.4) with matrices $\boldsymbol{W}^{(q)}, \boldsymbol{W}^{(k)}, \boldsymbol{W}^{(v)}$ as the BERT self-attentions. This is done in parallel and called *multi-head cross-attention*. In this

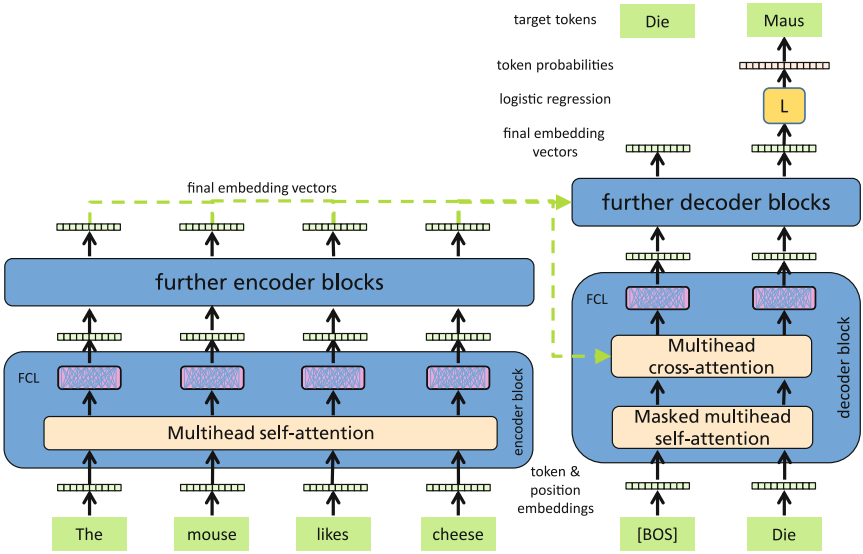


**Fig. 2.13** The transformer [141] uses an encoder with the same architecture as BERT to generate embeddings of all tokens of the input sentence. Each encoder block performs multi-head self-attention of the input sequence followed by a fully connected layer (FCL) . The decoder is similar to a GPT model and sequentially predicts the next token in the target language. Each encoder block contains a multi-head cross-attention including the final embeddings of the encoder. Using the last output embedding of the final decoder block, a logistic classifier $L$ predicts probabilities of the next token of the output sentence

way, information from the source text is taken into account. Subsequently, the embeddings computed by different heads are concatenated (2.6) and the result is transformed by a fully connected layer with ReLU activation (2.7). In addition, residual "bypass" connections are used as well as layer normalization [6] for regularization. The output of the fully connected layer yields a new 'output' embedding $\tilde{\boldsymbol{s}}_0, \ldots, \tilde{\boldsymbol{s}}_{t-1}$ for the target tokens $s_1, \ldots, s_{t-1}$. Together these layers are called a *decoder block* (Fig. 2.13).

The next decoder block gets the computed token output embeddings of the previous block as input and computes a new embedding of the target tokens $s_1, \ldots, s_{t-1}$. The decoder consists of several decoder blocks (6 in the original model). Using the output embedding $\check{\boldsymbol{s}}_{t-1}$ of the righmost token $s_{t-1}$ in the last decoder block, the token probabilities $p(S_t = s_t | s_1, \ldots, s_{t-1}, v_1, \ldots, v_{T_{\text{src}}})$ of the next token $s_t$ of the target text at position $t$ are predicted by a logistic classifier, e.g. for the token *"Maus"* in Fig. 2.13.

Note that for the prediction of a further token at position $t + 1$ the observed token $s_t$ is added to the computation (2.13) of the self-attentions in the decoder. Hence, the decoder embeddings change and all decoder computations have to be repeated. In this respect the model still works in a recursive way. Nevertheless, all

self-attentions and cross-attentions in each layer are computed in parallel. However, the computations for the encoder are only performed once.

Sequences of variable length are padded with a special token up to the maximal length. This is done for the input and the output sequence. If a sequence is very short, a lot of space is wasted. Therefore, the sequence length may be varied in different mini-batches called buckets in the training data.

The transformer has a large set of parameters. First it requires embeddings of the input and target token vocabularies. Then there are the $\boldsymbol{W}^{(q)}$, $\boldsymbol{W}^{(k)}$, $\boldsymbol{W}^{(v)}$ matrices for the multi-head self-attention, the masked multi-head self-attention and the multi-head cross-attention of the different heads and layers. In addition, the parameters of the fully connected networks and the final logistic classifier have to be specified. While the base model had an input sequence length of 512 and 65M parameters, the big model had an input sequence length of 1024 and 213M parameters [141]. The values of all these parameters are optimized during training.

The training data consists of pairs of an input sentence and the corresponding target sentence. Training aims to generate the target tokens with maximal probability for the given input tokens to maximize the joint conditional probability (2.13) of the output sequence by stochastic gradient descent. In our example in Fig. 2.13 for the given input text *"The mouse likes cheese"* the product of conditional probabilities of the output tokens *"Die Maus mag Käse"* has to be maximized. The original model [141], for instance, used 36M sentences of the WMT English-French benchmark data encoded as 32,000 wordpiece tokens. Both the encoder and decoder are trained simultaneously by stochastic gradient descent end-to-end, requiring 3.5 days with 8 GPUs.

Cross-attention is the central part of the transformer, where the information from the input sentence is related to the translated output sentence. In Fig. 2.14 a German input sentence is displayed together with its English translation. Both sentences are tokenized by byte-pair encoding, where the beginning of a word is indicated by "_". Below the strength of cross-attentions between the input tokens and output tokens is depicted for two different heads. Obviously the first input token "_*The*" has a special role.

### 2.3.2 Decoding a Translation to Generate the Words

After training, the Transformer is able to predict the probabilities of output tokens for an input sentence. For a practical translation, however, it is necessary to generate an explicit sequence of output tokens. Computing the output sequence with maximal probability is computationally hard, as then all output possible sequences have to be considered. Therefore, an approximate solution is obtained using greedy decoding or beam search.

**Greedy decoding** simply picks the most likely token with the highest probability at each decoding step until the end-of-sentence token is generated. The problem with this approach is that once the output is chosen at any time step $t$, it is impossible to

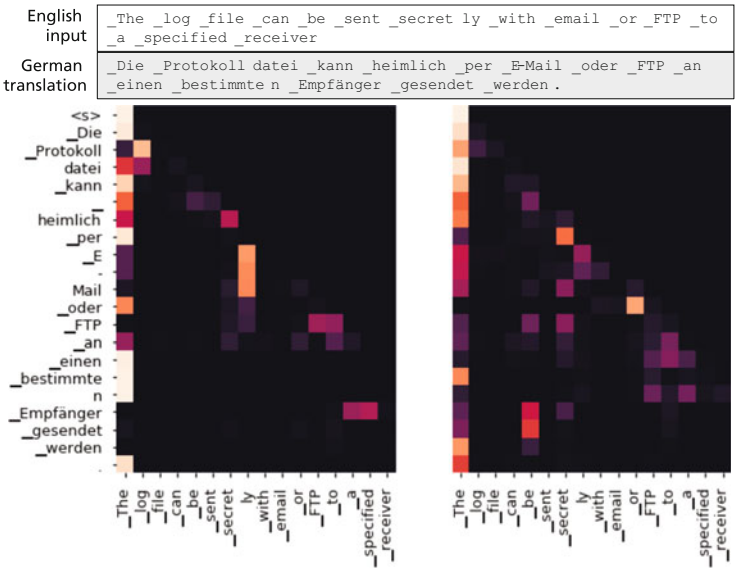


**Fig. 2.14** An English input sentence tokenized by Byte-Pair encoding and the translated tokenized German output sentence. Below are two cross-attention graphs from different heads of the 4-th decoder layer [126]. Dark values indicate a low cross-attention score. Image source: [126]

go back and change the selection. In practice there are often problems with greedy decoding, as the available probable continuation tokens may not fit to a previously assigned token. As the decision cannot be revised, this may lead to suboptimal generated translations.

**Beam search** [52] keeps a fixed number $k$ of possible translations $s_1, \ldots, s_t$ of growing length (Fig. 2.15). At each step each translation of length $t$ is enlarged by $k$ different tokens at position $t+1$ with the highest conditional probabilities $p(S_{t+1} = s_{t+1}|s_1, \ldots, s_t, v_1, \ldots, v_{T_{\text{src}}})$. From these $k*k$ token sequences only the $k$ sequences with largest total probabilities $p(s_1, \ldots, s_{t+1}|v_1, \ldots, v_{T_{\text{src}}})$ are retained. A complete translation (containing the end-of-sentence token) is added to the final candidate list. The algorithm then picks the translation with the highest probability (normalized by the number of target words) from this list. For $k = 1$ beam search reduces to greedy decoding. In practice, the translation quality obtained via beam search (size of 4) is significantly better than that obtained via greedy decoding. Larger beam sizes often lead to suboptimal solutions [31]. However, beam search is computationally very expensive (25%–50% slower depending on the base architecture and the beam size) in comparison to greedy decoding [29].

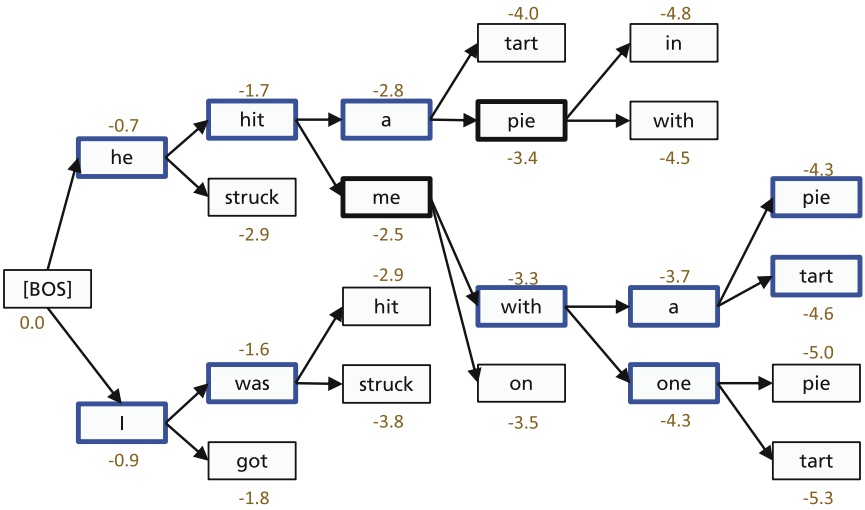


**Fig. 2.15** Beam search is a technique for decoding a language model and producing text. At every step, the algorithm keeps track of the $k$ most probable partial translations (bold margin). The score of each translation is equal to its log probability. The beam search continues until it reaches the end token for every branch [78]

### *2.3.3 Evaluation of a Translation*

Traditionally, evaluation is done by comparing one or more reference translations to the generated translation, as described in the survey [127]. There are a number of automatic evaluation metrics:

**BLEU** compares counts of 1-grams to 4-grams of tokens. The BLEU metric ranges from 0 to 1, where 1 means an identical output with the reference. Although BLEU correlates well with human judgment [110], it relies on precision alone and does not take into account recall—the proportion of the matched $n$-grams out of the total number of $n$-grams in the reference translation.

**ROUGE** [80] unlike BLEU is a recall-based measure and determines which fraction of the words or n-grams in the reference text appear in the generated text. It determines, among other things, the overlap of unigrams or bigrams as well as the longest common subsequence between a pair of texts. Different versions are used: ROUGE-1 measures the overlap of unigram (single words) between the pair of texts. ROUGE-2 determines the overlap of bigrams (two-words sequences) between the pair of texts. ROUGE-L: measures the length of the longest sequence of words (not necessarily consecutive, but still in order) that is shared between both texts. This length is divided by the number of words in the reference text.

**METEOR** [75] was proposed to address the deficits of BLEU. It performs a word-to-word alignment between the translation output and a given reference translation. The alignments are produced via a sequence of word-mapping modules. These

check, if the words are exactly the same, same after they are stemmed using the Porter stemmer, and if they are synonyms of each other. After obtaining the final alignment, METEOR computes an F-value, which is a parameterized harmonic mean of unigram precision and recall. METEOR has also demonstrated to have a high level of correlation with human judgment, often even better than BLEU.

**BERTscore** [164] takes into account synonyms and measures the similarity of embeddings between the translation and the reference. It computes the cosine similarity between all token embeddings of both texts. Then a greedy matching approach is used to determine assignments of tokens. The maximum assignment similarity is used as BERTscore.

For high-quality translations, however, there is a noticeable difference between human judgment and automatic evaluation. Therefore, most high-end comparisons today use human experts to assess the quality of translation and other text generation methods. Since the transformer was proposed by Vaswani et al. [141] in 2017, its variants were able to raise the SOTA in language translation performance, e.g. for translation on WMT2014 English-French from 37.5 to 46.4 BLEU score.

The transformer architecture was analyzed theoretically. Yun et al. [160, 161] showed that transformers are expressive enough to capture all continuous sequence to sequence functions with a compact domain. Pérez et al. [112] derived that the full transformer is Turing complete, i.e. can simulate a full Turing machine.

### *2.3.4 Pre-trained Language Models and Foundation Models*

A model *language model* either computes the joint probability or the conditional probability of natural language texts and potentially includes all information about the language. BERT is an *autoencoder* language models containing encoder blocks to generate contextual embeddings of tokens. GPT is an *autoregressive language models* which predicts the next token of a sequence and restricts self-attention to tokens which already have been generated. *Transformers* (or *Transformer encoder-decoders*) use a transformer encoder to convert the input text to contextual embeddings and generate the translated text with an autoregressive transformer decoder utilizing the encoder embeddings as inputs (Fig. 2.16). These models are the backbone of modern NLP and are collectively called *Pre-trained Language Models* (*PLM*).

All these models, especially BERT and GPT, are initialized via pre-training on a large corpus of text documents. During pre-training, parts of the input are hidden from the model, and the model is trained to reconstruct these parts. This has proven to be extremely effective in building strong representations of language and in finding parameter initializations for highly expressive NLP models that can be adapted to specific tasks. Finally, these models provide probability distributions over language that we can sample from.

Most network types have some built-in assumptions called *inductive bias*. Convolutional networks have local kernel functions that are shifted over the input matrix

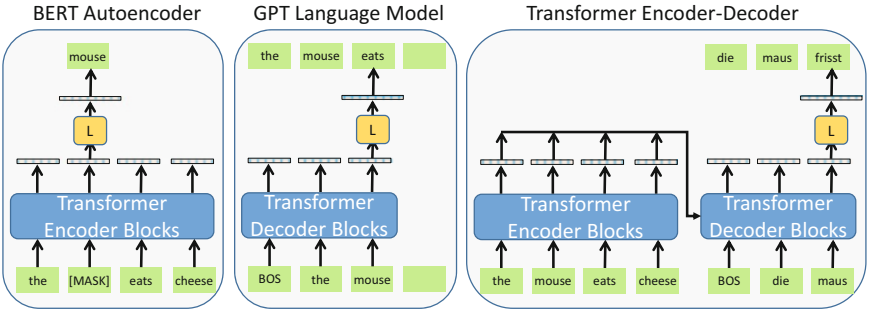


**Fig. 2.16** Autoencoders like BERT (left) and autoregressive LMs like GPT-2 (middle) use transformer blocks to generate contextual embeddings of tokens. The transformer (right) combines a transformer encoder and an autoregressive transformer decoder to produce a translation. All models predict the probability of tokens with a logistic classifier $L$. Collectively these models are called Pre-trained Language Models (PLMs)

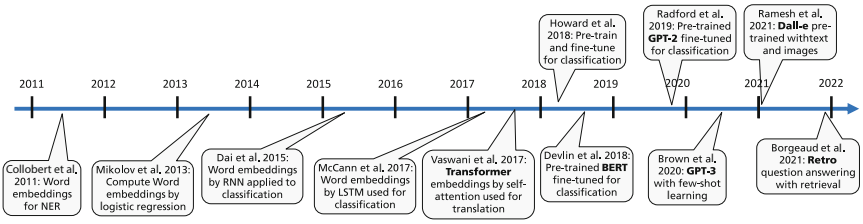


**Fig. 2.17** Timeline for the development of embeddings, pre-training and fine-tuning

and therefore have an inductive bias of translation invariance and locality. Recurrent networks apply the same network to each input position and have a temporal invariance and locality. The BERT architecture makes only few assumptions about the structural dependency in data. The GPT model is similar to the RNN as it assumes a Markovian structure of dependencies to the next token. As a consequence, PLMs often require more training data to learn the interactions between different data points, but can later represent these interactions more accurately than other model types.

Historically, learned embedding vectors were used as representations of words for downstream tasks (Fig. 2.17). As early as 2003 Bengio et al. [15] proposed a distributed vector representation of words to predict the next word by a recurrent model. In 2011 Collobert et al. [32] successfully employed word embeddings for part-of-speech tagging, chunking, named entity recognition, and semantic role labeling. In 2013 Mikolov et al. [93] derived their word embeddings using a logistic classifier. In 2015 Dai et al. [33] trained embeddings with an RNN language model in a self-supervised way and later applied it to text classification. In 2017 McCann et al. [87] pre-trained multilayer LSTMs for translation computing contextualized word vectors, which are later used for various classification tasks.

In the same year Vaswani et al. [141] developed the attention-only transformer for language translation. In 2018 Howard et al. [59] pre-trained a language model (ULMFiT), and demonstrated the effectiveness of fine-tuning to different target tasks by updating the full (pre-trained) model for each task. In the same year Howard et al. [116] used a pre-trained autoregressive part of the transformer [141] to solve a large number of text understanding problems by fine-tuned models. At the same time Devlin et al. [39] pre-trained the autoencoder using the masked language model objective and adapted this BERT model to many downstream tasks by fine-tuning. In 2019 Radford et al. [118] presented the GPT-2 language model, which was able to generate semantically convincing texts. Brown et al. [21] proposed the GPT-3 model, which could be instructed to solve NLP-tasks by a task description and some examples. In 2021 Ramesh et al. [121] applied language modeling to text and pictures and were able to create impressive pictures from textual descriptions. Borgeaud et al. [18] presented the Retro model that answers questions by retrieving information from a text collection of 2 trillion tokens and composes an answer in natural language.

Almost all state-of-the-art NLP models are now adapted from one of a few Pre-trained Language Models, such as BERT, GPT-2, T5, etc. PLMs are becoming larger and more powerful, leading to new breakthroughs and attracting more and more research attention. Due to the huge increase in performance, some research groups have suggested that large-scale PLMs should be called *Foundation Models*, as they constitute a 'foundational' breakthrough technology that can potentially impact many types of applications [17, p. 3]. In this book, we reserve the term 'Foundation Models' for large Pre-trained Language Models with more than a billion parameters, since these models are able of generating fluent text, can potentially handle different media, and can usually be instructed by prompts to perform specific tasks.

If one of these models is improved, this high degree of homogeneity can lead to immediate benefits for many NLP applications. On the other hand all systems could share the same problematic biases present in a few basic models. As we will see in later chapters PLM-based sequence modeling approaches are now applied to text (Sect. 2.2), speech (Sect. 7.1), images (Sect. 7.2), videos (Sect. 7.3), computer code (Sect. 6.5.6), and control (Sect. 7.4). These overarching capabilities of Foundation Models are depicted in Fig. 2.18.

The next Sect. 2.4 discusses some common techniques for optimizing and regularizing pre-trained language models. In addition, some approaches to modify the architecture of these networks are presented. In Chap. 3 we present a number of approaches to improve the capabilities of PLMs, especially by modifying the training tasks (Sect. 3.1.3). In the Chaps. 5–7 we discuss a number of applications of PLMs. Chapter 5 covers traditional NLP tasks like named entity recognition and relation extraction, where PLMs currently perform best. Most important applications of Foundation Models are on the one hand text generation and related tasks like question-answering and dialog systems, which are introduced in Chap. 6. On the other hand Foundation Models can simultaneously process different media and perform tasks like image captioning, object detection in images, image generation following a text description, video interpretation, or computer game control, which

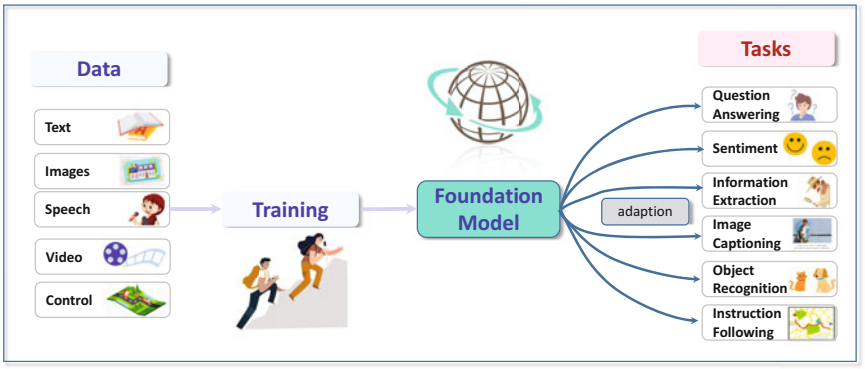


**Fig. 2.18** A Foundation Model can integrate the information in the data from different modalities. Subsequently it can be adapted, e.g. by fine-tuning, to a wide range of downstream tasks [17, p. 6]. Credits for image parts in Table A.1

are discussed in Chap. 7. Because of the potential social and societal consequences of such Foundation Models, it is particularly important that researchers in this field keep society's values and human rights in mind when developing and applying these models. These aspects are summarized in Sect. 8.2.

### Available Implementations

- The source code for many pre-trained language models (BERT, GPT, Transformers) as well as pre-trained models for different languages and text corpora can be downloaded from Hugging Face https://huggingface.co/transformers/, Fairseq https://github.com/pytorch/fairseq, TensorFlow https://www.tensorflow.org/ and PyTorch https://pytorch.org/. These toolkits also allow the flexible formulation of Deep Neural Networks and provide the automatic computation of gradients as well as optimization methods. All are able to execute computations in parallel and distribute them to different CPUs and Graphical Processing Units (GPUs).
- PLMs are getting larger than the memory of a single GPU and require to distribute training code among several GPUs. This is supported by libraries like FastSeq https://github.com/microsoft/fastseq, LightSeq https://github.com/bytedance/lightseq, and FastT5 https://github.com/Ki6an/fastT5.
- DeepSpeed [122] was used to train the MT-NLG autoregressive LM with 530B parameters (Sect. 3.1.2) https://github.com/microsoft/DeepSpeed.
- Ecco [2] https://github.com/jalammar/ecco and BertViz [144] https://github.com/jessevig/bertviz are tools to visualize the attentions and embeddings of PLMs.
- Transformers-interpret https://github.com/cdpierse/transformers-interpret is a model explainability tool designed for the Hugging Face package.
- Captum [70] is a library https://captum.ai/ to generate interpretations and explanations for the predictions of PyTorch models.

### 2.3.5 Summary

A transformer is a sequence-to-sequence model, which translates a source text of the input language into a target text in the target language. It consists of an encoder with the same architecture as an autoencoder BERT model that computes contextual embeddings of tokens of the source text. The decoder resembles an autoregressive GPT model and sequentially generates the tokens of the target text. Internally, contextual embeddings of the target tokens are computed in the different layers. Each decoder block has an additional cross-attention module in which the query vectors are taken from the embeddings of the target tokens and the key and value vectors are computed for the embeddings of the source tokens of the last layer. In this way, the information from the source text is communicated to the decoder. The embedding of the last token in the top layer is entered into a logistic classifier and this calculates the probability of the tokens for the next position. Subsequently, the observed token at the next position is appended to the target input and the computations are repeated for the next but one position.

During training the parameters of the transformer are adapted by stochastic gradient descent in such a way that the model assigns high probabilities to the observed target tokens of the translation in the training data. When the model has been trained on a large text dataset it can be applied for translation. Conditional on an input text, it can sequentially compute the probability of the next token of the translation.

During application of a trained model either the token with the maximal probability is selected or several alternatives are generated by beam search and the final output sequence with maximal probability is chosen. The evaluation of the translations quality is difficult as different translations may be correct. A number of metrics, e.g. BLEU, have been developed, which compare the machine translation to one or more reference translations by comparing the number of common word $n$-grams with $n = 1, \ldots, 4$. Often the results are assessed by human raters. The transformer was able to generate better translation than prior models. In the meantime the translation quality for a number of language pairs is on par with human translators.

In the previous sections, we discussed *autoencoder BERT* models, *autoregressive GPT* models and the *encoder-decoder Transformers*. Collectively these models are called *pre-trained language models*, as transfer learning with a pre-training step using a large training set and a subsequent fine-tuning step is a core approach for all three variants. The self-attention and cross-attention modules are central building blocks used by all three models. Despite the development of many variations in recent years, the original architecture developed by Vaswani et al. [141] is still commonly employed.

It turns out that these models can be applied not only to text, but to various types of sequences, such as images, speech, and videos. In addition, they may be instructed to perform various tasks by simple prompts. Therefore, large PLMs are also called *Foundation Models*, as they are expected to play a crucial role in the future development of text and multimedia systems.

## 2.4 Training and Assessment of Pre-trained Language Models

This section describes some techniques required to train and apply PLMs.

- We need *optimization techniques* which can process millions and billions of parameters and training examples.
- Specific *regularization* methods are required to train the models and to avoid overfitting.
- The *uncertainty* of model predictions has to be estimated to asses the performance of models.
- The *explanation* of model predictions can be very helpful for the acceptance of models.

Approaches to solving these problems are discussed in this section. PLMs are usually specified in one of the current Deep Learning frameworks. Most popular are *TensorFlow* provided from Google [137] and *PyTorch* from Meta [114]. Both are based on the Python programming language and include language elements to specify a network, train it in parallel on dedicated hardware, and to deploy it to different environments. A newcomer is the *JAX* framework [22], which is especially flexible for rapid experimentation. It has a compiler for linear algebra to accelerate computations for machine learning research.

### *2.4.1 Optimization of PLMs*

#### Basics of PLM Optimization

For the i.i.d. training sample $Tr = \{(\boldsymbol{x}^{[1]}, y^{[1]}), \ldots, (\boldsymbol{x}^{[N]}, y^{[N]})\}$ parameter optimization for Deep Neural Networks aims to find a model that minimizes the loss function $L(\boldsymbol{x}^{[i]}, y^{[i]}; \boldsymbol{w})$

$$\min_{\boldsymbol{w}} L(\boldsymbol{w}) = L(\boldsymbol{x}^{[1]}, y^{[1]}; \boldsymbol{w}) + \cdots + L(\boldsymbol{x}^{[N]}, y^{[N]}; \boldsymbol{w}). \tag{2.14}$$

First-order optimization methods, also known as gradient-based optimization, are based on first-order derivatives. A requirement is that the loss function $L(\boldsymbol{w})$ is smooth, i.e. is continuous and in addition differentiable at almost all parameter values $\boldsymbol{w} = (w_1, \ldots, w_k)$. Then the partial derivatives $\frac{\partial L(\boldsymbol{w})}{\partial w_j}$ of $L(\boldsymbol{w})$ with respect to any component $w_j$ of $\boldsymbol{w}$ can be computed at almost all points. The *gradient* of $L(\boldsymbol{w})$ in a specific point $\boldsymbol{w}$ is the vector

$$\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} = \left( \frac{\partial L(\boldsymbol{w})}{\partial w_1}, \ldots, \frac{\partial L(\boldsymbol{w})}{\partial w_k} \right)^{\mathsf{T}}. \tag{2.15}$$

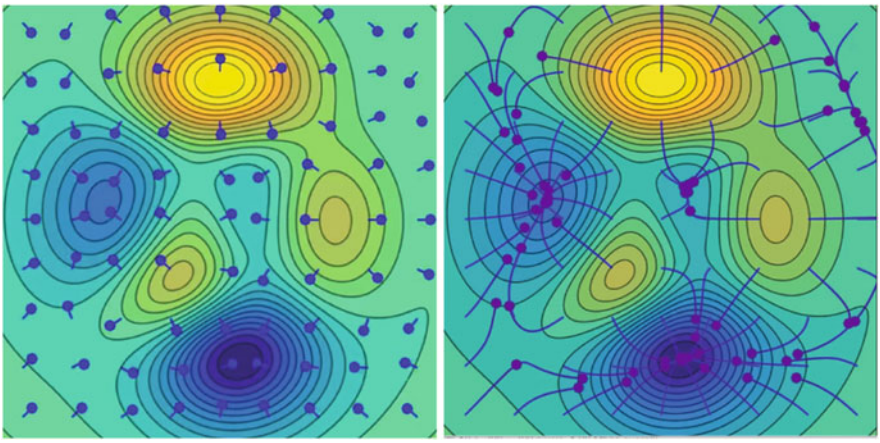

**Fig. 2.19** On all points of a grid the negative gradients are computed for this two-dimensional function $L(\boldsymbol{w})$ (left). The gradient descent algorithm follows the negative gradients and approaches the local minima (right). The blue lines are the paths taken during minimization. Image credits in Table A.1

The gradient points into the direction, where $L(\boldsymbol{w})$ in point $\boldsymbol{w}$ has its steepest ascent. Consequently, the direction of the steepest descent is in the opposite direction $-\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}}$. The batch *gradient descent* algorithm therefore changes the current parameter $\boldsymbol{w}_{(t)}$ in the direction of the negative gradient to get closer to the minimum

$$\boldsymbol{w}_{(t+1)} = \boldsymbol{w}_{(t)} - \lambda \frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}}. \tag{2.16}$$

The *learning rate* $\lambda$ determines the step-size or how much to move in each iteration until an optimal value is reached. As the gradient is usually different for each parameter $\boldsymbol{w}_{(t)}$ it has to be recomputed for every new parameter vector (Fig. 2.19). The iteration process is repeated until the derivative becomes close to zero. A zero gradient indicates a *local minimum* or a *saddle point* [51, p. 79]. In practical applications it is sufficient to repeat the optimization beginning with different $\boldsymbol{w}$-values and stop, if the derivative is close to zero.

Deep Neural Networks often require many millions of training examples. The repeated computation of the gradient for all these examples is extremely costly. The **Stochastic Gradient Descent** (*SGD*) algorithm does not use the entire dataset but rather computes the gradient only for a small *mini-batch* of $m$ training examples at a time. In general, a mini-batch has sizes $m$ ranging from 32 up to 1024, with even higher values for recent extremely large models. Subsequently, the parameters of the model are changed according to (2.16).

For each iteration a new mini-batch is selected randomly from the training data. According to the law of large numbers the gradients computed from these mini-

batches fluctuate around the true gradient for the whole training set. Therefore, the mini-batch gradient on average indicates an adequate direction for changing the parameters. Mertikopoulos et al. [91] show that by iteratively reducing the learning rate to 0, the SGD exhibits almost sure convergence, avoids spurious critical points such as saddle points (with probability 1), and stabilizes quickly at local minima. There are a number of variations of the SGD algorithm, which are described below [65].

An important step of optimization is the *initialization of parameters*. Their initial values can determine whether the algorithm converges at all and how fast the optimization approaches the optimum. To break symmetry, the initial parameters must be random. Furthermore, the mean and variance of the parameters in each layer are set such that the resulting outputs of the layer have a well-behaved distribution, e.g. expectation 0.0 and variance 1.0. In addition, all gradients also should have such a benign distribution to avoid exploding or vanishing gradients. All Deep Learning software frameworks contain suitable initialization routines. A thorough introduction is given by Goodfellow et al. [51, p. 292].

### Variants of Stochastic Gradient Descent

**Momentum** is a method that helps SGD to increase the rate of convergence in the relevant direction and reduce oscillations. Basically a moving average $\boldsymbol{u}_{(t)}$ of recent gradients with a parameter $\gamma \approx 0.9$ is computed and the parameter update is performed with this average by

$$\boldsymbol{u}_{(t)} = \gamma \boldsymbol{u}_{(t-1)} - \lambda \frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} \qquad \text{where} \qquad \boldsymbol{w}_{(t)} = \boldsymbol{w}_{(t-1)} - \boldsymbol{u}_{(t)}. \tag{2.17}$$

Note that in addition to the parameter vector $\boldsymbol{w}_{(t)}$ the moving average $\boldsymbol{u}_{(t)}$ of the same length has to be stored requiring the same memory as the parameter vector $\boldsymbol{w}$. This can consume a large additional memory size if the number of parameters approaches the billions. In recent years a number of further optimizers were developed [65]:

- **AdaGrad** adapts the learning rate dynamically based on the previous gradients. It uses smaller learning rates for features occurring often, and higher learning rates for features occurring rarely.
- **AdaDelta** modifies AdaGrad. Instead of accumulating all past gradients, it restricts the accumulation window of the past gradients to some fixed size $k$.
- **RMSProp** is also a method in which the learning rate is adapted for each of the parameters. The idea is to divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
- **Adam** combines the advantages of both AdaGrad and RMSProp. Adam is based on adaptive estimates of lower-order moments. It uses running averages of both the gradients and the second moments of the gradients.

Due to the extremely large number of parameters of PLMs second order optimization methods like *Conjugate Gradient* or *Quasi-Newton* are rarely employed. As the number of second order derivatives grows quadratically, only crude approximations may be used. An example is Adam, as described before.

An important architectural addition to PLMs to improve training are *residual connections*, which were proposed by Vaswani et al. [141] for the Transformer. Residual connections have been shown to be very successful for image classification networks such as ResNet [54] and allowed training networks with several hundred layers. The identity shortcuts skip blocks of layers to preserve features. Zhang et al. [163] analyze the representational power of networks containing residual connections.

### Parallel Training for Large Models

Recently, there have been suggestions to reduce the optimization effort by employing larger mini-batches. You et al. [159] propose the **LAMB optimizer** with layerwise adaptive learning rates to accelerate training of PLMs using large mini-batches. They prove the convergence of their approach to a stationary point in a general nonconvex setting. Their empirical results demonstrate the superior performance of LAMB. It is possible to reduce the BERT training time from 3 days to just 76 min with very little hyperparameter tuning and batch sizes of 32,868 without any degradation of performance. The LAMB program code is available online [97]. In addition, the memory requirements of the optimization may be reduced [119] to enable parallelization of models resulting in a higher training speed.

Large models such as GPT-3 have many billion parameters that no longer fit into the memory of a single computational device, e.g. a GPU. Therefore, the computations have to be distributed among several GPUs. There are different parallelization techniques [156]:

- *Data parallelism* assigns the same model code and parameters to each GPU but different training examples [72]. Gradients are computed in parallel and finally summarized.
- *Pipeline parallelism* partitions the model into different parts (e.g. layers) that are executed on different GPUs. If a part is computed it sends its results to the next GPU. This sequence is reversed in the backward pass of training.
- *Within-layer model parallelism* distributes the weights of a single layer across multiple GPUs.

The implementation of a parallelization strategy for a model is a tedious process. Support is given by the **DeepSpeed** library [122] that makes distributed training easy, efficient, and effective. Recently the **GSPMD** system [156] was developed which automates this process and is able to combine different parallelism paradigms in a unified way. GSPMD infers the distribution of computations to a network of GPUs based on limited user annotations to the model definition. It was, for instance, applied to distribute models with 1 trillion parameters on 2048 GPUs.

### 2.4.2 Regularization of Pre-trained Language Models

If a model contains too many parameters it can nearly perfectly adapt to the training data by optimization, reflecting nearly all details of the training data. During this *overfitting* the model learns the random variations expressed in the training data and deviates from the mean underlying distribution. Consequently, it has usually a lower performance on test data and a larger *generalization error*. To avoid this phenomenon, the representational capacity of the model has to be reduced by *regularization methods*, which often have the same effect as reducing the number of parameters. Well known approaches for Deep Learning models are the $L_2$ regularization and $L_1$ regularization penalizing large parameter values, or *Dropout* temporarily setting randomly selected hidden variables to 0. A survey of regularization strategies for Deep Neural Networks is given by Moradi et al. [96].

The training of PLMs is often non-trivial. One problem is the occurrence of vanishing or exploding gradients, which is connected to the problem of the vanishing or exploding variance of input values of different layers [55]. *Batch normalization* normalizes the values of the components of hidden units to mean 0.0 and variance 1.0 and thus reduces the variation of input values. For a mini-batch of training cases the component values are aggregated to compute a mean and variance, which are then used to normalize the input of that component on each training case [62]. It can be shown that batch normalization makes hidden representations increasingly orthogonal across layers of a Deep Neural Network [35].

In their paper on the Transformer, Vaswani et al. [141] use a variant called *layer normalization* [6] for regularization. The authors compute the mean and variance of the different components of hidden units for each training example and use this to normalize the input to mean 0.0 and variance 1.0. In addition, they apply dropout to the output of self-attention. Finally, they use *label smoothing* [133] where the loss function is reformulated such that the observed tokens are not certain but alternative tokens may be possible with a small probability. This is a form of regularization which makes optimization easier. The RMSNorm [162] is a variant of the layer normalization, which only normalizes the input by division with the root-mean-square error without shifting the mean. In experiments, it compares favorably with the layer normalization [101].

### 2.4.3 Neural Architecture Search

The structure of the self-attention block was manually designed, and it is not clear, whether it is optimal in all cases. Therefore, there are some approaches to generate the architecture of PLMs in an automatic way called *Neural Architecture Search* (*NAS*). A survey is provided by He et al. [56], who argue that currently the contributions of architecture search to NLP tasks are minor. Zöller [166] evaluate architecture search for machine learning models.

Wang et al. [149] propose an architecture search space with flexible encoder-decoder attentions and heterogeneous layers. The architecture search produces several transformer versions and finally concentrates on hardware restrictions to adapt the computations to processors at hand. The authors report a speedup of 3 and a size reduction factor of 3.7 with no performance loss. For relation classification Zhu et al. [165] design a comprehensive search space. They explore the search space by reinforcement learning strategy and yield models which have a better performance.

Architecture search may also be formulated as a ranking task. **RankNAS** [60] solves this by a series of binary classification problems. The authors investigate translation and language models. For translation the usual encoder-decoder is included in a super-net, where each of the $10^{23}$ subnetworks is a unique architecture. The importance of an architectural feature (e.g., the number of layers) is measured by the increase in the model error after permuting the feature. The authors use an evolutionary optimization strategy and evaluate their approach on translation (WMT2014 En-De). They get increases in BLEU-values at a fraction of cost of other approaches.

Recently differentiable architecture search has been developed, which embeds architecture search in a continuous search space and finds the optimal architecture by gradient descent. This leads to an efficient search process that is orders of magnitude faster than the discrete counterparts. This idea is applied by Fan et al. [43], who propose a gradient-based NAS algorithm for machine translation. They explore attention modules and recurrent units, automatically discovering architectures with better performances. The topology of the connection among different units is learned in an end-to-end manner. On a number of benchmarks they were able to improve the performance of the Transformer, e.g. from 28.8 to 30.1 BLEU scores for the WMT2014 English-to-German translation. There are other successful architecture search approaches for neural translation [130], named entity recognition [64], and image classification models [34, 147, 148], which may possibly be applied to other NLP tasks.

### *2.4.4 The Uncertainty of Model Predictions*

Variations in the outcome of a PLM can have two main sources:

- *Epistemic uncertainty* reflects our limited knowledge about the real world. The real world situation corresponding to the training set can change causing a distribution shift. Moreover, the collected documents can have biases or errors and cover unwanted types of content. It is clear that the structure of the real world and the PLM differ. Therefore, a PLM can only approximate the correct conditional probabilities of language. This type of uncertainty is often called *structural uncertainty* and is difficult to estimate.

- *Aleatoric uncertainty* is caused by random variations which can be assessed more easily. The training data is usually a sample of the underlying data in the population and therefore affected by the sampling variation. If a model is randomly re-initialized, it generates a completely different set of parameter values which leads to different predictions. Finally, language models predict probabilities of tokens and the generation of new tokens is also affected by uncertainty. The Bayesian framework offers a well-founded tool to assess this type of uncertainty in Deep Learning [44].

A recent survey of methods for estimating the model uncertainty is provided by Gawlikowski et al.[47]. We will describe three approaches to capture model uncertainty: Bayesian statistics, a Dirichlet distributions, and ensemble distributions.

### Bayesian Neural Networks

*Bayesian Neural Networks* directly represent the uncertainty of the estimated parameters $\boldsymbol{w} = (w_1, \ldots, w_{d_w})$ by the *posterior distribution*

$$p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{Y}) \propto p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})p(\boldsymbol{w}). \tag{2.18}$$

Here $\boldsymbol{X}$ and $\boldsymbol{Y}$ are the observed inputs and outputs in the training set and $p(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{w})$ is the *likelihood*, i.e. the probability of the outputs given $\boldsymbol{X}$ and a parameter vector $\boldsymbol{w}$. The *prior distribution* $p(\boldsymbol{w})$ describes the distribution of parameters before data is available. The distribution of predictions for a new input $\tilde{\boldsymbol{x}}$ is given by

$$p(\tilde{\boldsymbol{y}}|\tilde{\boldsymbol{x}}, \boldsymbol{X}, \boldsymbol{Y}) = \int p(\tilde{\boldsymbol{y}}|\tilde{\boldsymbol{x}}, \boldsymbol{w})p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{Y})d\boldsymbol{w}. \tag{2.19}$$

The integral usually cannot be solved analytically and has to be approximated. Often a *Monte Carlo* approximation is used, which infers the integral by a sum over different parameter values $\boldsymbol{w}^{[i]}$ distributed according to the posterior distribution $p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{Y})$. If $\tilde{\boldsymbol{y}}^{[i]} = f(\tilde{\boldsymbol{x}}, \boldsymbol{w}^{[i]})$ is a deterministic network predicting the output for a parameter $\boldsymbol{w}^{[i]}$ and input $\tilde{\boldsymbol{x}}$, the resulting sample $\tilde{\boldsymbol{y}}^{[1]}, \ldots, \tilde{\boldsymbol{y}}^{[k]}$ can be considered as a sample of the output distribution $p(\tilde{\boldsymbol{y}}|\tilde{\boldsymbol{x}}, \boldsymbol{X}, \boldsymbol{Y})$ [108].

Bayesian predictive distributions can be approximated in different ways:

- *Sampling approaches* use a *Markov Chain Monte Carlo* algorithm to generate parameter values distributed according to the posterior distributions, from which realizations can be sampled [102]. Markov Chain Monte Carlo defines a sampling strategy, where first a new parameter value $\boldsymbol{w}$ is randomly generated and then the algorithm computes the probability to accept $\boldsymbol{w}$, or to keep the previous parameter value. Welling et al. [150] combined this approach with stochastic gradient descent and demonstrated that Bayesian inference on Deep Neural Networks can be done by a noisy SGD. A review of the favorable convergence properties has

been given by Nemeth et al. [103]. Practical evaluations of this technique are performed by Wenzel et al. [152].

- *Variational inference* approximates the posterior distribution by a product $q(\boldsymbol{w})$ of simpler distributions, which are easier to evaluate [9]. Using multiple GPUs and practical tricks, such as data augmentation, momentum initialization and learning rate scheduling, and learning rate scheduling, Osawa et al. [105] demonstrated that variational inference can be scaled up to ImageNet size datasets and architectures.

  It can be shown [45] that dropout regularization (Sect. 2.4.2) can be considered as approximate variational inference. Hence, the predictive uncertainty can be estimated by employing dropout not only during training, but also at test time. A variant called *Drop connect* randomly removes incoming activations of a node, instead of dropping an activation for all following nodes. This approach yields a more reliable uncertainty estimate and can even be combined with the original dropout technique [88].
- *Laplace approximation* considers the logarithm of the posterior distribution around a local mode $\hat{\boldsymbol{w}}$ and approximate it by a normal distribution $N(\hat{\boldsymbol{w}}, [H + \beta I]^{-1})$ over the network weights [9]. $H$ is the Hessian, the matrix of second derivatives, of $\log p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{Y})$. This approximation may be computed for already trained networks and can be applied to Deep Neural Networks [76]. A problem is the large number of coefficients of $H$, which limits the computations to elements on the diagonal. Extensions have been proposed by George et al. [48].

### Estimating Uncertainty by a Single Deterministic Model

Most PLMs predict tokens by a discrete probability distribution. If the softmax function is used to compute these probabilities, the optimization over the training set usually leads to very extreme probabilities close to 0 or 1. The network is often overconfident and generates inaccurate uncertainty estimates. To assess uncertainty, the difference between the estimated distribution and the actual distribution has to be described. If $v_1, \ldots, v_{d_v}$ is the vocabulary of tokens and $\boldsymbol{\pi}$ a discrete distribution over these tokens, then we can use the *Dirichlet distribution* $p(\boldsymbol{\pi}|\boldsymbol{\alpha}(\boldsymbol{x}))$ to characterize a distribution over these discrete distributions. The vector $\boldsymbol{\alpha}$ depends on the input $\boldsymbol{x}$ and has a component $\alpha_i$ for each $v_i$. The sum $\sum_i \alpha_i$ characterizes the variance. If it gets larger, the estimate for the probability of $v_i$ has a lower variance.

Malinin et al. [85] use the expected divergence between the empirical distribution and the predicted distribution to estimate the $p(\boldsymbol{\pi}|\boldsymbol{\alpha}(\boldsymbol{x}))$ for a given input $\boldsymbol{x}$. In the region of the training data the network is trained to minimize the expected *Kullback-Leibler* (*KL*) divergence between the predictions of in-distribution data and a low-variance Dirichlet distribution. In the region of out-of-distribution data a Dirichlet distribution with a higher variance is estimated. The distribution over the outputs can be interpreted as a quantification of the model uncertainty, trying to emulate the behavior of a Bayesian modeling of the network parameters [44].

Liu et al. [83] argue that the distance between training data elements is relevant for prediction uncertainty. To avoid that the layers of a network cause a high distortion of the distances of the input space, the authors propose a spectral normalization. This **SNGP** approach limits the distance $\|h(\boldsymbol{x}^{[1]}) - h(\boldsymbol{x}^{[2]})\|$ compared to $\|\boldsymbol{x}^{[1]} - \boldsymbol{x}^{[2]}\|$, where $\boldsymbol{x}^{[1]}$ and $\boldsymbol{x}^{[2]}$ are two inputs and $h(\boldsymbol{x})$ is a deep feature extractor. Then they pass $h(\boldsymbol{x})$ into a distance-aware *Gaussian Process* output layer. The Gaussian Process posterior is approximated by a Laplace approximation, which can be predicted by a deterministic Deep Neural Network.

The authors evaluate SNGP on $\text{BERT}_{\text{BASE}}$ to decide, if a natural utterance input is covered by the training data (so that it can be handled by the model) or outside. The model is only trained on in-domain data, and their predictive accuracy is evaluated on in-domain and out-of-domain data. While ensemble techniques have a slightly higher prediction accuracy, SNGP has a better calibration of probabilities and out-of-distribution detection. An implementation of the approach is available [138].

A number of alternative approaches are described in [47, p. 10f], which also discuss mixtures of Dirichlet distributions to characterize predictive uncertainty. In general single deterministic methods are computational less demanding in training and evaluation compared to other approaches. However, they rely on a single network configuration and may be very sensitive to the underlying network structure and the training data.

#### Representing the Predictive Distribution by Ensembles

It is possible to emulate the sampling variability of a training set by resampling methods. A well-founded approach is *bagging*, where $n_b$ samples of size $n$ are drawn with replacement from a training set of $n$ elements [20, 107]. For the $i$-th sample a model may be trained yielding a parameter $\hat{\boldsymbol{w}}^{[i]}$. Then the distribution of predictions $f(\boldsymbol{x}, \hat{\boldsymbol{w}}^{[i]})$ represent the uncertainty in the model prediction for an input $\boldsymbol{x}$, and it can be shown that their mean value $\frac{1}{n_b} \sum_i f(\boldsymbol{x}, \hat{\boldsymbol{w}}^{[i]})$ has a lower variance than the original model prediction [73]. In contrast to many approximate methods, ensemble approaches may take into account different local maxima of the likelihood function and may cover different network architectures. There are other methods to introduce data variation, e.g. random parameter initialization or random data augmentation. A survey on ensemble methods is provided by Dong et al. [40].

Besides the improvement in the accuracy, ensembles are widely used for representing prediction uncertainty of Deep Neural Networks [73]. In empirical investigations, the approach was at least as reliable as Bayesian approaches (Monte Carlo Dropout, Probabilistic Backpropagation) [73]. Reordering the training data and a random parameter initialization induces enough variability in the models for the prediction of uncertainty, while bagging may reduce the reliability of uncertainty estimation [77]. Compared to Monte Carlo Dropout, ensembles yield more reliable and better calibrated prediction uncertainties and are applicable to real-world training data [13, 53]. Already for a relatively small ensemble size of

five, deep ensembles seem to perform best and are more robust to data set shifts than the compared methods [106].

Although PLMs have been adapted as a standard solution for most NLP tasks, the majority of existing models is unable to estimate the uncertainty associated with their predictions. This seems to be mainly caused by the high computational effort of uncertainty estimation approaches. In addition, the concept of uncertainty of a predicted probability distribution is difficult to communicate. However, it is extremely important to get a diagnosis, when a PLM is given an input outside the support of its training data, as then the predictions get unreliable.

Among the discussed approaches the ensemble methods seem to be most reliable. However, they require a very high computational effort. New algorithms like SNGP are very promising. More research is needed to reduce this effort or develop alternative approaches. Recently benchmark repositories and datasets have been developed to provide high-quality implementations of standard and SOTA methods and describe best practices for uncertainty and robustness benchmarking [99].

**Implementations**
Uncertainty Baselines [10, 98] provide a collection high-quality implementations of standard and state-of-the-art methods for uncertainty assessment.

### *2.4.5 Explaining Model Predictions*

PLMs such as BERT are considered as black box models, as it is hard to understand, what they really learn and what determines their outputs. Hence, a lot of research goes into investigating the behavior of these models. There are three main reasons to explain the model predictions. *Trust* in the model predictions is needed, i.e. that the model generates reliable answers for the problem at hand and can be deployed in real-world applications. *Causality* asserts that the change of input attributes leads to sensible changes in the model predictions. *Understanding* of the model enables domain experts to compare the model prediction to the existing domain knowledge. This is a prerequisite for the ability to adjust the prediction model by incorporating domain knowledge.

Explanations can also be used to debug a model. A striking example was an image classification, where a horse was not detected by its shape, but by a label in the image [74]. Explanations are most important for critical decisions that involve humans or can cause high damage. Examples are health care, the judicial system, banking, or self-driving cars.

Explanation methods roughly can be grouped into local explanations or global explanations. A local explanation provides information or justification for the model's prediction for a specific input $\boldsymbol{x}$, whereas global explanations cover the model in general. A large majority of models aims at local explanations, as these may be used to justify specific predictions. Surveys on methods for the explanation of PLMs are provided by Danilevsky et al. [36], Burkart and Huber [23], Xu et al.

[155], Bauckhage et al. [11], Tjoa and Guan [139], and Belle and Papantonis [12]. Molnar [95] devotes a whole book to this topic and Bommasani et al. [17, p. 125] provide a recent overview. For language models different types of explanation can be used:

- **Feature importance** measures the influence of single input features, e.g. tokens, on the prediction. It often corresponds to the first derivative of a feature with respect to the output [79]. As the meaning of input tokens is easily understood, this type of explanation is readily interpretable by humans.
- **Counterfactual explanations** investigate, how an input $\boldsymbol{x}$ has to be modified, to generate a different target output.
- **Surrogate models** explain model predictions by a second, simpler model. One well-known example is *LIME* [123], which trains a local linear model around a single input $\boldsymbol{x}$ of interest.
- **Example-driven** explanations illustrate the prediction of an input $\boldsymbol{x}$ by selecting other labeled instances that are semantically similar to $\boldsymbol{x}$. This is close to the nearest neighbor approach to prediction and has, for instance, been used for text classification [1].
- **Source citation** is a general practice of scientific work in which a claim is supported by citing respectable scientific sources. The same can be done for a text generated by language models with a retrieval component [57].

Other approaches like a sequence of reasoning steps or rule invocations are unusable for PLMs with many millions of parameters.

The self-attention mechanism is the central function unit of PLMs. **BertViz** [144] is a visualization tool that allows users to explore the strength of attention between different tokens for the heads and layers in a PLM and allows users to get a quick overview of relevant attention heads. However, Jain et al. [63] demonstrate that attention does not correlate with feature importance methods and counterfactual changes of attention do not lead to corresponding changes in prediction. This may, for instance, be caused by the concatenation of head outputs and their subsequent processing by a fully connected nonlinear layer. Attentions are noisy predictors of the overall importance of components, but are not good at identifying the importance of features [129].

### Linear Local Approximations

An important concept is the contribution of input $x_i$ towards an output $y_j$, e.g. a class probability. Gradient-based explanations estimate the contribution of input $x_i$ towards an output $y_j$, e.g. a class probability, by computing the partial derivative $\partial y_j / \partial x_i$. This derivative is often called *saliency* and can be interpreted as linear approximation to the prediction function at input $\boldsymbol{x}$. **LIME** [123] defines a local linear regression model around a single input $\boldsymbol{x}$. Because of correlation of features, the coefficients of the input features depend on the presence or absence of the other input features. The **SHAP** approach therefore determines the influence of a feature

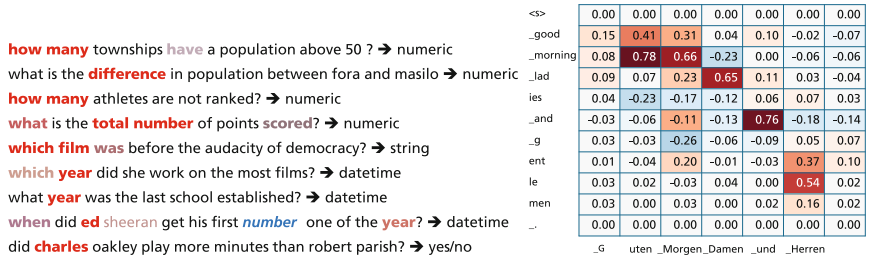


**Fig. 2.20** Contributions for the question classification task (left). Red marks positive influence, blue negative, and black tokens are neutral. Contributions for the task of translating *"good morning ladies and gentlemen"* to the German *"Guten Morgen Damen und Herren"* are shown on the right side [132]. Words are tokenized to word pieces

by the average influence of the feature for all combinations of other features [84]. The authors show the favorable theoretical properties of this approach and derive several efficient computation strategies.

### Nonlinear Local Approximations

Sundararajan et al. [132] formulate two basic requirements for this type of explanation. *Sensitivity*: if the inputs $\boldsymbol{x}^{[1]}$ and $\boldsymbol{x}^{[2]}$ differ in just one feature and lead to different predictions, then the differing feature should be given a non-zero contribution. *Implementation invariance*: i.e., the attributions are always identical for two functionally equivalent networks. As the prediction functions are usually nonlinear, gradient-based methods violate both requirements and may focus on irrelevant attributes.

**Integrated Gradients** [132] generates an approximation to the prediction function $F : \mathbb{R}^n \rightarrow [0, 1]$, which captures nonlinear dependencies. To assess the difference from baseline input $\boldsymbol{x}^{[1]}$ to another input $\boldsymbol{x}^{[2]}$, the authors compute the mean value of gradients $\partial F(\boldsymbol{x})/\partial \boldsymbol{x}$ of the output with respect to inputs along the line from $\boldsymbol{x}^{[1]}$ to $\boldsymbol{x}^{[2]}$ by an integral. It can be shown that this approach meets the above requirements. The authors apply the approach to question classification according to the type of the answer (Fig. 2.20). The baseline input is the all zero embedding vector. Another application considers neural machine translation. Here the output probability of every output token is attributed to the input tokens. As baseline all tokens were zeroed except the start and end markers. A similar analysis is based on a Taylor expansion of the prediction function [7] .

Liu et al. [82] propose a generative explanation framework which simultaneously learns to make classification decisions and generate fine-grained explanations for them. In order to reach a good connection between classification and explanation they introduce a classifier that is trained on their explanation. For product reviews they, for instance, generate the following positive explanations *"excellent picture,*

*attractive glass-backed screen, hdr10 and dolby vision*" and negative reasons "*very expensive*". The authors introduce an explanation factor, which represents the distance between the probabilities of the classifier trained on the explanations vs. the classifier trained on the original input and the gold labels. They optimize their models with minimum risk training.

### Explanation by Retrieval

Recently, Deep Learning models have been playing an increasingly important role in science and technology. The algorithms developed by Facebook are able to predict user preferences better than any psychologist [24, 71]. AlphaFold, developed by DeepMind, makes the most accurate predictions of protein structures based on their amino acids [131]. And the PaLM and Retro models are capable of generating stories in fluent English, the latter with the knowledge of the Internet as background. However, none of the programs were actually able to justify their decisions and cannot indicate why a particular sequence was generated or on what information a decision was based on.

In 2008, Anderson [5] predicted the end of theory-based science. In his view, theories are an oversimplification of reality, and the vast amount of accumulated data contains knowledge in a much more detailed form, so theories are no longer necessary. This is also the problem of *Explainable AI*, which aims to explain the decisions of Deep Learning models. It is always faced with a trade-off where predictive accuracy must be sacrificed in order to interpret the model output.

As large autoregressive language models are combined with retrieval components, document retrieval can be used not only to incorporate more accurate knowledge into the language generation process, but also to support the generated answers by authoritative citations. Metzler et al. [92] argues that future PLMs should justify created text by referring to supporting documents in the training data or background document collection. To implement this approach Nakano et al. [100] combine *GPT-3* with the search engine *BING* to enhance language generation for question-answering by retrieved documents. Their **WebGPT** [100] first creates a text in natural language (Sect. 6.2.3). After that, it enhances the generated sentences by different references to the found documents, similar to the way a scientist expands his texts by references. By this procedure WebGPT is able to justify and explain the created answer. This could be a way to make the generated text more trustworthy. Note that the advanced dialog model **LaMDA** can include links to external documents supporting an answer (Sect. 6.6.3).

### Explanation by Generating a Chain of Thought

Large autoregressive PLMs like GPT-3 are able to produce a very convincing continuation of a start text, and, for instance, generate the answer for a question. It turned out that their ability to generate the correct answer could drastically be

**Example Chain of Thoughts**: I will explain how to make logical inferences:
(1) John wakes to the sound of his children giggling outside the door. They bust in and say 'Happy Father's Day!' with a plate full of food. 2022 has been a great year. Can we infer that John was born before the inauguration of Barack Obama?
Answer: We can infer that John is a father because of 'his children'. We can infer that the current year is 2022. Barack Obama was inaugurated in 2009, so it has been about 13 years since Obama was inaugurated. John's children are old enough to speak, so if John were born after Obama's inauguration, he would have been younger than 10 when he had children. So, we can infer that John was born before Obama's inauguration. The answer is 'yes'.
(2) Sam sees a piano …

**Input Query**: Jennifer looked out her window and sees a really cool cloud below her. She unbuckles her seatbelt and heads to the bathroom. Is Jennifer probably traveling more than 300 miles per hour relative to the earth?

**Model Output**: 300 miles per hour is about 480 km/h. This is about the speed of a commercial airplane. Clouds are usually below airplanes, so Jennifer is probably on an airplane. The answer is 'yes'.

**Fig. 2.21** Explaining by a chain of thoughts. The first box contains two examples of thought chains, which are used for every query. This chain-of-thought prompt was input to the PaLM model together with the input query, and the model output was generated by PaLM [30, p. 38]

improved by giving a few examples with a chain of thought (Sect. 3.6.4) for deriving the correct answer. This has been demonstrated for the PaLM language model [30].

A generated *thought chain* can be used for other purposes. First, it can be checked whether the model produces the correct answer for the "right reasons", rather than just exploiting superficial statistical correlations. In addition, the explanation can potentially be shown to an end-user of the system to increase or decrease their confidence in a given prediction. Finally, for some queries (e.g., explaining a joke), the explanation itself is the desired output [30].

Figure 2.21 contains a few-shot query and the resulting answer. For application only a few example chains of thought are necessary, which can be reused. To generate the best answer for the question greedy decoding has to be used, yielding the optimal prediction. As PaLM shows, the enumeration of argument steps works empirically. However, a sound theory of how models actually use such arguments internally is still lacking. Further, it is not known under which circumstances the derivation of such a chain of thoughts succeeds. It should be investigated to what extent the reasoning of a model corresponds to the reasoning steps performed by humans.

**Implementations**
Ecco [2] and BertViz [143] are tools to visualize the attentions and embeddings of PLMs. An implementation and a tutorial on integrated gradients is available for TensorFlow [136]. Captum [26, 70] is an open-source library to generate interpretations and explanations for the predictions of PyTorch models containing most of the approaches discussed above. Transformers-interpret [113] is an alternative open-source model explainability tool for the Hugging Face package.

### *2.4.6 Summary*

Similar to other large neural networks, PLMs are optimized with simple stochastic gradient descent optimizers that are able to approach the region of minimal cost even for huge models with billions of parameters and terabytes of training data. This requires parallel training on computing networks which can be controlled by suitable software libraries. There are many recipes in the literature for setting hyperparameters such as batch size and learning rate schedules. Important ingredients are residual connections to be able to optimize networks with many layers and regularization modules to keep parameters in a manageable range.

Neural architecture search is a way to improve performance and reduce memory requirements of networks. A number of approaches have been proposed that significantly speed up training. Some methods provide models with better performance and lower memory footprint. There are new differential methods that have the potential to derive better architectures with little effort.

PLMs aim to capture relations between language concepts and can only do so approximately. Therefore, it is important to evaluate their inherent uncertainty. Three different approaches to analyze the uncertainty are described. Among these, ensemble methods appear to be the most reliable, but involve a high computational cost. New algorithms such as SNGP, which are based on a single model, are very promising.

To enable a user to decide whether a model result makes sense, it is necessary to explain how the result was obtained. Explanations can be provided by showing the importance of features for a result, by exploring the PLM by related examples or by approximating the PLM with a simple model. Some libraries are available that allow routine use of these methods. A new way of explaining texts generated by PLMs is to enhance the texts with appropriate citations of relevant supporting documents. Finally, a PLM can be instructed by chain-of-thought prompts to provide an explanation for the model response. This type of explanation is particularly easy to understand and can reflect the essential parts of a chain of arguments.

The next chapter discusses approaches to improve the three basic PLM types by new pre-training tasks or architectural changes. The fourth chapter examines the knowledge, which can be acquired by PLMs and that can be used to interpret text and to generate new texts.

# References


1. A. Abujabal, R. S. Roy, M. Yahya, and G. Weikum. "Quint: Interpretable Question Answering over Knowledge Bases". In: *Proc. 2017 Conf. Empir. Methods Nat. Lang. Process. Syst. Demonstr*. 2017, pp. 61–66.
2. J. Alammar. "Ecco: An Open Source Library for the Explainability of Transformer Language Models". In: *Proc. 59th Annu. Meet. Assoc. Comput. Linguist. 11th Int. Jt. Conf. Nat. Lang. Process. Syst. Demonstr*. 2021, pp. 249–257. URL: https://github.com/jalammar/ecco.
3. J. Alammar. *The Illustrated GPT-2 (Visualizing Transformer Language Models)*. Oct. 12, 2019. URL: http://jalammar.github.io/illustrated-gpt2/ (visited on 01/24/2021).
4. F. St-Amant. *How to Fine-Tune GPT-2 for Text Generation*. Medium. May 8, 2021. URL: https://towardsdatascience.com/how-to-fine-tune-gpt-2-for-text-generation-ae2ea53bc272 (visited on 07/29/2021).
5. C. Anderson. "The End of Theory: The Data Deluge Makes the Scientific Method Obsolete". In: *Wired* (June 23, 2008). ISSN: 1059–1028. URL: https://www.wired.com/2008/06/pb-theory/ (visited on 01/11/2022).
6. J. L. Ba, J. R. Kiros, and G. E. Hinton. "Layer Normalization". 2016. arXiv: 1607.06450.
7. S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek. "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation". In: *PloS one* 10.7 (2015), e0130140.
8. D. Bahdanau, K. Cho, and Y. Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". 2014. arXiv: 1409.0473.
9. D. Barber and C. M. Bishop. "Ensemble Learning in Bayesian Neural Networks". In: *Nato ASI Ser. F Comput. Syst. Sci*. 168 (1998), pp. 215–238.
10. baselines. *Uncertainty Baselines*. Google, Dec. 5, 2021. URL: https://github.com/google/uncertainty-baselines (visited on 12/06/2021).
11. C. Bauckhage, J. Fürnkranz, and G. Paass. "Vertrauenswürdiges, Transparentes Und Robustes Maschinelles Lernen". In: *Handbuch Der Künstlichen Intelligenz*. de Gruyter, 2021. ISBN: 978-3-11-065984-9.
12. V. Belle and I. Papantonis. "Principles and Practice of Explainable Machine Learning". In: *Front. Big Data* 4 (2021), p. 39. ISSN: 2624-909X. https://doi.org/10.3389/fdata.2021.688969.
13. W. H. Beluch, T. Genewein, A. Nürnberger, and J. M. Köhler. "The Power of Ensembles for Active Learning in Image Classification". In: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit*. 2018, pp. 9368–9377.
14. Y. Bengio, A. Courville, and P. Vincent. "Representation Learning: A Review and New Perspectives". In: *IEEE Trans. Pattern Anal. Mach. Intell*. 35.8 (2013), pp. 1798–1828.
15. Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. "A Neural Probabilistic Language Model". In: *J. Mach. Learn. Res*. 3 (Feb 2003), pp. 1137–1155.
16. Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. "Greedy Layer-Wise Training of Deep Networks". In: *Adv. Neural Inf. Process. Syst*. 19 (2006).
17. R. Bommasani et al. "On the Opportunities and Risks of Foundation Models". 2021. arXiv: 2108.07258.
18. S. Borgeaud et al. "Improving Language Models by Retrieving from Trillions of Tokens". Dec. 8, 2021. arXiv: 2112.04426 [cs].
19. G. Branwen. "GPT-2 Neural Network Poetry". In: (Mar. 3, 2019). URL: https://www.gwern.net/GPT-2 (visited on 01/27/2021).
20. L. Breiman. "Bagging Predictors". In: *Mach. Learn*. 24.2 (1996), pp. 123–140.
21. T. B. Brown et al. "Language Models Are Few-Shot Learners". 2020. arXiv: 2005.14165.
22. D. Budden and M. Hessel. *Using JAX to Accelerate Our Research*. Dec. 4, 2020. URL: https://www.deepmind.com/blog/using-jax-to-accelerate-our-research (visited on 06/21/2022).
23. N. Burkart and M. F. Huber. "A Survey on the Explainability of Supervised Machine Learning". In: *J. Artif. Intell. Res*. 70 (2021), pp. 245–317.

24. C. Cadwalladr and E. Graham-Harrison. “How Cambridge Analytica Turned Facebook ‘Likes’ into a Lucrative Political Tool”. In: *Guard. 17032018* (2018).
25. X. Cai, J. Huang, Y. Bian, and K. Church. “Isotropy in the Contextual Embedding Space: Clusters and Manifolds”. In: *Int. Conf. Learn. Represent*. 2020.
26. Captum. *Captum · Model Interpretability for PyTorch*. 2021. URL: https://captum.ai/ (visited on 12/06/2021).
27. S. Chaudhari, V. Mithal, G. Polatkan, and R. Ramanath. “An Attentive Survey of Attention Models”. In: *ACM Trans. Intell. Syst. Technol. TIST* 12.5 (2021), pp. 1–32.
28. S. F. Chen, D. Beeferman, and R. Rosenfeld. “Evaluation Metrics for Language Models”. In: (1998). URL: https://kilthub.cmu.edu/articles/EvaluationMetricsForLanguageModels/6605324/files/12095765.pdf.
29. Y. Chen, V. O. Li, K. Cho, and S. R. Bowman. “A Stable and Effective Learning Strategy for Trainable Greedy Decoding”. 2018. arXiv: 1804.07915.
30. A. Chowdhery et al. “PaLM: Scaling Language Modeling with Pathways”. Apr. 5, 2022. arXiv: 2204.02311 [cs].
31. E. Cohen and C. Beck. “Empirical Analysis of Beam Search Performance Degradation in Neural Sequence Models”. In: *Int. Conf. Mach. Learn*. PMLR, 2019, pp. 1290–1299.
32. R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. “Natural Language Processing (Almost) from Scratch”. In: *J. Mach. Learn. Res*. 12 (2011), pp. 2493–2537.
33. A. M. Dai and Q. V. Le. “Semi-Supervised Sequence Learning”. In: *Adv. Neural Inf. Process. Syst*. 2015, pp. 3079–3087.
34. Z. Dai, H. Liu, Q. V. Le, and M. Tan. “CoAtNet: Marrying Convolution and Attention for All Data Sizes”. Sept. 15, 2021. arXiv: 2106.04803 [cs].
35. H. Daneshmand, A. Joudaki, and F. Bach. “Batch Normalization Orthogonalizes Representations in Deep Random Networks”. June 7, 2021. arXiv: 2106.03970 [cs, stat].
36. M. Danilevsky, K. Qian, R. Aharonov, Y. Katsis, B. Kawas, and P. Sen. “A Survey of the State of Explainable AI for Natural Language Processing”. 2020. arXiv: 2010.00711.
37. A. de Santana Correia and E. L. Colombini. “Attention, Please! A Survey of Neural Attention Models in Deep Learning”. In: *Artif. Intell. Rev*. (2022), pp. 1–88.
38. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. “Annotated BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proc. 2019 Conf. North Am. Chapter Assoc. Comput. Linguist. Hum. Lang. Technol. Vol. 1 Long Short Pap*. NAACL-HLT 2019. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. https://doi.org/10.18653/v1/N19-1423.
39. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. “Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding”. 2018. arXiv: 1810.04805.
40. X. Dong, Z. Yu, W. Cao, Y. Shi, and Q. Ma. “A Survey on Ensemble Learning”. In: *Front. Comput. Sci*. 14.2 (2020), pp. 241–258.
41. K. Doshi. *Transformers Explained Visually (Part 3): Multi-head Attention, Deep Dive*. Medium. June 3, 2021. URL: https://towardsdatascience.com/transformers-explained-visuallypart-3-multi-head-attention-deep-dive-1c1ff1024853 (visited on 11/19/2021).
42. A. Fan, M. Lewis, and Y. Dauphin. “Hierarchical Neural Story Generation”. 2018. arXiv: 1805.04833.
43. Y. Fan, F. Tian, Y. Xia, T. Qin, X.-Y. Li, and T.-Y. Liu. “Searching Better Architectures for Neural Machine Translation”. In: *IEEEACM Trans. Audio Speech Lang. Process*. 28 (2020), pp. 1574–1585.
44. Y. Gal and Z. Ghahramani. “Bayesian Convolutional Neural Networks with Bernoulli Approximate Variational Inference”. 2015. arXiv: 1506.02158.
45. Y. Gal, J. Hron, and A. Kendall. “Concrete Dropout”. 2017. arXiv: 1705.07832.
46. A. Galassi, M. Lippi, and P. Torroni. “Attention in Natural Language Processing”. In: *IEEE Transactions on Neural Networks and Learning Systems* 32 (Oct. 1, 2021), pp. 4291–4308. https://doi.org/10.1109/TNNLS.2020.3019893.

47. J. Gawlikowski et al. “A Survey of Uncertainty in Deep Neural Networks”. 2021. arXiv: 2107.03342.
48. T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent. “Fast Approximate Natural Gradient Descent in a Kronecker-Factored Eigenbasis”. 2018. arXiv: 1806.03884.
49. M. Geva, R. Schuster, J. Berant, and O. Levy. “Transformer Feed-Forward Layers Are Key-Value Memories”. In: (Dec. 29, 2020). URL: https://arxiv.org/abs/2012.14913v2 (visited on 11/08/2021).
50. B. Ghojogh and A. Ghodsi. “Attention Mechanism, Transformers, BERT, and GPT: Tutorial and Survey”. In: (2020). URL: https://osf.io/m6gcn/download.
51. I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Vol. 1. MIT press Cambridge, 2016. URL: https://www.deeplearningbook.org/.
52. A. Graves. “Sequence Transduction with Recurrent Neural Networks”. 2012. arXiv: 1211.3711.
53. F. K. Gustafsson, M. Danelljan, and T. B. Schon. “Evaluating Scalable Bayesian Deep Learning Methods for Robust Computer Vision”. In: *Proc. IEEECVF Conf. Comput. Vis. Pattern Recognit. Workshop*. 2020, pp. 318–319.
54. K. He, X. Zhang, S. Ren, and J. Sun. “Deep Residual Learning for Image Recognition”. In: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit*. 2016, pp. 770–778.
55. K. He, X. Zhang, S. Ren, and J. Sun. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification”. In: *Proc. IEEE Int. Conf. Comput. Vis*. 2015, pp. 1026–1034.
56. X. He, K. Zhao, and X. Chu. “AutoML: A Survey of the State-of-the-Art”. In: *Knowl.-Based Syst*. 212 (2021), p. 106622.
57. J. Hilton. WebGPT: *Improving the Factual Accuracy of Language Models through Web Browsing*. OpenAI. Dec. 16, 2021. URL: https://openai.com/blog/improving-factual-accuracy/ (visited on 01/12/2022).
58. A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. “The Curious Case of Neural Text Degeneration”. Feb. 14, 2020. arXiv: 1904.09751 [cs].
59. J. Howard and S. Ruder. “Universal Language Model Fine-tuning for Text Classification”. In: *Proc. 56th Annu. Meet. Assoc. Comput. Linguist. Vol. 1 Long Pap*. ACL 2018. Melbourne, Australia: Association for Computational Linguistics, July 2018, pp. 328–339. https://doi.org/10.18653/v1/P18-1031.
60. C. Hu et al. “RankNAS: Efficient Neural Architecture Search by Pairwise Ranking”. 2021. arXiv: 2109.07383.
61. D. Hu. “An Introductory Survey on Attention Mechanisms in NLP Problems”. In: *Proc. SAI Intell. Syst. Conf*. Springer, 2019, pp. 432–448.
62. S. Ioffe and C. Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Int. Conf. Mach. Learn*. PMLR, 2015, pp. 448–456.
63. S. Jain and B. C. Wallace. “Attention Is Not Explanation”. 2019. arXiv: 1902.10186.
64. Y. Jiang, C. Hu, T. Xiao, C. Zhang, and J. Zhu. “Improved Differentiable Architecture Search for Language Modeling and Named Entity Recognition”. In: *Proc. 2019 Conf. Empir. Methods Nat. Lang. Process. 9th Int. Jt. Conf. Nat. Lang. Process. EMNLP-IJCNLP*. 2019, pp. 3576–3581.
65. M. Kastrati and M. Biba. “A State-of-the-Art Survey of Advanced Optimization Methods in Machine Learning”. In: *RTA-CSIT* (May 1, 2021), pp. 1–10.
66. R. Kehlbeck, R. Sevastjanova, T. Spinner, T. Stähle, and M. El-Assady. *Demystifying the Embedding Space of Language Models*. July 31, 2021. URL: https://bert-vs-gpt2.dbvis.de/.
67. N. S. Keskar, B. McCann, L. R. Varshney, C. Xiong, and R. Socher. “CTRL: A Conditional Transformer Language Model for Controllable Generation”. Sept. 20, 2019. arXiv: 1909.05858.
68. U. Khandelwal, O. Levy, D. Jurafsky, L. Zettlemoyer, and M. Lewis. “Generalization through Memorization: Nearest Neighbor Language Models”. Feb. 14, 2020. arXiv: 1911.00172.
69. D. P. Kingma and J. Ba. “Adam: A Method for Stochastic Optimization”. 2014. arXiv: 1412.6980.

70. N. Kokhlikyan et al. “Captum: A Unified and Generic Model Interpretability Library for PyTorch”. Sept. 16, 2020. arXiv: 2009.07896.
71. M. Kosinski, D. Stillwell, and T. Graepel. “Private Traits and Attributes Are Predictable from Digital Records of Human Behavior”. In: *Proc. Natl. Acad. Sci.* 110.15 (2013), pp. 5802–5805.
72. A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet Classification with Deep Convolutional Neural Networks”. In: *Adv. Neural Inf. Process. Syst.* 2012, pp. 1097–1105.
73. B. Lakshminarayanan, A. Pritzel, and C. Blundell. “Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles”. In: *Adv. Neural Inf. Process. Syst.* 30 (2017).
74. S. Lapuschkin, A. Binder, G. Montavon, K.-R. Muller, and W. Samek. “Analyzing Classifiers: Fisher Vectors and Deep Neural Networks”. In: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.* 2016, pp. 2912–2920.
75. A. Lavie and A. Agarwal. “METEOR: An Automatic Metric for MT Evaluation with High Levels of Correlation with Human Judgments”. In: *Proc. Second Workshop Stat. Mach. Transl.* 2007, pp. 228–231.
76. J. Lee, M. Humt, J. Feng, and R. Triebel. “Estimating Model Uncertainty of Neural Networks in Sparse Information Form”. In: *Int. Conf. Mach. Learn. PMLR*, 2020, pp. 5702–5713.
77. S. Lee, S. Purushwalkam, M. Cogswell, D. Crandall, and D. Batra. “Why M Heads Are Better than One: Training a Diverse Ensemble of Deep Networks”. 2015. arXiv: 1511.06314.
78. M. Lewis. *Decoding Language Models · Deep Learning. Apr.* 20, 2020. URL: https://atcold.github.io/pytorch-Deep-Learning/en/week12/12-2/ (visited on 07/30/2021).
79. J. Li, X. Chen, E. Hovy, and D. Jurafsky. “Visualizing and Understanding Neural Models in Nlp”. 2015. arXiv: 1506.01066.
80. C.-Y. Lin. “Rouge: A Package for Automatic Evaluation of Summaries”. In: *Text Summ. Branches Out.* 2004, pp. 74–81.
81. T. Lin, Y. Wang, X. Liu, and X. Qiu. “A Survey of Transformers”. 2021. arXiv: 2106.04554.
82. H. Liu, Q. Yin, and W. Y. Wang. “Towards Explainable NLP: A Generative Explanation Framework for Text Classification”. June 11, 2019. arXiv: 1811.00196.
83. J. Z. Liu, Z. Lin, S. Padhy, D. Tran, T. Bedrax-Weiss, and B. Lakshminarayanan. “Simple and Principled Uncertainty Estimation with Deterministic Deep Learning via Distance Awareness”. Oct. 25, 2020. arXiv: 2006.10108.
84. S. M. Lundberg and S.-I. Lee. “A Unified Approach to Interpreting Model Predictions”. In: *Proc. 31st Int. Conf. Neural Inf. Process. Syst.* 2017, pp. 4768–4777.
85. A. Malinin and M. Gales. “Reverse Kl-Divergence Training of Prior Networks: Improved Uncertainty and Adversarial Robustness”. 2019. arXiv: 1905.13472.
86. P. H. Martins, Z. Marinho, and A. F. Martins. “Sparse Text Generation”. 2020. arXiv: 2004.02644.
87. B. McCann, J. Bradbury, C. Xiong, and R. Socher. “Learned in Translation: Contextualized Word Vectors”. In: *Adv. Neural Inf. Process. Syst.* 2017, pp. 6294–6305.
88. P. McClure and N. Kriegeskorte. “Robustly Representing Uncertainty through Sampling in Deep Neural Networks”. 2016. arXiv: 1611.01639.
89. L. McInnes, J. Healy, and J. Melville. “Umap: Uniform Manifold Approximation and Projection for Dimension Reduction”. 2018. arXiv: 1802.03426.
90. C. Meister, T. Vieira, and R. Cotterell. “If Beam Search Is the Answer, What Was the Question?” Jan. 17, 2021. arXiv: 2010.02650 [cs].
91. P. Mertikopoulos, N. Hallak, A. Kavis, and V. Cevher. “On the Almost Sure Convergence of Stochastic Gradient Descent in Non-Convex Problems”. June 19, 2020. arXiv: 2006.11144.
92. D. Metzler, Y. Tay, D. Bahri, and M. Najork. “Rethinking Search: Making Experts out of Dilettantes”. May 5, 2021. arXiv: 2105.02274 [cs].
93. T. Mikolov, K. Chen, G. Corrado, and J. Dean. “Efficient Estimation of Word Representations in Vector Space”. 2013. arXiv: 1301.3781.

94. G. A. Miller. “WordNet: A Lexical Database for English”. In: *Commun. ACM* 38.11 (1995), pp. 39–41.
95. C. Molnar. *Interpretable Machine Learning*. Jan. 21, 2022. URL: https://christophm.github.io/interpretable-ml-book/ (visited on 01/26/2022).
96. R. Moradi, R. Berangi, and B. Minaei. “A Survey of Regularization Strategies for Deep Models”. In: *Artif. Intell. Rev*. 53.6 (2020), pp. 3947–3986.
97. S. Morgan. *Tensorflow/Addons*. tensorflow, Dec. 1, 2020. URL: https://github.com/tensorflow/addons/blob/0c0fd8dfb4427df6b824c88f700ba5c7efd43bec/tensorflowaddons/optimizers/lamb.py (visited on 11/08/2021).
98. Z. Nado. *Baselines for Uncertainty and Robustness in Deep Learning*. Google AI Blog. Oct. 14, 2021. URL: http://ai.googleblog.com/2021/10/baselines-for-uncertainty-and.html (visited on 10/25/2021).
99. Z. Nado et al. “Uncertainty Baselines: Benchmarks for Uncertainty & Robustness in Deep Learning”. June 7, 2021. arXiv: 2106.04015.
100. R. Nakano et al. “WebGPT: Browser-assisted Question-Answering with Human Feedback”. 2021. arXiv: 2112.09332.
101. S. Narang et al. “Do Transformer Modifications Transfer Across Implementations and Applications?” Sept. 10, 2021. arXiv: 2102.11972 [cs].
102. R. M. Neal. *Bayesian Training of Backpropagation Networks by the Hybrid Monte Carlo Method*. Technical Report CRG-TR-92-1, Dept. of Computer Science, University of Toronto. Citeseer, 1992.
103. C. Nemeth and P. Fearnhead. “Stochastic Gradient Markov Chain Monte Carlo”. In: *J. Am. Stat. Assoc*. 116.533 (2021), pp. 433–450.
104. Z. Niu, G. Zhong, and H. Yu. “A Review on the Attention Mechanism of Deep Learning”. In: *Neurocomputing* 452 (2021), pp. 48–62.
105. K. Osawa, S. Swaroop, A. Jain, R. Eschenhagen, R. E. Turner, R. Yokota, and M. E. Khan. “Practical Deep Learning with Bayesian Principles”. 2019. arXiv: 1906.02506.
106. Y. Ovadia et al. “Can You Trust Your Model’s Uncertainty? Evaluating Predictive Uncertainty under Dataset Shift”. 2019. arXiv: 1906.02530.
107. G. Paass. “Assessing and Improving Neural Network Predictions by the Bootstrap Algorithm”. In: *Adv. Neural Inf. Process. Syst*. Citeseer, 1993, pp. 196–203.
108. G. Paass and J. Kindermann. “Bayesian Classification Trees with Overlapping Leaves Applied to Credit-Scoring”. In: *Res. Dev. Knowl. Discov. Data Min*. Ed. by X. Wu, R. Kotagiri, and K. B. Korb. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 234–245. ISBN: 978-3-540-69768-8. https://doi.org/10.1007/3-540-64383-4_20.
109. Paperswithcode. *Browse State-of-the-Art in AI*. 2019. URL: https://paperswithcode.com/sota.
110. K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. “Bleu: A Method for Automatic Evaluation of Machine Translation”. In: *Proc. 40th Annu. Meet. Assoc. Comput. Linguist*. 2002, pp. 311–318.
111. K. Pearson. “On Lines and Planes of Closest Fit to Systems of Points in Space”. In: *Lond. Edinb. Dublin Philos. Mag. J. Sci*. 2.11 (1901), pp. 559–572.
112. J. Pérez, J. Marinkoviæ, and P. Barceló. “On the Turing Completeness of Modern Neural Network Architectures”. 2019. arXiv: 1901.03429.
113. C. Pierse. *Transformers Interpret*. Version 0.5.2. Feb. 2021. URL: https://github.com/cdpierse/transformers-interpret (visited on 11/23/2021).
114. Pytorch. *PyTorch.* 2019. URL: https://pytorch.org/.
115. M. Qudar and V. Mago. *A Survey on Language Models*. Sept. 7, 2020. URL: https://www.researchgate.net/publication/344158120ASurveyonLanguage_Models/.
116. A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. “Improving Language Understanding by Generative Pre-Training”. In: (2018).
117. A. Radford, J. Wu, D. Amodei, D. Amodei, J. Clark, M. Brundage, and I. Sutskever. “Better Language Models and Their Implications”. In: *OpenAI Blog* (2019). URL: https://openai.%20com/blog/better-language-models.

118. A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. "Language Models Are Unsupervised Multitask Learners". In: *OpenAI blog* 1.8 (2019), p. 9.
119. S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models". May 13, 2020. arXiv: 1910.02054v3.
120. P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. "Squad: 100,000+ Questions for Machine Comprehension of Text". 2016. arXiv: 1606.05250.
121. A. Ramesh, M. Pavlov, G. Goh, and S. Gray. {DALL·E}: *Creating Images from Text*. Jan. 5, 2021. URL: https://openai.com/blog/dall-e/.
122. J. Rasley. *DeepSpeed*. Microsoft, Dec. 20, 2021. URL: https://github.com/microsoft/DeepSpeed (visited on 12/20/2021).
123. M. T. Ribeiro, S. Singh, and C. Guestrin. "Model-Agnostic Interpretability of Machine Learning". 2016. arXiv: 1606.05386.
124. A. Rogers, O. Kovaleva, and A. Rumshisky. "A Primer in {Bertology}: What We Know about How {BERT} Works". In: *Trans. Assoc. Comput. Linguist.* 8 (2021), pp. 842–866.
125. S. Rönnqvist, J. Kanerva, T. Salakoski, and F. Ginter. "Is Multilingual BERT Fluent in Language Generation?" 2019. arXiv: 1910.03806.
126. A. Rush. "The Annotated Transformer". In: *Proc. Workshop NLP Open Source Softw. NLP-OSS* Melbourne, Australia: Association for Computational Linguistics, July 2018, pp. 52–60. https://doi.org/10.18653/v1/W18-2509.
127. A. B. Sai, A. K. Mohankumar, and M. M. Khapra. "A Survey of Evaluation Metrics Used for NLG Systems". 2020. arXiv: 2008.12009.
128. E. F. Sang and F. De Meulder. "Introduction to the CoNLL-2003 Shared Task: Language-independent Named Entity Recognition". 2003. arXiv: cs/0306050.
129. S. Serrano and N. A. Smith. "Is Attention Interpretable?" 2019. arXiv: 1906.03731.
130. D. So, Q. Le, and C. Liang. "The Evolved Transformer". In: *Int. Conf. Mach. Learn. PMLR*, 2019, pp. 5877–5886.
131. L. Spinney. "Are We Witnessing the Dawn of Post-Theory Science?" In: *The Guardian. Technology* (Jan. 9, 2022). ISSN: 0261-3077. URL: https://www.theguardian.com/technology/2022/jan/09/are-we-witnessing-the-dawn-of-post-theory-science (visited on 01/11/2022).
132. M. Sundararajan, A. Taly, and Q. Yan. "Axiomatic Attribution for Deep Networks". In: *Int. Conf. Mach. Learn*. PMLR, 2017, pp. 3319–3328.
133. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. "Rethinking the Inception Architecture for Computer Vision". In: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.* 2016, pp. 2818–2826.
134. Y. Tay, D. Bahri, D. Metzler, D.-C. Juan, Z. Zhao, and C. Zheng. "Synthesizer: Rethinking Self-Attention in Transformer Models". May 24, 2021. arXiv: 2005.00743 [cs].
135. A. Taylor, M. Marcus, and B. Santorini. "The Penn Treebank: An Overview". In: *Treebanks* (2003), pp. 5–22.
136. Tensorflow. *Integrated Gradients | TensorFlow Core. TensorFlow*. Nov. 25, 2021. URL: https://www.tensorflow.org/tutorials/interpretability/integratedgradients (visited on 12/06/2021).
137. Tensorflow. *Tensorflow Webseite*. 2019. URL: https://www.tensorflow.org/.
138. tensorflow. *Uncertainty-Aware Deep Learning with SNGP | TensorFlow Core*. Tensor-Flow. 2021. URL: https://www.tensorflow.org/tutorials/understanding/sngp (visited on 07/25/2021).
139. E. Tjoa and C. Guan. "A Survey on Explainable Artificial Intelligence (Xai): Toward Medical Xai". In: *IEEE Trans. Neural Netw. Learn. Syst.* (2020).
140. L. van der Maaten and G. Hinton. "Visualizing Data Using T-SNE". In: *J. Mach. Learn. Res.* 9 (Nov 2008), pp. 2579–2605.
141. A. Vaswani et al. "Attention Is All You Need". In: *Adv. Neural Inf. Process. Syst.* 2017, pp. 5998–6008.
142. J. Vig. "A Multiscale Visualization of Attention in the Transformer Model". 2019. arXiv: *1906.05714*.
143. J. Vig. *BertViz*. Nov. 23, 2021. URL: https://github.com/jessevig/bertviz (visited on 11/23/2021).

144. J. Vig. *BERTVIZ: A Tool for Visualizing Multihead Self-Attention in the BERT Model*. 2019. URL: https://debug-ml-iclr2019.github.io/cameraready/DebugML-19paper2.pdf.
145. Wang. *SuperGLUE Benchmark*. SuperGLUE Benchmark. 2021. URL: https://super.gluebenchmark.com/ (visited on 02/23/2021).
146. A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. "Glue: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding". Feb. 22, 2019. arXiv: *1804.07461*.
147. D. Wang, C. Gong, M. Li, Q. Liu, and V. Chandra. "AlphaNet: Improved Training of Supernet with Alpha-Divergence". 2021. arXiv: *2102.07954*.
148. D. Wang, M. Li, C. Gong, and V. Chandra. "Attentivenas: Improving Neural Architecture Search via Attentive Sampling". In: *Proc. IEEECVF Conf. Comput. Vis. Pattern Recognit.* 2021, pp. 6418–6427.
149. H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han. "Hat: Hardware-aware Transformers for Efficient Natural Language Processing". 2020. arXiv: 2005.14187.
150. M. Welling and Y. W. Teh. "Bayesian Learning via Stochastic Gradient Langevin Dynamics". In: *Proc. 28th Int. Conf. Mach. Learn. ICML-11*. 2011, pp. 681–688.
151. L. Weng. *Attention? Attention!* Lil'Log. June 24, 2018. URL: https://lilianweng.github.io/2018/06/24/attention-attention.html (visited on 11/19/2021).
152. F. Wenzel et al. "How Good Is the Bayes Posterior in Deep Neural Networks Really?" 2020. arXiv: 2002.02405.
153. G. Wiedemann, S. Remus, A. Chawla, and C. Biemann. "Does BERT Make Any Sense? Interpretable Word Sense Disambiguation with Contextualized Embeddings". 2019. arXiv: 1909.10430.
154. Y. Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". 2016. arXiv: 1609.08144.
155. F. Xu, H. Uszkoreit, Y. Du, W. Fan, D. Zhao, and J. Zhu. "Explainable AI: A Brief Survey on History, Research Areas, Approaches and Challenges". In: *CCF Int. Conf. Nat. Lang. Process. Chin. Comput. Springer*, 2019, pp. 563–574.
156. Y. Xu et al. "GSPMD: General and Scalable Parallelization for ML Computation Graphs". Dec. 23, 2021. arXiv: 2105.04663 [cs].
157. Z. Yang, Z. Dai, R. Salakhutdinov, and W. W. Cohen. "Breaking the Softmax Bottleneck: A High-Rank RNN Language Model". 2017. arXiv: 1711.03953.
158. Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le. "Xlnet: Generalized Autoregressive Pretraining for Language Understanding". In: *Adv. Neural Inf. Process. Syst.* 2019, pp. 5753–5763.
159. Y. You et al. "Large Batch Optimization for Deep Learning: Training Bert in 76 Minutes". 2019. arXiv: 1904.00962.
160. C. Yun, S. Bhojanapalli, A. S. Rawat, S. J. Reddi, and S. Kumar. "Are Transformers Universal Approximators of Sequence-to-Sequence Functions?" 2019. arXiv: 1912.10077.
161. C. Yun, Y.-W. Chang, S. Bhojanapalli, A. S. Rawat, S. J. Reddi, and S. Kumar. "$O(n)$ Connections Are Expressive Enough: Universal Approximability of Sparse Transformers". 2020. arXiv: 2006.04862.
162. B. Zhang and R. Sennrich. "Root Mean Square Layer Normalization". 2019. arXiv: 1910.07467.
163. C. Zhang et al. "Resnet or Densenet? Introducing Dense Shortcuts to Resnet". In: *Proc. IEEECVF Winter Conf. Appl. Comput. Vis.* 2021, pp. 3550–3559.
164. T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi. "BERTScore: Evaluating Text Generation with BERT". Feb. 24, 2020. arXiv: 1904.09675.
165. W. Zhu, X. Wang, X. Qiu, Y. Ni, and G. Xie. "AutoRC: Improving BERT Based Relation Classification Models via Architecture Search". 2020. arXiv: 2009.10680.
166. M.-A. Zöller and M. F. Huber. "Benchmark and Survey of Automated Machine Learning Frameworks". In: *J. Artif. Intell. Res.* 70 (2021), pp. 409–472.