Leibniz
Universität
Hannover

Deconstructing Ranking Abilities of Language Models

Gottfried Wilhelm Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für verteilte Systeme
Fachgebiet Wissensbasierte Systeme
Forschungszentrum L3S

Thesis by
Fabian Beringer

First examiner:	xxxx
Second examiner:	xxxx
Advisors:	Abhijit Aanand
	Jonas Wallat

Hannover, xx.xx.2022

Abstract

...

Contents

1 Introduction

Due to large successes in computer vision, natural language processing (NLP) and a variety of other fields, machine learning (ML) and in particular deep learning (DL) have gained large popularity among researchers and practitioners alike. Both as object of research and in real world applications these methods have been widely adopted and shown to be effective for several use cases.

One major field, in which ML and DL has been playing an increasingly important role is information retrieval (IR). Not only has "Neural Information Retrieval" [MC18] been acknowledged as promising field of research within the IR research community, but it is already present in modern web IR systems used by google, amazon, netflix and other major companies, which do heavily rely on ML for delivering relevant content to their users.

While these systems most certainly do not solely operate on raw text, when it comes to to finding relevant documents for a given natural language query, the task of text ranking still lies at the heart of the problem.

In 2018 NLP experienced a major breakthrough when [Dev+19]. Core idea pretrain, finetune... While progress in pretraining lms and using for downstream task have been made before this that, this was first time the model achieved sota by large marging on variety of nlp benchmarks. and is often referred to as "the imagenet moment of NLP".

subfields including text ranking of nlp became aware and started to use.

text ranking bert consideredbbreakthrough

- modern ml system often black boxes

- need to understand: mitigate biases, make clearer to use, make better ... we deal with text ranking - advances in nlp, large models, hard to understand

1.1 Motivation

Nowadays, modern approaches for NLP often rely on large language models. These models are first pre-trained on huge amounts of text data and then fine-tuned on a smaller, task specific dataset. Although this approach has shown great effectiveness compared to traditional approaches, due to their size and complexity, these language models are mostly being treated as black boxes.

This is where several issues arise.

goal shine more light on how probing specifically targeted to IR use knowledge to improve

1.2 Problem Statement

I am still alive. - shine light on knowledge dist - research question

1.3 Contribution

1.4 Thesis Outline

2 Foundations

2.1 Information Retrieval - Ranking Text

Ranking is an integral part of the information retrieval (IR) process. The general IR problem can be formulated as follows: A user with a need for information expresses this information need through formulation of a query. Now given the query and a collection of documents, the IR system's task is to provide a list of documents that satisfy the user's information need. Further, the retrieved documents should be sorted by relevance w.r.t. the user's information need in descending order, i.e. the documents considered most relevant should be at the top of the list.

While from this formulation only, the task might appear simple, there are several caveats to look out for when it comes to ranking. For instance, there is no restriction on the structure of the query. While we might expect a natural language question like "What color are giraffes?" a user might decide to enter a keyword query like "giraffes color". The same applies to documents: Depending on the corpus we are dealing with, the documents might be raw text, structured text like HTML or even another type of media e.g. image, audio or a combination thereof.

Another possible issue is a mismatch in information need of the user and the corresponding query. Even if we find a perfect ordering of documents with respect to the query, we can not know for certain that the query actually reflects the user's information need. The user might not even know exactly what they're looking for until discovery through an iterative process, i.e. the information need is fuzzy and can not be specified through an exact query from the beginning on.

Further, a query might require additional context information in order for an IR system to find relevant documents. For example, depending on the time at which a query is prompted, the correct answer might change: "Who is president?" should return a different set of documents, as soon as a new president has been elected. Also, since not specified further, it is up to interpretation which country's president the user is interested in and might depend on their location. In addition, even the corpus might not be static either and change or grow over time, e.g. web search has to deal with an ever-growing corpus: the internet.

While this list of issues is not comprehensive, at this point the complexity of the ranking problem should have become apparent.

Because this work focuses on the ranking of text in the context of web search, we will now give a formal definition with that scenario in mind:

Given a set of $|Q|$ natural language queries $Q = \{q_i\}_{i=1}^{|Q|}$ and a collection of $|D|$ documents $D = \{d_i\}_{i=1}^{|D|}$, we want to find a scoring function $s : Q \times D \to \mathbb{R}$, such that for any query $q \in Q$ and documents $d, d' \in D$, it holds true that $s(q, d) > s(q, d')$ if d is more relevant w.r.t q than d'.

To give the reader a more concrete idea and as we are going to build upon it throughout this work, we will now discuss two traditional approaches to text retrieval which, unlike neural retrieval, are based on exact matching, meaning query and document terms are compared directly. Further, they're "bag of word" models, meaning queries and documents are treated as sets of terms without considering order.

2.1.1 TF-IDF

Term Frequency - Inverted Document Frequency weighting (TF-IDF), is a traditional ranking approach that, given a query, assigns a relevance score to each document based on two assumptions:

1. A document is relevant if terms from the query appear in it often.
2. A document is relevant if the terms shared with the query are also rare in the collection.

From these assumptions, two metrics are derived:

1. Term-Frequency
$$w_{t,d} = \begin{cases} 1 + \log \mathrm{tf}_{t,d} & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$
where $\mathrm{tf}_{t,d}$ is the count of term t in document d. The logarithmic scaling is motivated by the idea that a document does not linearly become more relevant by the number of terms in it: A document containing a term 10 times more often doesn't necessarily mean it is 10 times more important, e.g. the document might just be very long and contain more words in general. Note that this is just one possible normalization scheme out of many.

2. Inverted Document Frequency
$$\mathrm{idf}(t, d) = \log \frac{|D|}{\mathrm{df}_t} \tag{2.2}$$
where df_t counts the number of documents that a term occurs in over the full corpus. This way, terms that occur less frequent relative to the corpus size will receive a high IDF score and those that are more frequent a lower score.

To compute TF-IDF we can simply sum over the product of TF and IDF for each term in the query to produce a relevance score:

$$\text{score}(q, d) = \sum_{t \in q} w_{t,d} \times \text{idf}_t \tag{2.3}$$

Alternatively, vector space idf vector

2.1.2 BM25

2.2 Machine Learning

Machine learning can be described as a set of statistical methods, for automatically recognizing and extracting patterns from data. Typically, we can distinguish between two main types of machine learning: Supervised learning and unsupervised learning.

In the case of supervised learning, we have a set of training instances $X = \{x_i\}_{i=1}^{N}$ and corresponding labels $Y = \{y_i\}_{i=1}^{N}$, assigning a certain characteristic to each data point. For example, this characteristic might be a probability distribution over a set of classes or a regression score. If each y_i represents one or more categories from a fixed set of classes $C = \{c_i\}_{i=1}^{|C|}$, this is called a classification problem.

Now given the training data and labels, the goal is to find a hypothesis that explains the data, such that for unseen data points $x' \notin X$, the labels $y' \notin Y$ can be inferred automatically. One way to estimate the generalization ability of a model or algorithm, is to divide the dataset into a training and a test set, and only train on the training set while using the test set for evaluation. If the test set models the full distribution of data adequately, it can act as a proxy for estimating the error on unseen samples.

In contrast, in unsupervised learning there is no access to any labels whatsoever. Characteristics of the data need to be learned solely from the data X itself. Examples for this include clustering where X is clustered into groups, representation learning which usually tries to find vector representations for X, as well as dimensionality reduction that, if each X is already a vector, tries to compress them into more compact but still informative representations.

That being said, the separating lines between supervised and unsupervised learning are blurry. Especially with the emergence of semi-supervised approaches and "end2end" representation learning, modern ML methods often integrate parts of both.

2.3 Deep Learning

Deep learning is a subfield of ML that makes use of a class of models called Deep Neural Networks (DNN). Typically, DNNs find application in the supervised learning scenario and are often used for classification tasks. In the following we explain the basic mechanisms of DNNs and common approaches to train them.

2.3.1 Deep Neural Networks

In essence, a Deep Neural Network (DNN) is a function approximator $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ that applies a series of non-linear transformations to its inputs, in order to produce an output. In its simplest form, an input vector $x \in \mathbb{R}^n$ is multiplied by a single weight matrix, a bias vector is added, and the resulting vector is passed through a non-linear activation function σ.

$$f(x) = \sigma(Wx + b) \tag{2.4}$$

where $W \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ are learnable parameters. This model is commonly referred to as single layer feed-forward neural network (FFN) or single layer perceptron.

When used for classification, a single layer FFN is limited to problems that require linear separation. In order to learn more complex, non-linear decision boundaries, multiple layers can be applied in sequence.

An L-layer DNN can be described as follows:

$$\begin{aligned} h^{(1)} &= \sigma^{(1)}(W^{(1)}x + b^{(1)}) \\ h^{(2)} &= \sigma^{(2)}(W^{(2)}h^{(1)} + b^{(2)}) \\ &\vdots \\ f(x) &= \sigma^{(L)}(W^{(L)}h^{(L-1)} + b^{(L)}) \end{aligned} \tag{2.5}$$

popular activations

2.3.2 Optimization

Arguably, the most common way for optimizing a neural network are the gradient descent (GD) algorithm and its variants. For this, an objective function $J(\theta)$ is defined, based on the DNN's outputs and corresponding target labels over the training set.

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f(x_i;\theta), y_i) \tag{2.6}$$

Here, $\mathcal{L}$ is a differentiable loss function and θ represents the vector of all learnable parameters of the neural network $f(x)$.

Gradient Descent

For GD, the gradient of $J(\theta)$ with respect to θ is computed and scaled by a hyperparameter called learning rate η. If the objective is to minimize, the scaled gradient is subtracted from the original parameter vector.

$$\theta_{new} = \theta - \eta \nabla_{\theta} J(\theta) \tag{2.7}$$

By repeating this procedure iteratively, we can gradually minimize $J(\theta)$.

Common choices for $\mathcal{L}$ include:

- **Cross Entropy Loss**
  $$\mathrm{CE}(y, \hat{y}) = -\sum_{k=1}^{C} y_k \log \hat{y}_k \tag{2.8}$$
  for classification tasks. Where y_k is the ground truth probability of class k and $\hat{y}_k$ the corresponding prediction.
- **Mean Squared Error**
  $$\mathrm{MSE}(y, \hat{y}) = (y - \hat{y})^2 \tag{2.9}$$
  in the case of regression.

Stochastic Gradient Descent

The aforementioned algorithm is also known as the batch gradient descent (BGD) variant. Stochastic Gradient Descent (SGD) differs from BGD in the number of training samples that are used for a gradient update. Where BGD uses the gradient of the full training set for updating θ, SGD only considers a single, randomly picked sample for each update. Not only can this approach be more efficient, since less redundant computations are performed, due to its stochastic nature and high variance, it is more likely to break out of local minima, allowing additional exploration for better solutions. [Rud16]

Mini-Batch Gradient Descent

While SGD's high variance during training makes it more likely to escape local minima, it can also come with the disadvantage of unstable training. In this scenario, convergence might be hindered by overshooting desirable minima.

To mitigate this issue, we can simply use more than 1 sample, in order to achieve a more accurate estimate of the full gradient. Now, at each step a small subset of the dataset is sampled to reduce variance and stabilize training while retaining a level of stochasticity. This variant of gradient descent is called mini-batch gradient descent.

Building on mini-batch GD, many algorithms have been introduced in the context of DNNs, that employ further optimizations in order to improve convergence speed and quality. Notable examples include:

- Adagrad [DHS11]
- RMSProp [HSS12]
- Adam [KB14]

Algorithm 1: Mini-Batch Gradient Descent with batch size k, learning rate η

```
Data: $X = \{(x_0, y_0), ..., (x_n, y_n)\}$ training examples and target labels.
Input: function $f$ with trainable parameters $\theta$
initialize $\theta$ with random values ;
while not converged do
    $B \leftarrow$ next k training pairs $\in X$ ;
    $\theta \leftarrow \theta - \eta \nabla_\theta \big( \frac{1}{k} \sum_{(x_i, y_i) \in B} \mathcal{L}(f(x_i; \theta), y_i) \big)$ ;
end
```

Backpropagation

Because a neural network can consist of multiple layers and thus, is a composition of multiple non-linear functions, computing the gradient w.r.t. to each parameter of the network can become a non-trivial and even cumbersome task, if done by hand. One popular way of automatically computing the gradients of a DNN is the backpropagation algorithm [RHW+88].

Backpropagation is a direct application of the chain rule for calculating the derivative of the composition of two functions. Given two differentiable functions $f(x)$ and $g(x)$, the chain rule states that the derivate of their composition $f(g(x))$ is equal to the partial derivative of f w.r.t. g, times the partial derivate of g w.r.t x.

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x} \tag{2.10}$$

Let $a^{(k)} = W^{(k)} h^{(k-1)} + b^{(k)}$ be the intermediate output of an L-layer DNN at layer k, before passing it through an activation function σ (See 2.5). With a single application of the chain rule, we can compute the gradient of the objective function J w.r.t. $a^{(L)}$ like so:

$$\frac{\partial J}{\partial a^{(L)}} = \frac{\partial J}{\partial \sigma(a^{(L)})} \frac{\partial \sigma(a^{(L)})}{\partial a^{(L)}} \tag{2.11}$$

If we now apply the chain rule a second time, we can produce a term for computing the derivative w.r.t. $W^{(L)}$.

$$\frac{\partial J}{\partial W^{(L)}} = \frac{\partial J}{\partial \sigma(a^{(L)})} \frac{\partial \sigma(a^{(L)})}{\partial a^{(L)}} \frac{\partial a^{(L)}}{W^{(L)}} \tag{2.12}$$

Note that we now only need to know the derivatives of J, σ and $a^{(L)}$ separately, in order to compute the derivative of their composition. By recursively applying this rule, we can compute partial derivatives of J w.r.t to parameters of the DNN, up to an arbitrary depth, as long as all functions it is composed of are differentiable.

By modeling the chain of operations in a DNN as a computation graph, deep learning frameworks like PyTorch [Pas+19] or Tensorflow [Mar+15] can automatically perform backpropagation, as long as each operation's derivative is known and pre-defined in the library.

2.3.3 Regularization

Regularization includes a number of techniques to improve the generalization capabilities of an ML model. If an ML model achieves a low error rate on training data, but a high error rate on test data, it is said to be overfitting. In this scenario, the model has essentially "memorized" the training data and can no longer adapt to unseen examples. Regularization techniques tackle this problem by limiting the hypothesis space of models, through favoring simple solutions over complex ones.

Weight Decay

Weight decay constraints the number of possible hypothesis, by adding a penalty based on model parameters. For example, L2-regularization encourages small weights that lie on a hypersphere, by adding the sum of squares over all parameters to the loss function.

$$J(\theta) = \mathcal{L}(\theta) + \lambda ||\theta||_2^2 \tag{2.13}$$

As L2 is only a soft constraint, its effect can be regulated by hyperparameter λ.

Dropout

Dropout is a DNN specific method that, during training time, randomly sets entries in the input vector of a layer to 0 with probability p [Hin+12]. The initial idea of this approach is, to prevent groups of neurons from co-adapting, i.e. requiring the activation of one another in order to detect a certain feature. If dropout is employed, a neuron can no longer rely on the presence of another neuron. Dropout can also be seen as a way of training an ensemble of sub-networks of the original network which share the same parameters.

2.4 Transformer Models

One of the most prominent deep learning architectures of the past years is the transformer [Vas+17]. The transformer and its variants have set multiple state-of-the-art records in a variety of NLP tasks [Dev+19; Liu+19; Cla+20; Sho+19; Bro+20], and have since then also been adapted to other domains such as computer vision [Dos+20] or audio generation [Dha+20].

In this section we will discuss the architecture and ideas behind it and explain one of the most popular training approaches for NLP, named BERT [Dev+19].

2.4.1 Architecture

The transformer architecture is based on a single building block which, after an input layer, is repeatedly applied in order to form the full model. Throughout this thesis we will also refer to these building blocks as layers, e.g. a 12-layer model consists of an input layer followed by 12 blocks. A single transformer block consists of:

- a multi-head attention layer
- a point-wise feed-forward layer
- residual connections
- layer normalization

We will first explain the input layer, then go over each of these elements and explain how a transformer block is constructed from them.

2.4.2 Input Layer

The transformer's input layer takes in a sequence of tokens and produces continuous representations, by selecting corresponding vectors from a learned embedding matrix $M \in \mathbb{R}^{d \times |V|}$. Here $|V|$ denotes the size of the vocabulary and d the hidden dimension of the model. Because the transformer model itself does not encode the order of inputs, *positional encodings* are added to the initial embeddings, in order to inject positional information.

One way for generating positional encodings, is to introduce a new set of learned embeddings of dimension d, one for each input position. However, this approach requires a fixed maximum input length, as all embeddings have to be defined before training. Alternatively, [Vas+17] propose to use sine and cosine waves, as a function of the position:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{2.14}$$

$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{2.15}$$

Where pos and i denote position along in along sequence and hidden dimension respectively. They found this approach to perform nearly identical to learned embeddings in the case of machine translation.

2.4.3 Multi-Head Self-Attention

The Attention Mechanism

Originally, attention mechanisms have been proposed in the context of neural machine translation (NMT) [BCB14; LPM15]. Particularly, they were used for aligning words from a source language with their corresponding translations, i.e. pointing out the source words that are relevant for predicting the next translation target. The alignment is important, as different languages usually do not share the same word order, making a sequential word-to-word translation infeasible.

Generally speaking, attention is a mechanism that allows a model to focus on parts of its inputs, usually while considering a certain context. This could for example be words in a text (inputs) that are regarded important for answering a question (context) [XZS16] or patches of pixels in an image (inputs) that are important for detecting a certain object type (context) [Xu+15].

Given a sequence of N input vectors $X = (x_1, \dots, x_N) \in \mathbb{R}^{N\times d}$ and M context vectors $C = (c_1, \dots, c_M) \in \mathbb{R}^{M\times d}$, we can describe the general attention mechanism as follows:

$$s_{ij} = a(x_i, c_j) \tag{2.16}$$

$$\alpha_{ij} = \frac{\exp(s_{ij})}{\sum_{k=1}^{N}\exp(s_{kj})} \tag{2.17}$$

$$h_j = \sum_{k=1}^{N} \alpha_{kj} x_k \tag{2.18}$$

Where $a : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is a scoring function that assigns importance scores to input x_i given context c_j. The attention weights α_{ij} are then used to produce a context-sensitive representation h_j as weighted sum of X.

Common choices for a include:

- $a(u, v) = u \cdot v$ (dot product)
- $a(u, v) = w^\top \tanh(Wu + Uv)$ (additive)
- $a(u, v) = \sigma(w^\top \tanh(Wu + Uv + b) + c)$ (MLP)

Self-Attention

Self-Attention is a special case of attention where input vectors and context vectors stem from the same input sequence. It can be seen as the model attending to a sequence, given the sequence itself as context. In [Vas+17] a third set of *value* vectors is introduced, resulting in three sequences termed *query*, *key* and *value* vectors, in analogy to memory lookups.

To produce these vectors, the initial input sequence is transformed by three different learned weight matrices, namely $W^{(q)}, W^{(k)} \in \mathbb{R}^{d \times d_k}$ and $W^{(v)} \in \mathbb{R}^{d \times d_v}$.

$$Q = XW^{(q)} \tag{2.19}$$

$$K = XW^{(k)} \tag{2.20}$$

$$V = XW^{(v)} \tag{2.21}$$

Then, using the obtained query and key vectors Q and K, a matrix of attention scores is computed and matrix multiplied with V:

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2.22}$$

As a consequence, each vector in the resulting sequence becomes an attention weighted sum over the vectors in V. Note the scaling by $\sqrt{d_k}$ which is supposed to prevent oversaturation of the softmax function, due to large dot-products.

From the memory lookup perspective: Query vectors Q are matched with key vectors K, in order to produce compatibility scores. These scores are then used to retrieve value vectors from V via soft-lookup.

Multi-Head Attention

Self-Attention can further be extended to multi-head attention by running multiple self-attention layers in parallel, then concatenating and projecting their outputs:

$$\text{MultiHeadAttention}(Q, K, V) = \left[\,\Big\Vert_{i=1}^{H} \text{SelfAttention}_i(Q, K, V)\right] W^{(o)} \tag{2.23}$$

Where H is the number of attention-heads, $||$ denotes concatenation and $W^{(o)} \in \mathbb{R}^{d_v H \times d}$ is a learned matrix for projecting back to the model's original hidden dimension.

Note that in the default case, each attention layer has its own weight matrices. However, we omit layer indices to keep the notation more simple.

2.4.4 Point-wise Feed-forward

The point-wise feed-forward component is a 2-layer MLP that is applied to each position along the sequence dimension, meaning weight parameters are shared across all positions. It follows the following architecture:

$$\text{FFN}(x) = \text{ReLU}(xW^{(0)} + b^{(0)})W^{(1)} + b^{(1)} \tag{2.24}$$

2.4.5 Residual Connection

After applying a layer or block of layers in a DNN, if we add the inputs back to its outputs, it is called a residual connection or skip connection:

$$\text{Residual}(x) = f^{(k)}(x) + x \tag{2.25}$$

Where $f^{(k)}(x)$ is a layer at depth k.

The most prominent motivation for residual connections is that they facilitate the training of deeper neural networks. A common issue with deep neural networks is the vanishing gradient problem. As computing gradients through backpropagation relies on a series of multiplications of potentially small values (Section 2.3.2), gradients tend to become smaller the further we propagate back. This makes training very deep networks harder, as early layers might receive little to no updates.

Since $\text{residual}'(x) = f'(x) + 1$, the gradient of the residual connection will be $>$ 1, even if the gradient is $<$ 1, alleviating the effect of vanishing gradients. It can also be interpreted as preserving more of the initial input information throughout the network, treating the DNN layers as an addition to the identity function, instead of a full transformation of the input.

2.4.6 Layer Normalization

Layer Normalization [BKH16] is another technique for training deeper neural networks. When training machine learning models on numerical features, it is common practice to normalize inputs, e.g. such that their distribution is centered at 0 and has a standard deviation of 1. This way, there's less variance across features, resulting in more stable training and hence improving convergence.

However, since DNNs pass features through multiple layers, there's no guarantee that hidden representations will maintain a reasonable scale, meaning each layer might have to adapt to a new distribution [IS15]. Layer Normalization tackles this problem by computing mean $\mu^{(k)}$ and standard deviation $\sigma^{(k)}$ over the feature dimension of each hidden layer k:

$$\mu^{(k)} = \frac{1}{D} \sum_{i=1}^{D} z_i^{(k)} \tag{2.26}$$

$$\sigma^{(k)} = \sqrt{\frac{1}{D}\sum_{i=1}^{D}(z_i^{(k)} - \mu^{(k)})^2} \tag{2.27}$$

Here $z_i^{(k)}$ denotes the i-th output of layer k with hidden dimension D, before applying an activation function.
These layer statistics are then used to normalize the hidden layer representation $z^{(k)}$:

$$\text{LayerNorm}(z^{(k)}) = \gamma^{(k)} \circ \frac{z^{(k)} - \mu^{(k)}}{\sigma^{(k)}} + \beta^{(k)} \tag{2.28}$$

Where $\gamma^{(k)}$ and $\beta^{(k)}$ are learned parameter vectors for layer k and $\circ$ denotes the element-wise product. Further, $\gamma^{(k)}$ and $\beta^{(k)}$ are additional learnable parameters for adjusting scale and shift of the normalized distribution if required.

2.4.7 The Full Transformer Block

The full transformer block can be described with the following equations:

$$\begin{aligned} A &= \text{MultiHeadAttention}(X, X, X) \quad (2.29)\\ Z &= \text{LayerNorm}(A + X) \quad (2.30)\\ \text{TBlock}(X) &= \text{LayerNorm}(\text{FFN}(Z) + Z) \quad (2.31) \end{aligned}$$

It consists of a multi-head attention layer and a point-wise fully connected layer, each followed by residual connection and layer normalization.

2.4.8 BERT Pre-Training

2.5 Probing

2.6 MTL

3 Previous Work

4 Datasets

4.1 Probing

To assess the distribution of knowledge across layers of a BERT model, we design a set of tasks that require information on different abstraction levels in order to be solved.

For each task, we generate a dataset by sampling instances from the MSMARCO validation and test set and automatically infer targets. In the following we provide a list of all probing tasks and details on the corresponding dataset generation process.

4.1.1 BM25 Prediction

4.1.2 Term Frequency Prediction

4.1.3 Named Entity Recognition

4.1.4 Semantic Similarity

4.1.5 Coreference Resolution

4.1.6 Fact Checking

4.1.7 Relevance Estimation

4.2 Multitask Learning

5 Approach

5.1 Methodology

5.2 Experimental Setup

5.3 Evaluation Measures

5.3.1 MDL

5.3.2 Compression

5.3.3 F1

5.3.4 Accuracy

5.3.5 Ranking

MAP

MRR

NDCG

Precision

6 Results

7 Conclusion

7.1 Future Work

Plagiarism Statement

I hereby confirm that this thesis is my own work and that I have documented all sources used.

Hannover, xx.xx.2022

(Fabian Beringer)

List of Figures

List of Tables

Bibliography

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).

[Bro+20] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[Cla+20] Kevin Clark et al. “ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators”. In: *CoRR* abs/2003.10555 (2020). arXiv: 2003.10555. URL: https://arxiv.org/abs/2003.10555.

[Dev+19] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. URL: https://www.aclweb.org/anthology/N19-1423.

[Dha+20] Prafulla Dhariwal et al. *Jukebox: A Generative Model for Music.* 2020. DOI: 10.48550/ARXIV.2005.00341. URL: https://arxiv.org/abs/2005.00341.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[Dos+20] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *CoRR* abs/2010.11929 (2020). arXiv: 2010.11929. URL: https://arxiv.org/abs/2010.11929.

[Hin+12] Geoffrey E. Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors”. In: *CoRR* abs/1207.0580 (2012). arXiv: 1207.0580. URL: http://arxiv.org/abs/1207.0580.

[HSS12] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent". In: *University of Toronto, Technical Report* (2012).

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: `1502.03167`. URL: `http://arxiv.org/abs/1502.03167`.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[Liu+19] Yinhan Liu et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *CoRR* abs/1907.11692 (2019). arXiv: `1907.11692`. URL: `http://arxiv.org/abs/1907.11692`.

[LPM15] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. "Effective approaches to attention-based neural machine translation". In: *arXiv preprint arXiv:1508.04025* (2015).

[Mar+15] Martìn Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[MC18] Bhaskar Mitra and Nick Craswell. "An Introduction to Neural Information Retrieval". In: *Foundations and Trends® in Information Retrieval* 13.1 (2018), pp. 1–126. URL: `https://www.microsoft.com/en-us/research/publication/introduction-neural-information-retrieval/`.

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[RHW+88] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. "Learning representations by back-propagating errors". In: *Cognitive modeling* 5.3 (1988), p. 1.

[Rud16] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[Sho+19] Mohammad Shoeybi et al. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism". In: *CoRR* abs/1909.08053 (2019). arXiv: `1909.08053`. URL: `http://arxiv.org/abs/1909.08053`.

[Vas+17] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems.* 2017, pp. 5998–6008.

[Xu+15] Kelvin Xu et al. “Show, attend and tell: Neural image caption generation with visual attention”. In: *International conference on machine learning*. PMLR. 2015, pp. 2048–2057.

[XZS16] Caiming Xiong, Victor Zhong, and Richard Socher. “Dynamic coattention networks for question answering”. In: *arXiv preprint arXiv:1611.01604* (2016).