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

general retrieval ?

1.1 Motivation

Over the last couple of years, contextualized language representations from deep neural network (DNN) models have become the go-to approach for tackling natural language processing (NLP) tasks. In particular, the *transformer* [**vaswani2017attention**] and its variants, combined with large-scale, unsupervised pre-training, have shown unprecedented performance in a variety of NLP benchmarks.

However, unlike traditional approaches, these models consist of billions of automatically learned parameters, effectively turning them into huge black box functions. Because of this, understanding why and how such a model arrives at a certain decision, becomes a challenge in itself. Yet, as more and more NLP systems rely on these kinds of models, gaining a better understanding of their internal workings becomes crucial, especially when facing problems such as learned social biases [**Nadeem2021StereoSetMS**; **Bender2021OnTD**; **kurita2019measuring**], falsely motivated predictions [**10.1145/2939672.2939778**; **DBLP:journals/corr/abs-1802-00614**] or simply for determining causes of prediction error. In addition, a better understanding might provide insights on model weaknesses and guide model improvement, e.g. when adapting a model to new domains.

Recent work on understanding neural language models has aimed at measuring the extent to which certain information is encoded in their word representations. To achieve this, a *probing* classifier or *probe* is trained to predict certain linguistic properties from these representations. The probe's performance is then expected to reflect how well said property is captured. By employing multiple probing tasks, the presence of different types of information can be estimated. For example, by training a probe for part-of-speech tagging, it can be tested whether it is possible to extract part-of-speech tags from the representations.

while has recently emerged for nlp, but retrieval which important: google guides how we access information etc

we look at bert, initial break through approach, more recent iterations still similar in essence

1.2 Problem Statement

two part thesis...research question:

- given set of tasks, presumably ranking related, which knowledge does bert encode?
- how does ranking trained model differ from base language model
- can we exploit these differences to inform training for ranking?

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 encompasses 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 $f : \mathbb{R}^n \to \mathbb{R}^m$ 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:

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

Common choices for σ include:

- $\sigma(x) = \frac{1}{1+e^{-x}}$ (sigmoid)
- $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ (hyperbolic tangent)
- $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$ (softmax)
- $\text{ReLU}(x) = \max(0, x)$ (rectified linear unit [**nair2010rectified**])

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. [**ruder2016overview**]

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 [**duchi2011adaptive**]
- RMSProp [**hinton2012neural**]
- Adam [**kingma2014adam**]

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 [**rumelhart1988learning**].

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 [**NEURIPS2019˙9015**] or Tensorflow [**tensorflow2015-whitepaper**] 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 [**DBLP:journals/corr/abs-1207-0580**]. 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 [**vaswani2017attention**]. The transformer and its variants have set multiple state-of-the-art records in a variety of NLP tasks [**devlin-etal-2019-bert**; **DBLP:journals/corr/abs-1907-1169**
DBLP:journals/corr/abs-2003-10555; **DBLP:journals/corr/abs-1909-08053**; **DBLP:journals/c**
and have since then also been adapted to other domains such as computer vision [**DBLP:journals/corr/ab**
or audio generation [**https://doi.org/10.48550/arxiv.2005.00341**]. 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 [**devlin-etal-2019-bert**].

unlike previous methods like word embeddings, contextualized ...

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, [**vaswani2017attention**] 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) [**bahdanau2014neural**; **luong2015effective**]. 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) [**xiong2016dynamic**] or patches of pixels in an image (inputs) that are important for detecting a certain object type (context) [**xu2015show**].

Given a sequence of N input vectors $X = (x_1, \ldots, x_N) \in \mathbb{R}^{N\times d}$ and M context vectors $C = (c_1, \ldots, c_M) \in \mathbb{R}^{M\times d}$, 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 [**vaswani2017attention**] 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:

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

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 [**ba2016layer**] 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 [**DBLP:journals/corr/IoffeS15**]. 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:

$$A = \text{MultiHeadAttention}(X, X, X) \tag{2.29}$$

$$Z = \text{LayerNorm}(A + X) \tag{2.30}$$

$$\text{TBlock}(X) = \text{LayerNorm}(\text{FFN}(Z) + Z) \tag{2.31}$$

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

A key part in leading to the success of transformer models, is their effectiveness in large-scale transfer learning. In the context of NLP, the transfer learning procedure typically consists of two stages: First, a model is pre-trained on large amounts of human generated text data (e.g. from the internet), using language modeling as an objective. Then, the model is fine-tuned on a downstream task for which only limited amounts of data are available. As the model already learned basic language concepts during pre-training, it can now leverage this knowledge for the new task instead of learning it from scratch.

One particular approach to transfer learning for transformers, which has pushed state-of-the-art on several NLP benchmarks, is BERT (Bidirectional Encoder Representations from Transformers) [**devlin-etal-2019-bert**]. Unlike previous pre-training approaches that optimized for traditional *left-to-right* language modeling [**DBLP:journals/corr/abs-1801-06146**; **Peters:2018**; **radford2018improving**], BERT employs a bidirectional objective.

In left-to-right language modeling the goal is to predict the next word in a sequence of words, given the preceding words. BERT on the other hand employs a *masked language model* (MLM) objective. Here, tokens from the input sequence are randomly masked out or replaced with random tokens. The model then has to predict what the actual token should be.

This means that during pre-training, traditional language models only have access to the context left to their current prediction, while BERT has access to context in both directions.

In addition to MLM, BERT introduces a *next-sentence prediction* (NSP) task where, given two sentences, the model has to determine whether the second sentence follows the first, or was selected randomly. This is motivated by language tasks often requiring to model the relation between two sentences.

While many improved training procedures have been proposed since BERT, which often replace NSP or introduce additional objectives [**DBLP:journals/corr/abs-2003-10555**; **DBLP:journals/corr/abs-1907-11692**; **10.5555/3454287.3454804**; **Lan2020ALBERT**], MLM still remains an integral part of most of these approaches.

2.5 Probing

Probing is a method for estimating the presence of certain properties in dense vector representations of neural network models. For this, a classifier (*probe*) P is trained on the fixed representations of a *subject model* $\mathcal{M}$, to predict property Y. In NLP we typically probe for linguistic properties in word embeddings. For example, we could estimate the presence of part-of-speech (POS) information, by predicting POS tags. Based on the probe's performance, e.g. when compared to embeddings from a random baseline, we can conclude whether the property can be decoded from the subject model's representations or not.

3 Previous Work

two part thesis, these are relevant ...

tenney, hewitt all the good probing effects of finetuning https://arxiv.org/abs/2004.14448 the idf paper probing different berts bertnesia

3.1 Probing

3.2 Multitask Learning

4 Datasets

4.1 TREC 2019 - Deep Learning Track

The TREC 2019 deep learning track focuses on studying text retrieval on large-scale data [**DBLP:journals/corr/abs-2003-07820**]. It provides two datasets, one for passage retrieval and one for document retrieval. The datasets are based on MS MARCO [**DBLP:journals/corr/NguyenRSGTMD16**] which consists of $\sim$ 1mio real world user queries from the Bing search engine and a corpus of $\sim$ 8.8mio passages. For this thesis we focus on the passage retrieval dataset (TREC2019) if not stated otherwise. Further, we will refer to passages from this dataset as documents, to be consistent with the common information retrieval terminology.

Dataset	**Train**	**Validation**	**Test**
Passage	502,939	55,578	200
Document	367,013	5,193	200

Table 4.1: Number of queries for each dataset split in the two TREC 2019 datasets.

Passage	Document
8,841,823	3,213,835

Table 4.2: Corpus size for each TREC dataset.

While with TREC2019, two types of tasks are provided, namely full ranking and re-ranking, we will only perform the re-ranking task. This means, given a pool of 1000 documents for each query, we need to provide an ordering, such that relevant documents are placed at the top. On average, a query has $\sim$ 1.1 relevant documents in its pool which were marked as relevant by human annotators. Note that each annotator only had access to $\sim$ 10 passages during annotation, meaning a pool is likely to contain false negatives, i.e. relevant passages that are not marked as such.

4.2 Probing Dataset Generation

For all of our probing tasks, we automatically generate datasets from the TREC2019 passage-level test set. To achieve this, we sample 60k query-document pairs from the test set of which 40k are used as training set and 10k as validation and test set, respectively. We then use existing tools to extract the properties that we are interested in and use them to label the data. Details on how we generate labels for each task are described in section 5.1.

example for each task table like tenney?

5 Approach

edge probing graphic

Our goal is to understand whether certain properties that we expect to be relevant for ranking can be decoded from the hidden representations of a pre-trained transformer model. Further, we want to know to which degree these properties are encoded at different layers within the model.

To test this, we conduct the following experiment: First, we generate a set of datasets $D_{\text{probing}} = \{D_1, D_2, \dots, D_n\}$, each aimed at predicting a property Y_i that, based on traditional ranking methods, can be considered relevant for ranking. Then, for each dataset, we train a probing classifier $P_i : \mathbb{R}^d \rightarrow \mathbb{R}^c$ on top of the fixed hidden representations $H^{(k)} = \{h_i\}_{i=1}^N \in \mathbb{R}^{N \times d}$ of subject model $\mathcal{M}$. This procedure is repeated at every layer k. Finally, we compare the classifier's performance (compression, accuracy) across layers, to get a relative measure of how task-specific information is distributed throughout $\mathcal{M}$. To further put our measurements into perspective, we also employ a random baseline model $\mathcal{B}$ which is also probed for each task separately.

Following the probing experiments, we then attempt to develop an improved fine-tuning procedure, based on our findings.

5.1 Task Design

We propose a set of classification tasks with ranking properties as target variable. [**tenney-etal-2019-bert**] findings suggest that BERT stores language concepts in a hierarchical manner, with lower-level semantics being captured in early layers, while higher level concepts can be found closer to the output layer. To test whether this holds true for ranking, we choose tasks that require different levels of semantic abstraction in order to be solved. In this section we provide a list of the tasks we've chosen and explain the reasoning behind our selection, as well as how labels are generated automatically.

BM25 Prediction

BM25 (subsection 2.1.2) is a well-known text-retrieval method and still considered a first choice when it comes to computational efficiency. As heuristic designed around exact term matching, BM25 quantifies query-document pairs on a symbolic level, without the notion of higher level semantics.

Being able to decode BM25 from BERT embeddings might give a hint on whether exact matching properties like term-frequency or even corpus level statistics like inverted document-frequency are encoded by the neural ranking models.

To generate BM25 scores for each query-document pair, we use the Elasticsearch BM25 implementation[1].

Semantic Similarity

To measure semantic similarity, we compute cosine distance between query and document vectors in the GloVe[**pennington2014glove**] embedding space:

$$\mathrm{Sim}(q, d) = \frac{q^\top d}{||q|| \times ||d||} \tag{5.32}$$

While these dense word representations encode semantics, unlike BERT they are not contextualized, meaning they do not change based on surrounding words in a sentence. In that sense, using semantic similarity as ranking measure may be interpreted as a kind of soft term-matching: Words between query and document that are similar in meaning increase the estimated relevance.

In order to be able to compute semantic similarity between query and document we first embed all words in the GloVe [**pennington2014glove**] vector space, then compute the average embedding for query and document over the sequence dimension and use the resulting averages for cosine similarity.

Coreference Resolution

Coreference resolution is the task of deciding whether two mentions in a text refer to the same entity. To model coreference we use binary classification over two spans of text in query and document, respectively.

We argue that recognizing whether an entity is shared between query and document can be an important indicator on whether the document is relevant. For instance, knowing that "the us president" in a query refers to "joe biden" in a document is certainly helpful.

To detect and score entity pairs, we use the neuralcoref pipeline extension for spacy[2]. It consists of a rule-based mentions-detection module, followed by a feed-forward neural network which produces a binary coreference score for each detected pair.

Named Entity Recognition

Similar to coreference resolution, recognizing entities can be used to match concepts between query and document when performing retrieval. However, in the case of named entity recognition this matching is less restricted, as only entity *types* are considered,

[1]`https://www.elastic.co/de/elasticsearch/`
[2]`https://github.com/huggingface/neuralcoref`

instead of specific entities. For example, given the query: "How much PS does a jaguar have?", a document that contains car entities, should be prioritized over one with animal entities.

For our task, we solely test the general ability of the model to encode entities, meaning we treat document and query as a whole and predict entity types of separate text spans.

To identify named entities we use spacy's [**spacy2**] named entity recognition module. It is capable of detecting and assign one of 18 different types of entities. Naturally, we only include pairs that contain at least one entity.

Fact Checking

For fact checking, we leverage the existing FEVER [**thorne-etal-2018-fever**] dataset. Here, instead of query and document, a claim and evidence text are provided. The goal is to classify whether the evidence supports or refutes the claim. This task requires high level semantic reasoning, making it relevant for more fine-grained ranking, i.e. providing documents that contain *coherent* answers to a query and not only similar ones.

Since this is the only dataset that we don't sample from TREC2019, we can simply leverage the existing labels of FEVER and sample claim-evidence pairs from its train set.

Relevance Estimation

Because the TREC2019 dataset provides relevance labels, we can also directly probe what layers contain most information with respect to relevance prediction. This especially becomes interesting when comparing between models tuned for ranking and pure language models.

5.1.1 Probing Models

Subject Models

Subject of our probing experiments is the pre-trained BERT [**devlin-etal-2019-bert**] transformer model. In particular, we use the *bert-base-uncased*[3] variant, which consists of 12 layers, with $h = 12$ attention heads each and a hidden dimension of $d = 786$. The model has been trained on BooksCorpus [**7410368**] and text passages from English Wikipedia[4], which consist of 800 million and 2.5 billion words, respectively.

In addition, we probe two fine-tuned *bert-base-uncased* models that we term *bert-msm-passage* and *bert-msm-doc*. Both are trained on datasets from the TREC2019 deep learning track [**DBLP:journals/corr/abs-2003-07820**]. While *bert-msm-passage* is

[3]`https://huggingface.co/bert-base-uncased`
[4]`https://en.m.wikipedia.org/`

trained to predict relevancy of *passages* given a query, *bert-msm-doc* is trained on *document*-level data. Hence, for this purpose we use the TREC2019 passage- and document-level dataset respectively.

Note that all three models share the same architecture and only differ by which datasets they were trained on. Having access to additional versions of *bert-base-uncased* that were fine-tuned specifically for ranking, gives us a way to compare if and how the distribution of information throughout the model changes when being adapted to the new task.

Finally, as a baseline model we probe *random-embeddings*. For this we take a BERT embedding matrix and initialize its weights from a normal distribution $\sim \mathcal{N}(0, 1)$ and fix it during probing.

Probe

One problem in probing arises when trying to choose a probe of appropriate complexity [**hewitt-liang-2019-designing**]. If the classifier is too complex, it might end up modeling new, complex features itself and hence, rely less on the information that is already present in the subject model's representations. On the other hand, if the classifier is too simple, it might not be able to properly decode the information at all.

While there is recent debate on whether more complex models are actually problematic for probing [**pimentel-etal-2020-information**], following previous literature [**Tenney2019WhatDY**; **tenney-etal-2019-bert**; **hewitt-liang-2019-designing**] we decide on a simple 2-layer MLP which, despite its simplicity, is still capable of modeling non-linear relationships. Specifically, we use the same MLP as [**Tenney2019WhatDY**]:

$$\hat{P}(x) = \text{LayerNorm}(\tanh(xW^{(0)} + b^{(0)}))W^{(1)} + b^{(1)} \tag{5.33}$$

Where $W^{(0)} \in \mathbb{R}^{|S|d \times d}$, $W^{(1)} \in \mathbb{R}^{d \times c}$ and $b^{(0)} \in \mathbb{R}^{d}$, $b^{(1)} \in \mathbb{R}^{c}$ are learned parameters with hidden size d and number of target classes c.

Since some tasks require operating over one or multiple spans $S = \{(\text{start}_i, end_i)\}_{i=1}^{|S|}$, we further need to use a pooling mechanism, such that a fixed-length representation can be provided to the probe. Again, like [**Tenney2019WhatDY**] we employ the simple attention pooling operator also used in [**lee-etal-2017-end**; **lee-etal-2018-higher**]:

$$\begin{aligned} \alpha_n &= \frac{\exp(w^\top h_n)}{\sum_{k=i}^{j} \exp(w^\top h_k)} \\ \text{pool}(h_i, h_{i+1}, \ldots, h_j) &= \sum_{k=i}^{j} \alpha_k (W h_k + b) \end{aligned} \tag{5.34}$$

Where $w \in \mathbb{R}^d$, $W \in \mathbb{R}^{d \times d_{probe}}$, $b \in \mathbb{R}^{d_{probe}}$ are learned parameters which, in the case of multiple spans, are not shared between spans. The pooled span representations are concatenated and passed to the MLP, resulting in the full probe:

$$P(h_1, \ldots, h_N) = \hat{P}\left(\big\Vert_{(i,j)\in S} \mathrm{pool}(h_i, h_{i+1}, \ldots, h_j) \right) \tag{5.35}$$

Pooling Graphic

5.2 Probing Setup

5.2.1 Fine-tuning Subject Models

We fine-tune a *bert-base-uncased* model on both, TREC2019 passage- and document-level datasets (section 4.1), to obtain ranking subject models *bert-msm-passage* and *bert-msm-doc* (subsubsection 5.1.1), respectively. Each model is trained with a binary cross-entropy objective, for a maximum of 20 epochs. Early stopping is performed after 3 epochs, if no increase in validation MAP is observed. For hyperparameters, we keep the default settings suggested by [**devlin-etal-2019-bert**], using the Adam optimizer[**kingma2014adam**] with a learning rate of 1e-5, a mini-batch size of 16 and linear increase in learning rate over the first 1000 steps. As BERT uses a fixed set of learned positional embeddings, the input length is limited to 512 tokens. Hence, we truncate any passages and documents exceeding this maximum length, after applying BERT's word-piece tokenization.

5.2.2 Probe Training

All three subject models (subsubsection 5.1.1) are probed at their intermediate output representations, meaning a classifier is learned for each layer separately. This includes the initial sequence of non-contextualized embeddings from the input embedding-matrix, which we refer to as layer 0. We repeat this procedure for all of our proposed tasks (section 5.1).

As objective function, all probing tasks use cross-entropy loss (Equation 2.8). The Adam optimization algorithm [**kingma2014adam**] with a learning rate of 1e-4 and batch size 32 is employed for updating the probe parameters, while the subject model's parameters stay fixed. Learning rate is halved each epoch if the validation loss does not improve. The maximum number of epochs is set to 50, and we stop early if validation loss has not improved in 10 epochs. We set the probe classifier's hidden size to $d_{probe} = 256$ and apply dropout with rate 0.3 before the output layer.

5.3 Evaluation Measures

In the following, we explain the evaluation measures that we employ for probing and any subsequent experiment.

5.3.1 MDL

As mentioned in subsubsection 5.1.1, selecting a proper size for a probe can be difficult. With a large probing classifier, it becomes unclear whether the property probed for is decoded, or the classifier simply learns the task at hand. One way to address this problem, is to compare how well the classifier performs on randomly initialized baseline representations [**zhang-bowman-2018-language**]. Further, [**DBLP:journals/corr/abs-1909-03368**] propose the use of *control tasks*, for which task labels are randomly assigned, and a probe is selected based on the difference in accuracy to the original task. Both approaches however, often do not reflect a large difference in accuracy, when compared to the original representations or task.

As a solution, [**voita-titov-2020-information**] propose an information theoretic approach for measuring probe performance, instead of accuracy. By recasting learning a probe model to transmitting label data with the least amount of bits, a new measure can be applied: The *minimum description length* (MDL) required for transmitting the task labels, given the probed representations. Not only does MDL measure the probe's predictive performance, it also takes into account the amount of effort for achieving this performance, manifesting in model size or amount of training data required.

Since [**voita-titov-2020-information**] find that MDL is a more representative and also reliable measure than accuracy, we also choose it as preferred method for measuring probe performance. To compute MDL, we use the online code definition [**Rissanen1984UniversalCI**]. For this, the probing dataset $D = \{(x_i, y_i)\}_{i=1}^{n}$ is divided into timesteps $1 = t_0 < t_1 < \ldots < t_S = n$. After encoding block t_0 with a uniform code, for each following timestep a probing model P_{θ_i} is trained on the samples $(1, \ldots, t_i)$ and used to predict over data points $(t_i + 1, \ldots, t_{i+1})$. The full MDL is then computed as sum over the codelengths of each P_{θ_i} and the uniform encoding of the first block:

$$\begin{aligned}\mathrm{MDL}(y_{1:n}|x_{1:n}) &= t_1 \log_2 c \\ &- \sum_{i=1}^{S-1} \log_2 P_{\theta_i}(y_{t_i+1:t_{i+1}}|x_{t_i+1:t_{i+1}})\end{aligned} \tag{5.36}$$

where c is the number of target classes. Following [**voita-titov-2020-information**], we choose timesteps at 0.1, 0.2, 0.4, 0.8, 1.6, 3.2, 6.25, 12.5, 25, 50 and 100 percent of the dataset.

5.3.2 Compression

Because MDL depends on the number of targets in a probing dataset, it is not reasonable to directly compare it between different tasks. A common way to turn MDL into a relative measure, is by computing *compression*. Compression divides the codelength that would result from a uniform encoding by MDL:

$$\text{compression} = \frac{n \log_2(c)}{\text{MDL}(y_{1:n}|x_{1:n})} \tag{5.37}$$

This means compression is a measure of how much easier it is for the probe to decode a property from the probed representations, or in other words how well they encode the task labels.

5.3.3 Accuracy

As a secondary measure for probing, we further employ accuracy:

$$\text{Accuracy}(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}[\hat{y}_i = y_i] \tag{5.38}$$

This is simply the percentage of predictions $\hat{y}$ that match the correct target y.

5.3.4 Ranking

Typically, ranking metrics require a set of queries $Q = \{q_i\}_{i=1}^{|Q|}$, with each query being associated with a list of labeled candidate documents $C = \{c_i\}_{i=1}^{|C|}$. The list of candidate documents is expected to be ordered in terms of their predicted relevance and each document has a ground truth label with respect to the query.

MRR

Mean reciprocal rank measures the inverse position at which the first relevant document occurs in the result set, averaged over all queries:

$$\text{MRR}(Q, C) = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\text{rank}(q, C)} \tag{5.39}$$

Here, $\text{rank}(q, C)$ refers to the rank, i.e. the position of the first document in C that is labeled as relevant.

Precision@k

Precision@k measures precision at a specific threshhold in C, i.e. only the top-k results are considered when computing precision:

$$\text{Precision@}k = \left(\frac{\#\text{true positives}}{\#\text{true positives} + \#\text{false positives}} \right)_k \tag{5.40}$$

MAP

Unlike MRR, mean average precision (MAP) considers the rank of all relevant documents in the candidate set, not just the first one. Initially, average precision (AP) is computed over the set of candidates for a query:

$$\mathrm{AP}(q, C) = \frac{1}{|C_{rel}|} \sum_{k=1}^{|C|} \text{Precision@k} \times y_k \tag{5.41}$$

Where $|C_{rel}|$ is the total number of documents marked as relevant w.r.t. the query and y_k is the ground truth relevance at position k in C. Then taking the average AP over all queries gives us MAP:

$$\mathrm{MAP}(Q, C) = \frac{1}{|Q|} \sum_{q \in Q} \mathrm{AP}(q, C) \tag{5.42}$$

While MRR indicates the quality of the top-ranked document, MAP provides a better picture of the full candidate set ranking.

NDCG

Normalized Discounted Cumulative Gain (NDCG) quantifies a ranked list relative to its ideal ranking. NDCG can also be applied to non-binary labels. At first, a gain is computed, that rewards early placement of relevant documents in C, by logarithmically scaling relevance label y_i as a function of the position and summing over all labels:

$$\mathrm{DCG} = \sum_{i=1}^{|D|} \frac{y_i}{\log_2(i+1)} \tag{5.43}$$

Then, the ideal DCG is computed by sorting C according to the ground truth labels and using it to scale DCG.

$$\mathrm{NDCG} = \frac{\mathrm{DCG}}{\mathrm{IDCG}} \tag{5.44}$$

This way, a perfect ordering (of which multiple may exist) will result in an NDCG of 1 while any suboptimal ordering will lie in $[0, 1)$. Because of the logarithmic scaling, more importance is attributed to relevant documents early in the list.

6 Probing Results

In this section we will analyze and discuss the results from the probing experiments. Since the overall distribution of properties across layers follows a similar pattern across models, we will first focus on the distribution on *bert-base-uncased*, then discuss the effects of fine-tuning in the following section.

6.1 Distribution of Ranking Properties

Firstly, a general trend we can observe across tasks, is an increase in both compression and accuracy over the first couple of layers up to layer 4-6, suggesting that ranking concepts arise mostly in the mid-range of layers. Then, after a peak at some mid to upper level layer, we observe a constant decrease until the last layer. It is notable that accuracy appears to exhibit a less stable curve, with sudden peaks and drops in between adjacent layers, especially in the case of coreference resolution (COREF) and semantic similarity (SEM) Figure 6.1. This observation coincides with [**voita-titov-2020-information**] findings, that MDL and as a consequence compression, are a more reliable measure for probing than accuracy.

For the most part, based on compression, ranking properties can be decoded more efficiently from our trained models than from the random baseline, meaning the probe model is not solely adapting to the task, but instead leveraging information present in the pre-trained representations.

We can further observe that the difference in compression between early layers and the peak value varies depending on the task. This Indicates that some properties are more uniformly distributed across layers, while others are more concentrated at a particular layer. For example, decoding named entities results in similar compression scores from layer 1-11, while SEM shows a distinct peak at layer 4.

To better understand this behavior, we can have a look at the row-normalized heatmap in Figure 6.3 of the *bert-base-uncased* model. We can see that COREF and fact checking strongly center around layers 4-7 and 6-9 respectively, with fact checking showing a slightly wider spread. While BM25 exhibits a similar pattern, it is less focused around a particular layer, but instead more evenly distributed from layer 4-6. SEM and NER on the other hand look almost identical, with a rather flat distribution that is more prominent in early layers 1-4.

When considering that COREF and fact checking are both tasks that require higher level semantics, based on the observation in [**tenney-etal-2019-bert**], it makes sense that both distributions are leaning more towards mid to upper layers. On the other hand, based on this, we would expect a lower level concept such as SEM to be mostly present in early layers. Especially since it is a property that can already be captured by non-contextual word embeddings. However, except a slight peak at layer 1, we observe it to be more evenly spread across layers. We hypothesize that, as a fundamental concept which higher level tasks build on, it's a property of the embedding space that needs to be preserved throughout the whole model.

Further, even though NER appears to be a higher level concept than SEM at first, considering that similar categories of entities are likely grouped together in the embedding space [**DBLP:journals/corr/abs-1301-3781**; **pennington2014glove**], a relation to SEM property becomes apparent. Given that NER requires us to predict *categories* of entities, it might explain that NER shares a very similar distribution with SEM.

BM25 depends on both local and corpus-level frequency statistics of words. Finding that layers 4-6 are best for inferring BM25 might indicate that some kind of direct term comparison between query and document is modeled within this range of the model. [**https://doi.org/10.48550/arxiv.2202.12191**] who probe BERT for IDF, find that the IDF property is present in early layers and constantly decreases up to the final layer. However, unlike IDF, BM25 also encodes term frequency and considers the length of a document, supporting the idea that local interactions are more prominent in the mid-layers, while corpus level statistics are encoded more evenly across the model.

Finally, as compression is a relative measure that compares MDL to the MDL of a uniform encoding, it is possible to compare it *between* tasks. For this, we show absolute compression values in Figure 6.5. Firstly, the highest compression can be measured for NER with values > 6. Further, a compression around 4.5 is achieved for BM25, SEM and COREF, with COREF showing some peaks closer to 5. In contrast, for fact checking, we barely reach a compression of > 3.5. These findings indicate that NER is the most easily extractable property from the pre-trained base model, while fact checking appears to be the hardest.

compute centers of gravity? maybe table

6.2 Effects of Fine-tuning for Ranking

First of all, we can observe that for BM25, SEM and COREF resolution, the *bert-msm-passage* representations result in higher compression scores for most layers, suggesting that the extractability of these properties indeed increases through fine-tuning for ranking. Whereas this holds true for *bert-msm-doc* in COREF, for the other two tasks we can observe values closer to *bert-base-uncased.* Moreover, the difference starts to first become apparent around layer 2-3 and begins to increase from there on. This coincides with [**merchant-etal-2020-happens**], who find that a change in BERT's representations primarily occurs in the upper layers when fine-tuning.

Surprisingly, for both NER and fact checking, *bert-base-uncased* preserves most of the property while the fine-tuned versions appear to lose some of it. However, this difference also grows with increasing depth.

Another pattern resulting from fine-tuning that we can observe, is a sudden drop in compression at the last layer across all tasks. This might be a sign that through fine-tuning, the final layer becomes more specialized and as a result loses some of the general linguistic knowledge that has been acquired during pre-training. This is also supported by [**merchant-etal-2020-happens**].

We can gain further insights on how fine-tuning affects BERT's representations by comparing Figure 6.3 and Figure 6.4. For SEM, we can see that after fine-tuning, the property's distribution seems to center more around layer 4 and fade out gradually to both sides. In contrast, the pre-trained representations exhibit a more even distribution of the SEM property, with no apparent center. Similarly, the concentration of the NER property in early layers becomes more pronounced.

In contrast, for BM25 a more spread distribution that is shifted towards upper layers can be observed. This behavior is similar to the observations of [**https://doi.org/10.48550/arxiv.2202.1**
who found the IDF property to be more prominent in higher layers of BERT after fine-tuning for ranking. Likewise, COREF related information appears to shift towards higher layers, however with a distribution that concentrates more around a single layer. On the contrary, the fact checking property appears to spread further across layers with fine-tuning.

Lastly, we can use a heatmap of the absolute compression values to visualize how well knowledge w.r.t. different tasks is extractable from *bert-msm-passage*. While it was already recognizable from the line-plots that for the SEM and BM25 properties, *bert-msm-passage* representations achieve higher compression, from Figure 6.6 it becomes much more apparent that the COREF property becomes easier to decode from the embeddings than prior fine-tuning with almost double the compression across the board. This might be evidence that matching entities is an important part of a neural ranking model.

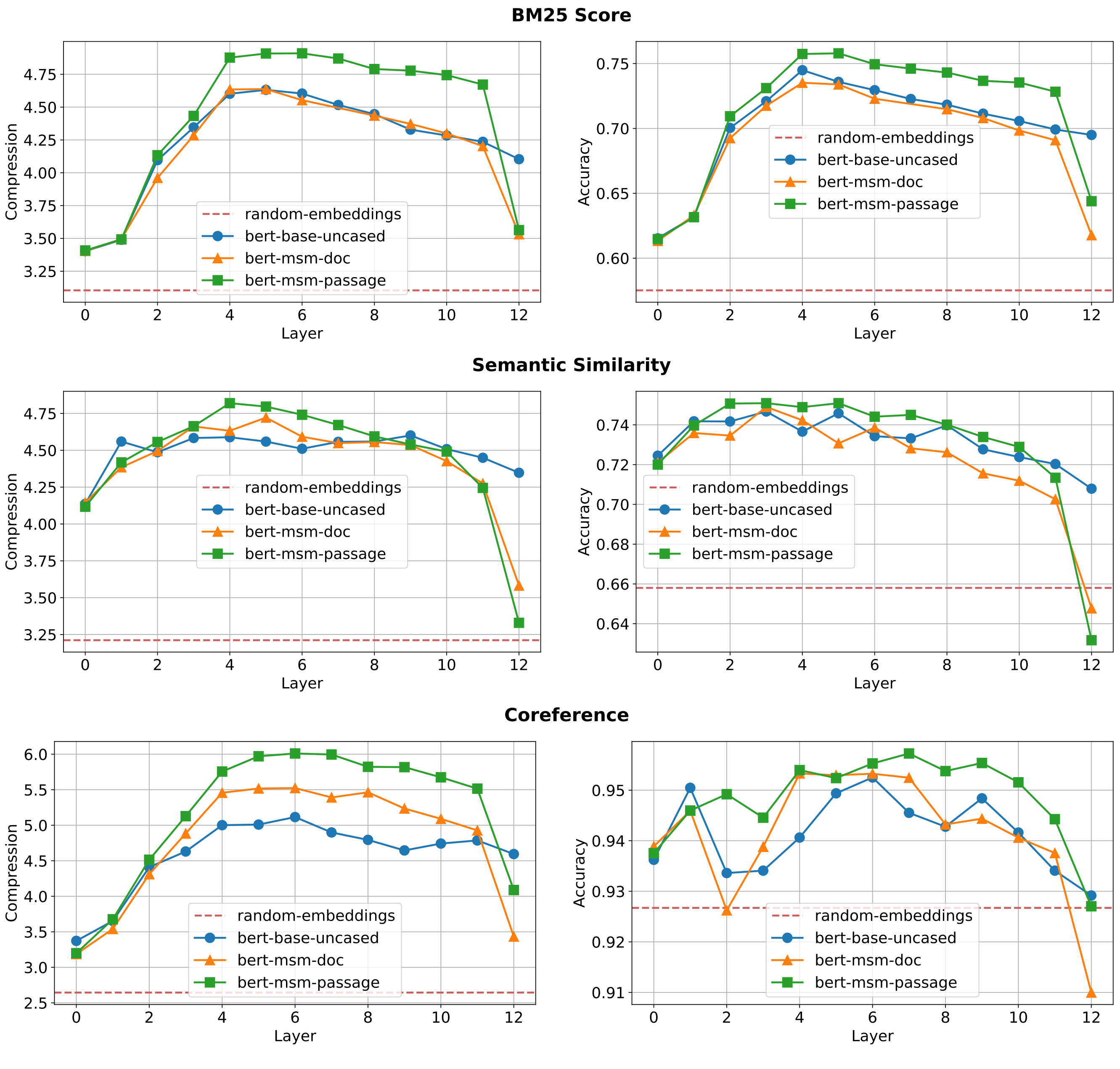

Figure 6.1: Layer to probing score for BM25, semantic similarity and coreference properties. We report accuracy on the test set.

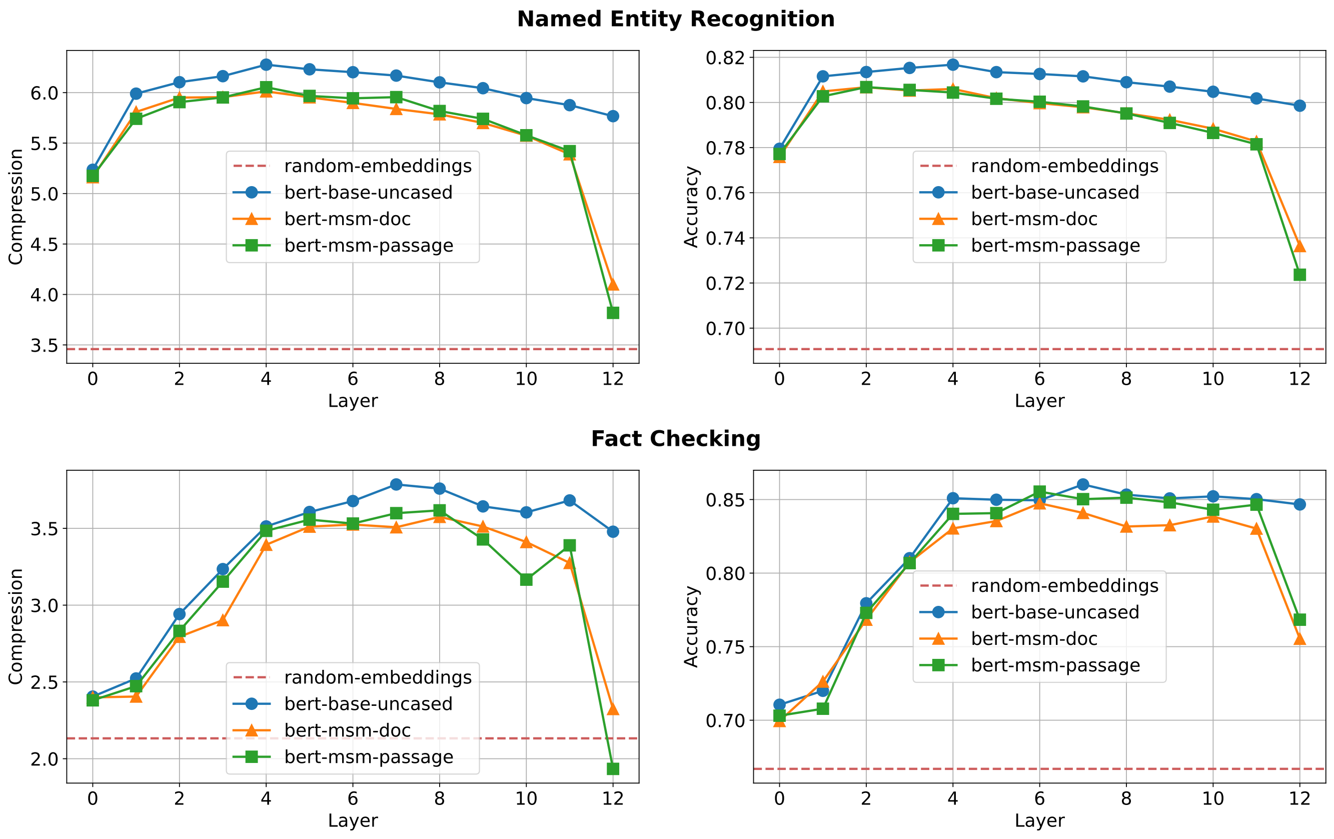

Figure 6.2: Layer to probing score for NER and fact checking properties. We report accuracy on the test set.

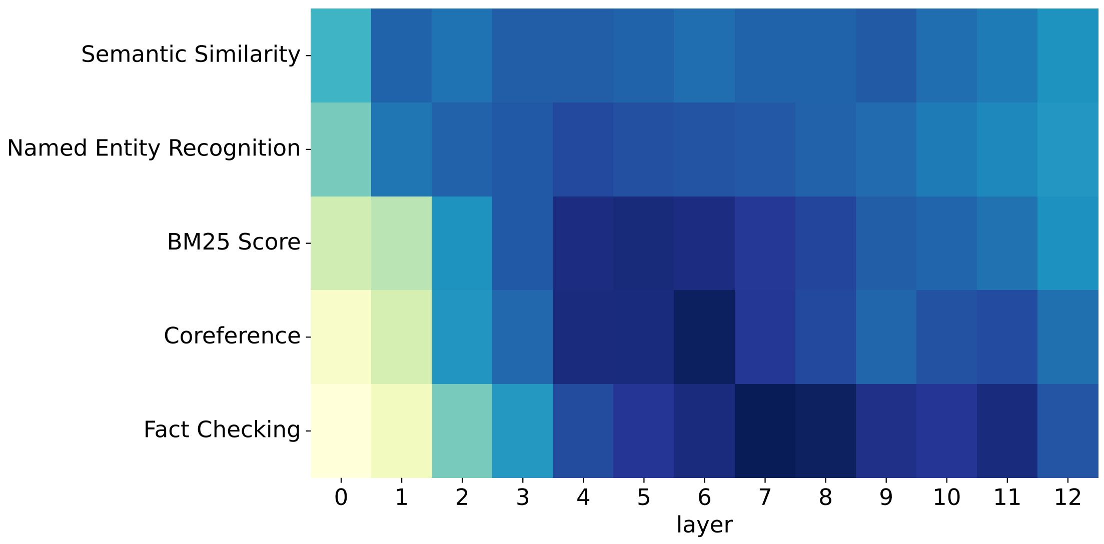

Figure 6.3: *bert-base-uncased*: compression as a function of task and layer, row-normalized. Darker colors represent higher values.

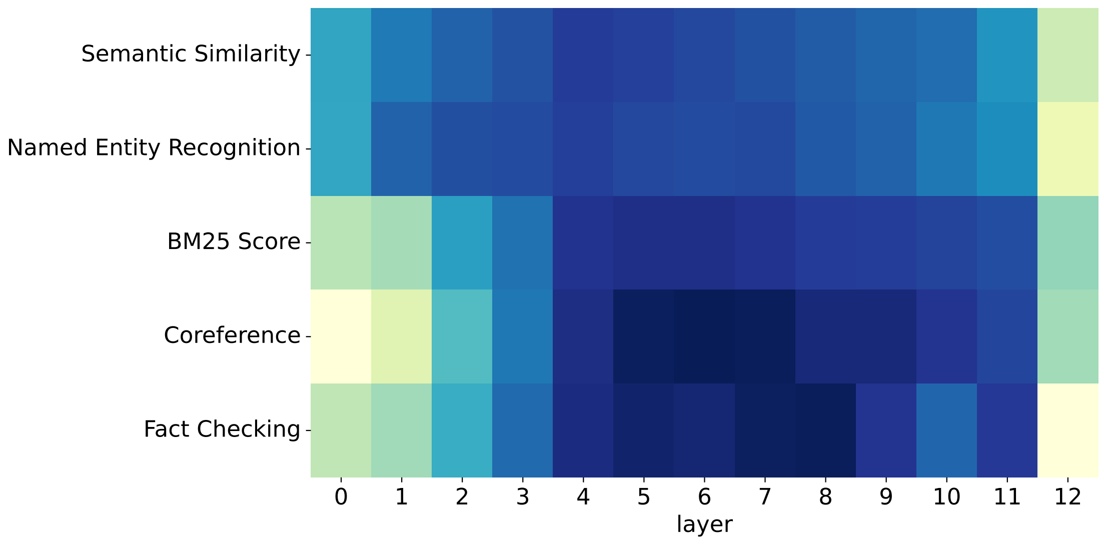

Figure 6.4: *bert-msm-passage* - compression as a function of task and layer, row-normalized. Darker colors represent higher values.

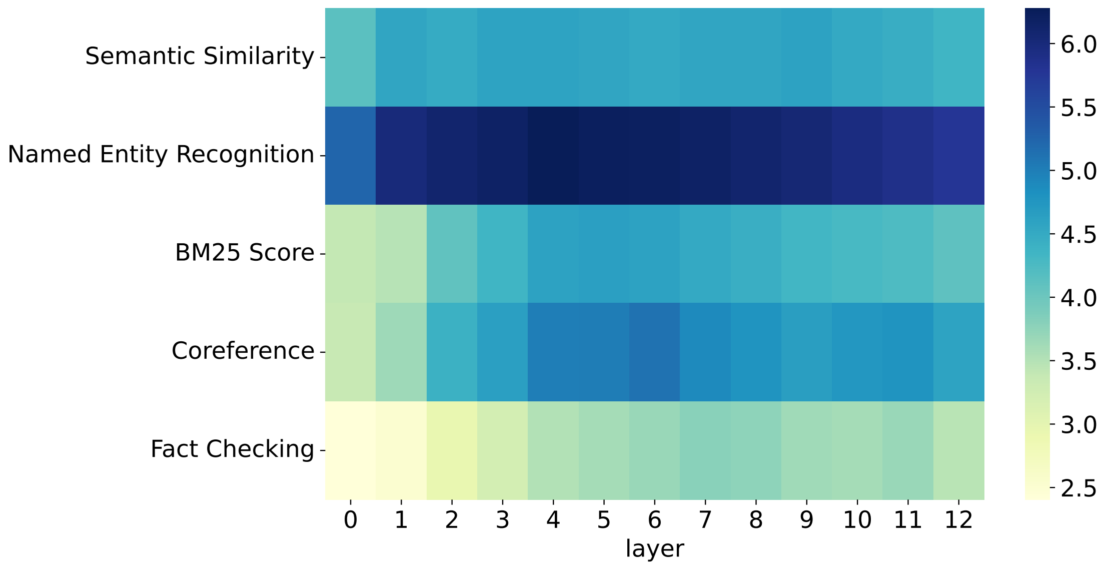

Figure 6.5: *bert-base-uncased* - compression as a function of task and layer, absolute values.

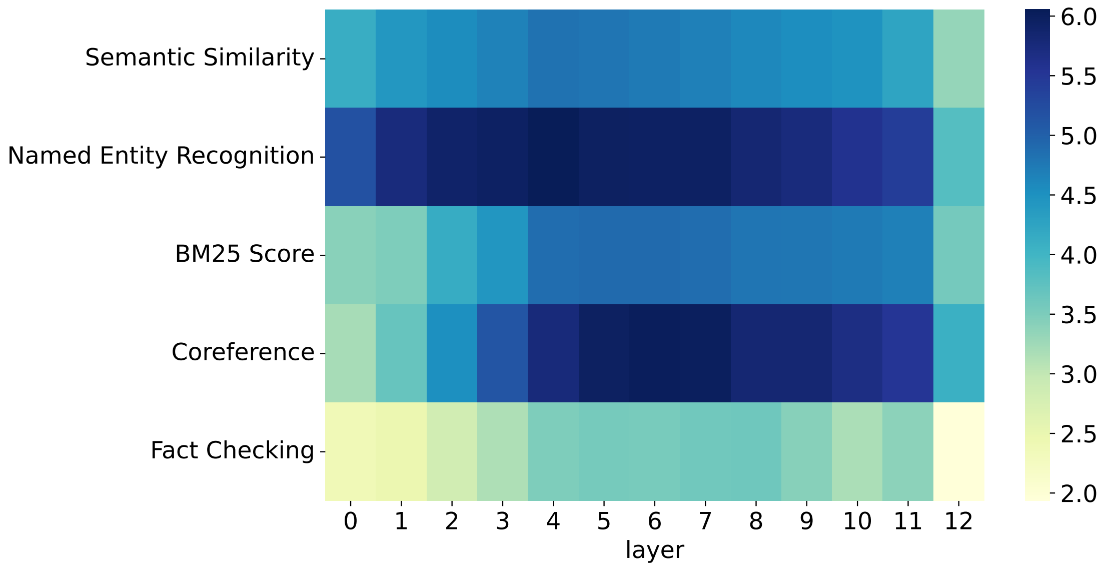

Figure 6.6: *bert-msm-passage* - compression as a function of task and layer, absolute values.

7 Informed Multi-Task Learning

Based on our findings from the probing experiments, we now want to address the question on whether the knowledge obtained from probing can be leveraged to improve the BERT model for ranking.

Hence, our knew objective becomes re-ranking: Given a query q and a set of candidate documents $C = \{c_i\}_{i=1}^{|C|} \subseteq D$, we want to learn a ranker $s : Q \times D \to \mathbb{R}$, such that for any two candidate documents $c, c' \in C$, it holds true that $s(q, c) > s(q, c')$ if c is more relevant than than c', regarding q.

During probing we made these key observations:

- Most of the probed ranking properties can be decoded from mid layers.
- Property are captured around specific layers.
- Fine-tuning for ranking can amplify the presence of a property in specific layers.

Therefore, we want our experiment to satisfy the following criterions:

- Exploit the fact that ranking properties emerge in intermediate layers.
- Use the knowledge on which layers capture what property best.

The main idea of our approach is to aid fine-tuning, by infusing task knowledge into specific layers, through *multitask learning*. We want to simultaneously learn ranking properties at the layers that we found to be best at capturing them, especially when being fine-tuned for ranking.

To achieve this, for each task we first select a layer that we found to efficiently encode the corresponding ranking property during probing. We then fine-tune the *bert-base-uncased* model to perform ranking on the TREC2019 dataset. At the same time, we learn classifiers on top of BERT's intermediate layer representations of the pre-selected layers, to predict the respective ranking properties.

7.1 Model Architecture

Given the intermediate layer representations of the pre-trained *bert-base-uncased* model $\{H^{(i)}\}_{i=0}^{12}$, with $H^{(i)} = (h_1, h_2, \ldots, h_N)$ being the sequence of token embeddings at layer i, for each task (section 5.1), we select a layer based on the probing results. We then apply average pooling across the sequence dimension to retrieve a fixed size embedding:

$$\text{pool}(h_i, h_{i+1}, \dots, h_j) = \frac{1}{j} \sum_{k=i}^{j} h_k \tag{7.45}$$

In the case of tasks that require multiple spans, we first average along each span $(i, j) \in S$ and then concatenate the resulting vectors:

$$\text{multi-pool}(h_1, \dots, h_N) = \bigg|\bigg|_{(i,j)\in S} \text{pool}(h_i, h_{i+1}, \dots, h_j) \tag{7.46}$$

Following the pooling we apply a simple MLP classifier of the form:

$$\text{FFN}(x) = \text{ReLU}(xW^{(0)} + b^{(0)})W^{(1)} + b^{(1)} \tag{7.47}$$

7.2 Experimental Setup

7.2.1 Datasets and Sampling

Because our objective is now re-ranking on TREC2019, we can no longer rely on the original probing datasets, as they were sampled from the test set and hence, this would result in test set overlap. Instead, we sample new query-document pairs from the TREC2019 train set. Our sampling goes as follows: We uniformly sample 100k train queries for each task. Then, for each query we retrieve 10 documents from the corpus using BM25, resulting in a dataset size of 1mio samples which is approximately the size of the TREC2019 ranking dataset. Each task dataset is then constructed by applying the same procedure as in section 4.2 and section 5.1, to automatically generate labels.

We exclude the fact-checking task, as we do not have a straight forward way to automatically generate samples and the dataset itself is too small in comparison to the other datasets.

7.2.2 Training

During training, we assemble mini-batches of size 32, by sampling from the TREC2019 train set and all generated task datasets with a probability proportional to their size. As loss function we use the objective proposed in [**aghajanyan-etal-2021-muppet**]:

$$\mathcal{L}(y, \hat{y}) = \sum_{t\in\text{tasks}} \frac{\text{CE}(y_t, \hat{y}_t)}{\log c_t} \tag{7.48}$$

Where c_t is the number of target classes and y_t, $\hat{y}_t$ are predictions and ground-truth labels with respect to task t, respectively. It scales each task loss, such that all losses would have equivalent values, if the class distribution were uniformly distributed, along with the predictions. Analogously to the probing experiments, regression tasks are cast to classification tasks by binning the targets into $k = 10$ categories. We train each model for up to a maximum of 100 epochs and perform early stopping after 3 epochs of no improvement in MAP on the TREC2019 validation set. As optimizer, we use Adam[**kingma2014adam**] with a learning rate of 2e-6 and linearly increase the learning rate over the first $10k$ steps. Each task specific classifier has a hidden size of 128 and dropout with rate 0.2 is applied before each layer.

7.3 Results

Tasks	Layer	MAP	MRR	NDCG@10	P@10	avg
TREC	12	0.436	0.926	0.678	0.784	0.706
+BM25	5	0.437	0.947	0.682	**0.791**	0.714
	6	0.439	**0.953**	**0.690**	0.772	0.714
	12	0.420	0.912	0.659	0.749	0.685
+NER	4	**0.447**	**0.950**	0.685	0.788	**0.717**
	5	**0.444**	0.934	0.680	0.772	0.708
	12	**0.447**	0.944	**0.688**	**0.791**	**0.717**
+SEM	1	0.436	0.934	0.682	0.784	0.709
	4	0.440	0.928	0.682	0.779	0.707
	12	0.436	0.928	0.669	0.774	0.702
+COREF	5	0.442	**0.965**	**0.694**	**0.798**	**0.725**
	7	0.425	0.944	0.668	0.770	0.702
	12	0.442	0.948	0.681	0.788	0.715

Table 7.1: Test results for single additional task runs. The first row corresponds to a baseline that was solely trained for ranking on TREC2019. +[TASK] indicates multitask training on TREC2019 and [TASK]. Top 3 runs for each metric are highlighted in bold.

7.4 Ablation

The prior experiment has shown that explicitly infusing additional task information at specific layers, can result in increased ranking performance. Because of this, we further want to investigate whether this could be a valid augmentation technique in a limited data scenario.

For this, we conduct the following ablation study: Given a subset of TREC2019 query-document pairs, we resample multiple times from this subset and automatically create additional training samples for each of the tasks used in the MTL experiment. We then proceed to train using our proposed multitask method and repeat the experiment for different subset sizes.

8 Conclusion

8.1 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