Leipzig University
Institute of Computer Science
Degree Programme Computer Science, B.Sc.

Estimating Corpus Statistics with Large Language Models

Bachelor's Thesis

Max Staats
Born Jan 23, 1997 in Göttingen

Matriculation Number 3756555

1. Referee: Prof. Dr. Martin Potthast

Submission date: July 14, 2023

Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work.

Leipzig, July 14, 2023

. . . .
Max Staats

Abstract

This thesis explores the potential of transformer models to estimate corpus statistics by training them to predict word probabilities based on short contexts. A theoretical analysis demonstrates that training a transformer on contradicting examples, the Cross-Entropy loss is minimized if the prediction of the transformer exactly matches the distribution of the contradicting answers, opening the door for simple training algorithms. We further show that stochastic gradient descent successfully finds this minimum for both toy problems and transformer models. Experiments on Wikitext and Google n-gram show that when training on subsets of their n-gram distribution, memorization happens, i.e. the tranformer predicts the correct distribution of words for the training n-grams. While large improvements in the predictions on a test set promise generalization, we find that these improvements are independent of the training set used, showing that these improvements are not due to an adoption of the Corpus statistics. The results indicate that our methods are not yet ready to extrapolate the exact enumeration of n-gram distribution to unknown queries but provide insights into potential paths with larger transformers models.

Contents

1 Introduction **1**

2 Background and Related Work **3**
2.1 Tokenization . . . 3
2.2 Transformer-Encoder . . . 3
2.3 Transformer-Decoder . . . 5
2.4 Bert-Architecture . . . 8
2.5 Generating Text . . . 8

3 Feasibility Assessment **10**
3.1 Proof that Distributional Learning Happens with Cross-Entropy 10
3.2 Objective Minimization . . . 12
3.2.1 Fully Connected Networks . . . 12
3.2.2 Transformer Models . . . 14

4 Experimental Evaluation **16**
4.1 Wiki-Text . . . 16
4.1.1 Dataset . . . 16
4.1.2 Training and Evaluation . . . 17
4.2 Google n-Grams . . . 18
4.2.1 Absolute Frequencies Dataset . . . 19
4.2.2 Training and Evaluation . . . 20
4.2.3 Relative Frequencies Dataset . . . 22
4.2.4 Training and Evaluation . . . 23
4.3 Cross-Validation on Different Datasets . . . 24
4.3.1 Datasets . . . 24
4.3.2 Training and Evaluation . . . 25

5 Discussion and Conclusion **27**

Bibliography **29**

Chapter 1

Introduction

In the field of Natural Language Processing (NLP), corpora play a pivotal role as the foundation of modern language tools. A corpus refers to a large and structured collection of text samples that is organized for linguistic analysis and computational processing. These corpora can either serve as valuable resources for understanding various aspects of language or train large scale language models that perform translation tasks, sentiment analysis and text generation. One of the essential statistical techniques employed in corpora analysis is n-gram statistics, where n-grams refer to contiguous sequences of n items, which are typically words in the case of language analysis. N-gram statistics provide insights into the frequency of word combinations, thereby shedding light on the co-occurrence of patterns or collocations within a given corpus.

The usage of co-occurring patterns in language is something that native speakers are familiar with. However writing in a foreign language has many pitfalls. Depending on the level, it can be difficult to understand how native speakers use certain expressions or continue their sentences. A potential help in this regard can either be a translation tool which fully translates the native language into a foreign language or a writing assistants tool, that suggest a word at specific position or a continuation of the current sentence. With the emergence of powerful translation tools a full translation of your native language into foreign texts is a useful option. However, it comes at the cost of loosing the ability to understand a foreign language on your own. Using a writing assistance which suggest the use of specific words is different in this regard, as it allows you to construct your own text and potentially improve your language understanding.

A realization of such a writing assistance can be based on the n-gram statistics. By analyzing which words are being used in the corpora, given context of the writer, a set of suggestions is created. This approach is taken in Ref. [Stein et al., 2010] where given a query q with q = `"What is ?"` it is possible to see what words where used at the position of `"?"` in some underlying corpora. The words are ranked by their probability of occurring in this position to quickly suggest relevant words. Using different symbols then `"?"` it is also possible to get several words, word alternatives to an already used one, or other query based features.

While this exact evaluation of the underlying corpora is a powerful tool, it falls flat for small corpora as many relevant queries might no be present leading to irrelevant or no suggestions. But even for large corpora the number of patterns in the corpora that match the query falls off exponentially with the query size n leading to irrelevant or no suggestions at all for queries of size $n \geq 5$ [Wiegmann et al., 2022].

A potential solution to this problem is to extrapolate from the data of the corpora to queries which have no or unreliable statistics using language transformer models [Vaswani et al., 2017]. The goal of this thesis is to evaluate to which degree a transformer is able to mimic the exact statistics of an underlying corpus and if the extrapolation to unseen queries of such a transformer yields additional insight into the language used in the corpora. We show with analytic calculations in Chapter 3 that the loss minimum given a set of contradicting examples in a classification task for the cross-entropy loss is given by the prediction that follows exactly the presented example distribution. This means that if we train a transformer on the contradicting queries ”`i love [mask]`” with the answers ”`cats`” (three occurrences) and ”`dogs`” (seven occurrences) the prediction with minimal global loss is $(0.3, 0.7)$. The finding is verified numerically in Chapter 4 where we also show that this minimum is not only theoretically, but is also found by stochastic gradient descent.

These findings allow for a rather simple training technique where a transformer is trained with masked language modelling on contradicting short queries. After some training time we expect the transformer to perfectly model the distribution of the presented n-grams. The results in Chapter 4 show that memorization of the n-gram distribution happens as expected for our training method. Moreover, predictions of the transformer on a test set where improved significantly, however, as these improvements where independent of the underlying corpus used to train the transformer a final statement on the adoption of a hidden underlying distribution of the corpus could not be drawn.

Chapter 2

Background and Related Work

In the following we will discuss some of the theory on how transformer models are able to process natural language. This will help us to understand the later parts of this thesis where transformer models are trained to predict the statistics of a given corpus and help as a writing assistance. We will discuss the concept of the different layers involved and mainly focus on the original architecture presented in [Vaswani et al., 2017].

2.1 Tokenization

Before we can process words with a transformer model, we must first translate them into numbers. The first step in this translation is done by a human designed tokenizer. The two most common approaches are word-level tokenization and subword-level tokenization. Upper case is converted to lower case to avoid additional tokens. By going through the training data, we start adding words or subwords to the vocabulary continuously. Each of the new words gets an integer as a representation. Depending on the algorithm we can also neglect some symbols to keep the vocabulary in reasonable size. In the end we have a mapping that is able to map most of the training data into integers. When the architecture is fixed and we come to deployment of the model or fine-tuning there might still be words that are not characterized by tokenization. However, this is solved by a special token which is identified with all word that are not recognized.

2.2 Transformer-Encoder

After tokenization replaced the words by a first set of numbers, the Encoder starts its work by further transforming this set of numbers into hidden states, which are encoded as vectors in a high dimensional space. These vector representations should still represent the meaning of the original words but must be learned during training. In detail, this is done by having an embedding matrix in which each row represents one of the tokens that the tokenization process is able to create. This means that there are as many rows in the matrix as we have words/sub-words in our vocabulary, while the number of columns is given by the dimension of the word embedding. This vector representation of a word or sub-word has the main advantage that, in comparison to integer numbers, they can express contextual similarity by spacial distance. The values contained in the matrix which map tokens to embeddings are parameters that a transformer must learn and are hence unclear at its first creation. On top of this learnable transformation we also add positional encoding to the vectors by adding different "time-vectors" to the embeddings, depending on their position. These "time-vectors" are fixed from the start of training such that the network learns to interpret their meaning.

After calculating the word embeddings and the additional time-vector, the transformer has an "attention" layer. This layer works by mapping each of the embeddings ($\mathrm{emb_i}$) to another vector by also considering the surrounding words. This is done by calculating a query $Q_i = Q(\mathrm{emb_i})$ a key $K_i = K(\mathrm{emb_i})$ and a value $V_i = V(\mathrm{emb_i})$ for each of the embeddings via three matrices such that

$$Q_i = \boldsymbol{Q} \times \mathrm{emb_i} \quad \mathrm{K_i} = \boldsymbol{K} \times \mathrm{emb_i} \quad \mathrm{V_i} = \boldsymbol{V} \times \mathrm{emb_i} \ , \tag{2.1}$$

where the index i is running over all embeddings and the parameters of the matrices must be learned. The relations between the words are now expressed by calculating a new vector R_i for each of the original embeddings. This is done by scoring the relation that one embedding has to another with the dot product of the query, key pair. These relation scores are transformed into to "probabilities" $P_{i,n}$ by applying the softmax function to these scores. Hence, the P_{in} are defined as

$$P_{in} = \frac{\exp(Q_i * K_n)}{\sum_j \exp(Q_i * K_j)} \ , \tag{2.2}$$

where a normalization factor is omitted. Given the matrix P_{in} that encodes the relations that each word has with each other word, we can express the new embeddings R_i after the attention layer as

$$R_i = \sum_n P_{in} * V_i \ . \tag{2.3}$$

These new embeddings contain mostly their own V_i embedding if the score with itself ($Q_i K_i$) was high, but they can also contain significant portions of the other ones. While this describes the concept of attention, transformer use multihead attention which means that they have several attention matrices $\boldsymbol{Q}, \boldsymbol{V}, \boldsymbol{K}$ each producing there own attention mechanism leading to several new embeddings for each word.

After putting a lot of information in the individual vectors-groups by first mapping an integer representation of words to a space that encodes contextual similarity and second, adding an multihead attention layer that directly considers surrounding word we apply identical convolutional feed-forward neural networks to each of these vector-groups. This naturally produces another set of embeddings. However, after all these operations these embeddings are only loosely related to the original words or sub-words. To keep the connection to the original embedding-word and to improve learning in this deep transformer architecture the concept of residual learning [He et al., 2016] is adapted. This works by adding the original word embedding to the output of the feed-forward neural network. This reduces the complexity of learning as the network parameters can be driven to zero without losing information about the input. Furthermore, a batch normalisation layer is added before and after the feed-forward neural network to keep the variance of the neuron activations in a range which the neural network can easily interpret. This concept has proven to be useful in conventional feed-forward networks and is adopted in transformers [Ba et al., 2016].

In Figure 2.1 a schematic drawing of the computation from the original word embeddings (Positional Encoding) through the self-Attention layer into the batch normalisation is presented. The dotted line represents the residual connection which adds the original embedding to the output. This is followed by the feed forward neural networks and another batch normalisation. Again we have a residual connection such that the output that the attention layer produces directly contributes in the final result. A fully functional encoder is obtained by stacking several of these layers. In the original paper of [Vaswani et al., 2017] this number is six.

2.3 Transformer-Decoder

The Decoder part of the model uses a lot of the components that we already spoke above, but it also adds a so-called encoder-decoder attention layer to the stack of layers. As an input, the Decoder takes the embeddings of the previously computed words or, in case where no word has been predicted yet, a special "start of output" embedding. We again add a time vector to the embeddings to encode their position in the created sentence. The already predicted words are again put through a multihead attention layer with additional residual connection and batch normalization afterwards. In the following so-

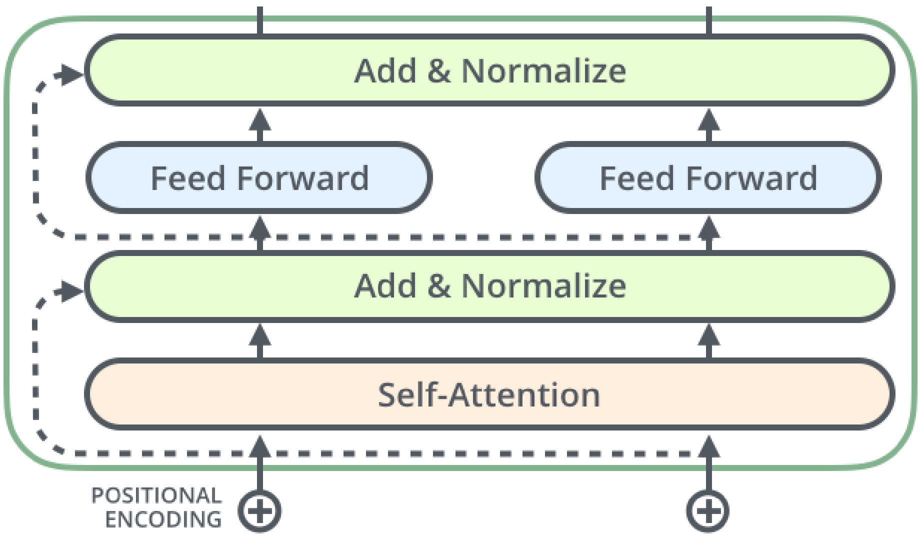

Figure 2.1: Schematic drawing of the flow that two input tokens take through a single encoder stack. The positional encoding goes through a residual connection and the self-attention layer into the batch normalisation. This is followed by another residual connection and a feed-forward neural network which are added together and normalized to complete a full block of the encoder. Figure adopted from Reference [Alammar, 2018].

called encoder-decoder attention layer the information about the input enters the Decoder. This is again realized by having three matrices $\boldsymbol{QKV}$ that create a query, key and a value. However, this time only the query is calculated based on the output of the previous attention mechanism. The key and the value are obtained by the output of the Encoder! This way we connect the already produced output of the Decoder with the information that was given to the Encoder resulting in new embeddings that contain both information. Again a residual connection together with batch normalisation is added. After the encoder-decoder attention layer, we find another feed-forward neural network similar to the architecture of the Encoder.

The schematic drawing of the Decoder can be seen in Figure 2.2. We clearly see the similarity to the Encoder with the only difference being that the Encoder-Decoder Attention layer adds the information of the Encoder to the Decoder (indicate by small dotted arrow). A complete Decoder again consists of several of these blocks stacked behind each other.

Another difference to the Encoder is that after several of the blocks described in Figure 2.2 we have a fully connected feed-forward neural network that connects all the outputs of the last block. While the number of parameters in this fully connected neural network can vary, the output dimension is again

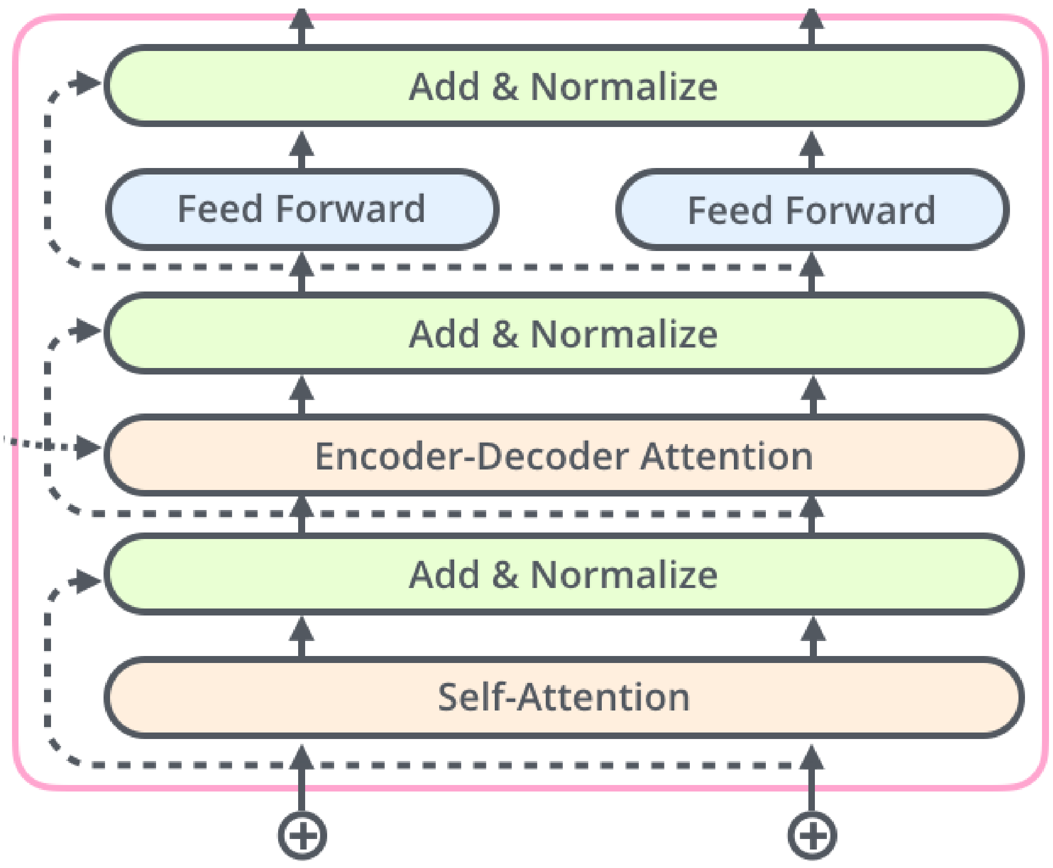

Figure 2.2: Schematic drawing of a Decoder in a Transformer model. The dotted arrow from the left indicates the information coming from the Encoder while the bottom entries are the output that has already been generated. Figure adopted from Reference [Alammar, 2018]

of the size of the vocabulary. Each of the output neurons encodes a token and after applying the softmax activation function we can interpret these outputs as probabilities that the specific token that is represented by the neuron should be the next one in the output sequence. To generate the output, we can choose the token with the highest probability and reenter the new output sequence into the Decoder until an "end of output" token is created. When generating text it is also possible to take a different word than the one with the highest probability to avoid repetitive behaviour. This will be discussed later.

2.4 Bert-Architecture

In this thesis we will primarily be dealing with the architecture of a Bert-like transformer [Devlin et al., 2018]. For this reason we will briefly discuss the specialties of this architecture here.

The initial input is embedded using WordPiece [Wu et al., 2016], then an additional vector is added that indicates the positional encoding. This representation of thee words is given to the Encoder stack which works just as described above. However, this time we are missing the decoder side and interpret the output of the last Encoder as the embedding of a predicted word. These embeddings are transformed into tokens using a final fully connected layer.

One of the advantages compared to the full transformer architecture is that information about words can now flow bidirectional e.g. from left to right and from right to left in a text. In the Decoder of the classical transformer model this was not the case. To train such a network the paper suggests Masked-Language modelling which hands the network a sequence of tokens, say `[123, 456, [Mask], 826]` and asks the network to predict the correct token at the `[Mask]` position based on the information given by the surrounding words. When applying the softmax activation function to the logits prediction of the transformer we get a probability distribution for what the masked word could be. This probability distribution is then treated as a classification task. The cross-entropy loss function is computed, and the derivative with respect to the logits is backpropagated through the network. Here, the missing words can be interpreted as one-hot encoded examples where the token that was maksed in the example is the correct answer.

2.5 Generating Text

The above discussed new transformer architecture lead to great improvements in the field of natural language processing as for example in question-answering problems [Haque et al., 2022]. We already discussed how next word prediction can be done on a bert-like transformer using masked language modelling. A general Encoder-Decoder architecture predicts the next words in a similar fashion as it considers all the past output to generate a new word. The training case of the transformer can easily be utilized as text generation by iterative next word prediction. In the best case scenario, we would get a structured answer to our input just by next word prediction. However, when always taking the word with the highest logit value, and hence also highest probability, as the next word prediction in text generation, the model might get stuck in certain loops or produce meaningless output [Holtzman et al., 2019].

Other methods are based on sampling from the top k words, where the sampling can be uniform or be weighted by probability [Holtzman et al., 2019]. The later case has clear benefits as the number k can be chosen much larger in this case without producing unrelated output. However, we find that in the case of Netspeak [Wiegmann et al., 2022] the prediction of the next word distribution of a transformer model often disagrees with the actual probabilities in natural language on which it was trained. It appears that the top words are too likely, while other words are suppressed. This has the issue that probability based sampling is not really an option when generating text using the transformer. In the following we will analyze why the probability distributions predicted by the transformer could differ from the distribution found in the training text.

Chapter 3

Feasibility Assessment

We want to look at a setup where a transformer is given a query `q=''text [Mask] text''` and has to predict the word hidden under `[Mask]`. In the training process we might train the transformer with the query `''A [Mask] B''` where the correct answer for `[Mask]` is `''dog''` in six of the ten occurrences, while the four other examples have `[Mask]=''Human''` and `''[Mask]=''Cat''`. It is now a question of principle how a transformer or even any other neural network is answering the question `''A [Mask] B''` after training sufficiently long on the examples to reach a (potentially local) loss minimum. Two possibilities appear reasonable:

- The neural network tries to be as correct as possible on the majority and predicts `''dog''` as a certain answer with nearly 100%.
- While `''dog''` is the favorite with 60% the other examples did mark their spot and receive their 40% probability.

An important note is that even though neural networks can approximate any function [Hornik et al., 1989] it is not a priori clear that the second case will be realized as we are not approximating a function as there are two different labels for the same example. We are not actually training it to be the function realized by the second case (even though this would be possible by merging the examples and having a probability distribution as a label).

3.1 Proof that Distributional Learning Happens with Cross-Entropy

When training the transformer, we move the weights in a high dimensional weight space to minimize the loss. In our case we have a classification task with the network h and the Cross-Entropy loss function. The examples will be denoted as ξ^α and the corresponding labels are σ^α. This gives the global loss

$$L(\xi^\alpha, \sigma^\alpha) = \sum_\alpha \sum_i -\sigma_i^\alpha \log(h(\xi^\alpha)_i) \tag{3.1}$$

In the case where we want to optimize this loss for a query where different one hot encoded solutions exist we can rewrite the loss as a sum over all possible answers

$$L(\xi, \sigma^\alpha) = -\sum_\alpha p(\sigma^\alpha) \log(h(\xi)_{1_\alpha}) \tag{3.2}$$

where $p(\sigma^\alpha)$ is the probability of σ^α to occur as a solution in the examples and the sub index 1_α denotes the index where the label of the example is one. Calculating the minimum of this expression also reveals how the other contradicting examples are minimized as the loss is additive. For the above equation we want to know what prediction $h(\xi)$ minimizes the loss under the constrain that the prediction is a probability distribution i.e. the entries are non negative and sum to one. Further simplifying the notation, we now write $p(\alpha)$ for the probability of σ^α to occur in the examples and x_α for the output of the network at the position where the solution σ^α requires a one. This gives us the simple equation

$$L = -\sum_\alpha p(\alpha) \log(x_\alpha). \tag{3.3}$$

We find the minimum of the above equation with respect to the x_α and the probability condition by rewriting the loss with Lagrange multipliers

$$L' = -\sum_\alpha p(\alpha) \log(x_\alpha) + \lambda \left(1 - \sum_\alpha x_\alpha\right). \tag{3.4}$$

Calculating the derivatives yields

$$\frac{\partial L'}{\partial \lambda} = 1 - \sum_\alpha x_\alpha \overset{!}{=} 0 \tag{3.5}$$

$$\frac{\partial L'}{\partial x_\alpha} = \frac{-p(\alpha)}{x_\alpha} + \lambda \overset{!}{=} 0 \rightarrow \lambda = \frac{p(\alpha)}{x_\alpha}. \tag{3.6}$$

We find that the only set of x_α that fulfills the second equation for all α are the ones given by $x_\alpha = p(\alpha) \times c$ where $c = 1$ is fixed to one by the first equation. We hence showed that

$$x_\alpha = p(\alpha) \tag{3.7}$$

minimized the loss function. So even though the top prediction will always be given by majority vote of the examples, the loss landscape is aware of the example distribution and (if the minimum is found and cross entropy is used) the network will correctly display the probability distribution of the examples.

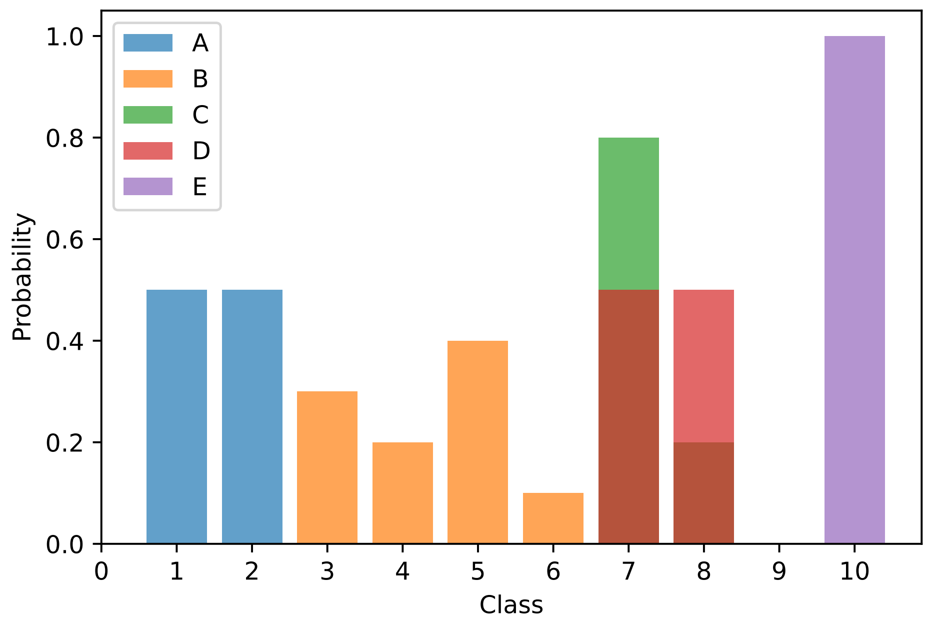

Figure 3.1: Distribution of the labels onto the ten classes for the symbols $\{A, B, C, D, E\}$. A variety of distributions is tested from simple one class solutions (symbol E) to a more divided distribution (symbol B) or overlapping distributions (symbols C, D).

3.2 Objective Minimization

We showed in the previous section 3.1 that when a neural network is trained with cross entropy on contradicting examples, i.e. there are different (one-hot encoded) target vectors for identical input encodings, the minimum of the loss function is given by the network output that models the distribution of the labels for the given example. We might assume that such a minimum is found by gradient descent, however, to make learning feasible we will only do stochastic gradient descent i.e compute the gradient based on small parts of the training set. This training protocol induces an additional uncertainty on if the "correct" minimum can be found as showing contradicting examples in different batches might lead to poor learning in general.

3.2.1 Fully Connected Networks

To test this experimentally we will first construct a simple neural network which is trained on symbols with contradicting one hot encoded labels. The

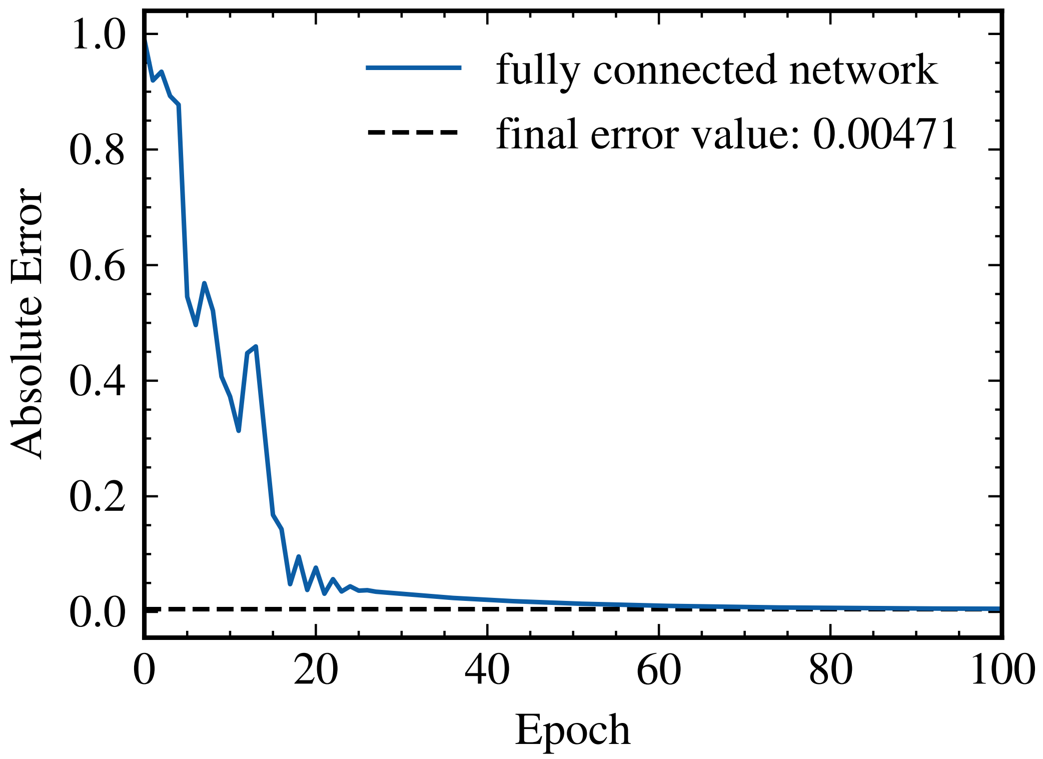

Figure 3.2: Learning progress of a simple feed forward neural network when trained on contradicting examples. The absolute error is given by the Manhattan distance between the output distribution given by the network and the actual distribution of the labels as given in the training data. We find perfect learning which is in agreement with the theoretical prediction.

neural network is a simple feed forward architecture with an input size of five, three hidden layers of size $[100, 50, 50]$ and output layer of size 10. The activation function is ReLU for all but the last layer where a softmax activation is used. The weights are initialized Glorot uniform [Glorot and Bengio, 2010], we use a learning rate of 0.005, momentum of 0.95 and the learning rate decays exponentially with decay constant 0.95 per epoch. We use a batch size of one to make it as hard as possible for the network. We define five symbols as

$$A = [1, 0, 1, 1, 0], \quad B = [0, 0, 1, 1, 1], \quad C = [1, 1, 1, 0, 0],$$
$$D = [0, 1, 0, 1, 1], \quad E = [1, 1, 0, 0, 1]$$

and the related distributions are given in Figure 3.1. Symbol A has a 50%, 50% split over the classes one and two, symbol B has a distribution which is split over the classes three to six and symbol C and D share a distribution on the classes seven and eight where C is unbalanced and D is not. As a final

Table 3.1: Final prediction of a transformer that was fine tuned on the queries below. In $(x|y)$ the x is the probability predicted by the network, and the y is the probability of this answer to occur in the training set.

Query	Answer 1	Answer 2	Answer 3
this is a ?	god (0.6314|0.625)	human (0.2510|0.25)	dog (0.1140|0.125)
we ? chill	always (0.3747|0.37)	love (0.3099|0.333)	never (0.3120|0.296)
? is dead	he (0.3967|0.4)	humanity (0.3829|0.4)	god (0.20525|0.2)

testing symbol we have E which is always combined with class ten. Class nine remains empty. The network is now trained on fifty examples per epoch, each symbol having ten examples with one hot encoded labels. For the symbol A we have five examples which have one as the correct label five with label two.

We are now interested in the progress that the network h makes in learning the probability distribution of each example. To monitor the progress, we define the distributional error e as

$$e = \sum_{i \in \{A,B,C,D,E\}} |h(i) - p(i)|_1 / 2 \tag{3.8}$$

where $p(i)$ is the vector which encodes the distribution given in Figure 3.1 and the norm is the sum over the absolute values of the differences also know as the Manhattan distance. This error is in the range of $[0, 1]$ where zero corresponds to perfect agreement and a value of one indicates that no overlap exists at all. The error can be intuitively interpreted as the probability weight that is put into the wrong class.

The results when evaluating this metric over 100 epochs are shown in Figure 3.2. As theoretically analyzed we find that the network perfectly learns the probability distribution given by the labels.

3.2.2 Transformer Models

To further strengthen the empirical results additional simulations on the pre-trained distill-bert[1] [Sanh et al., 2019] are made. The examples that the model was trained on are found in Table 3.1 where the ''?'' denotes the ''`[Mask]`'' token and the three possible answers that where assigned as correct (with corresponding probability) are shown in the remaining columns. We again use one hot encoded labels where the number of examples per label exactly mimics the probability distribution. The examples are put in a pytorch dataset and we use

[1]`https://huggingface.co/distilbert-base-uncased`

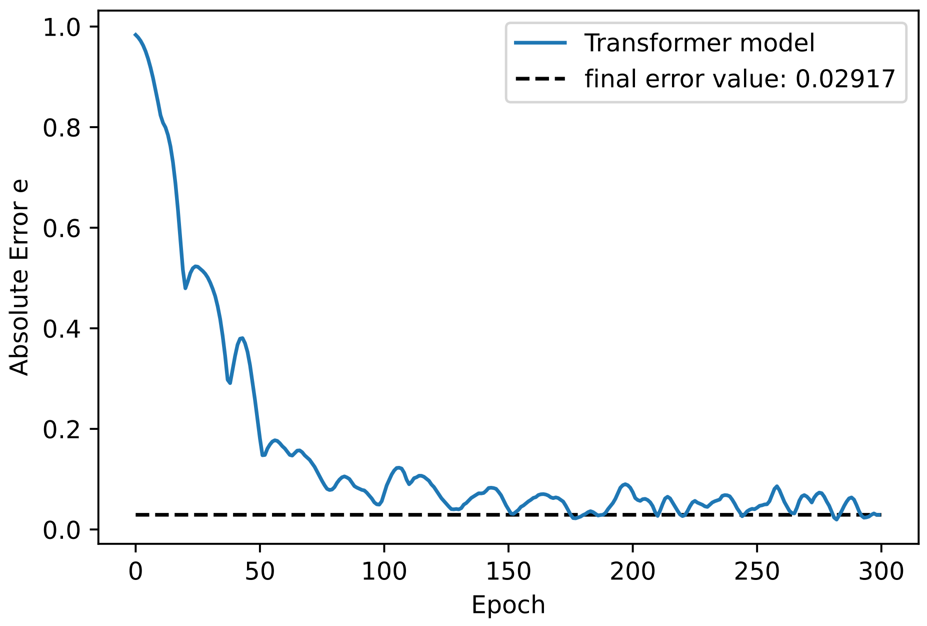

Figure 3.3: Learning progress of a transformer trained on three queries, each having a different distribution of possible answers. The training is performed with full batch learning on the one hot encoded examples. We find that even though it is harder to find the optimal weights of the network, perfect learning is possible.

the same metric as defined in Equation 3.8 to evaluate the training progress of the transformer. To keep the computational effort small we only train the model head. We use the AdamW optimizer with full batch learning and a learning rate of $5e-5$ to optimize the weights of the model head. The result are shown in Figure 3.3 where we again find almost perfect learning. The final absolute error of $\simeq 0.03$ means that only 1.5% of probability is put into wrong words. Further improvements are certainly possible when simply adjusting the hyperparameters. To accurately find the final minimum a decaying learning rate might be the key as fluctuations are high. The final predictions of the finetuned transformer are shown in Tabel 3.1.

While a direct comparison of the training time might be misleading due to the different optimizers, batchsizes and models it is still interesting how many epochs are necessary for the transformer to fully memorize the three queries. Given this insight together with the analytic result we might already be able to tell why the predicted distribution of words on a finetuned transformer differs from the distribution it was trained on. The training time was simply not sufficient to learn the examples by heart.

Chapter 4

Experimental Evaluation

We showed in the previous chapter 3 that it is theoretically possible to finetune a sufficiently large transformer on a corpus to a degree where the answer of the transformer to a given query is identically to scanning the whole corpus for the exact probability distribution of the possible words. In terms of fast query evaluation on a given corpus, this is already an interesting result. However, it is still an open question if a transformer that is fitted to such a high degree is still able to produce useful answer to queries that where not shown in the training data. In the best case scenario, a transformer would adopt the language of the corpus and speak similarly on unknown queries. In the worst-case scenario, the transformer would forget the rules of the language and produce outputs which only use the most frequent words of the training data in situations where they might not fit. In the following we will investigate this question on several datasets and training setups.

4.1 Wiki-Text

We will start with a subset of the corpus 'wikitext-103-v1' [Merity et al., 2016] by constructing a train and a test set. We train the transformer on the training set and test afterwards if the answers of the transformer became similar to the answers of the queries in the test set.

4.1.1 Dataset

The dataset construction is done by splitting the text into chunks of similar length, where the length is defined based on the number of white spaces. We discard strings with non-alphabet characters. Afterwards, we create as many examples as there are words in the query by masking each of the words once. As this procedure creates large amounts of queries rather fast, we only consider the first 200 rows of ”wikitext-103-v1”[1] to keep training feasible. To get the distribution of answers for each query we have to group all identical ones. We then split the collection of grouped queries into a train and a test dataset such that no query of the train dataset is contained in the test dataset. For example: Given the three previous queries: `”this is a ?”`, `”we ? chill”` and `”? is dead”`, we might train the network on all examples containing the queries `”this is a ?”` and `”we ? chill”` and check the generalization error with the query `”? is dead”`. An inconvenient part of this training setup is that if we want to know the training error we need a training evaluation set containing the probabilities for each query as we can not evaluate the training error on the one hot encoded examples during training.

The training set contains (2/3) of the data while the test set gets the other (1/3) such that the queries of the test and train data are disjoint sets. We create based on these two sets the correct distributional answer for each query and use the distributional answers for training and test data evaluation.

4.1.2 Training and Evaluation

To keep the training fast we will only train the model head using the AdamW optimizer with a learning rate of $5e - 5$. We still use the Manhattan distance between the prediction and the correct distribution as an evaluation metric as described in Equation 3.8. We again train on ”distilbert-base-uncased”[2] from Hugging face.

We see in Figure 4.1 that the training error converges to zero after around 800 epochs, which shows that perfect learning is possible even for a large number of different examples. Furthermore, we find that the network does not loose its ability to generalize to the fitting procedure. The generalization performance increases monotonically with the memorization of examples from the same underlying text which is surprising as one could expect overfitting to set in at some point. Just by complete memorization of queries from a text the transformer is able increase the prediction accuracy on other queries of the same text. This raises the question whether complete fitting of a training set might give the best generalization with respect to the style of the training set.

Some critique to the testing procedure could be raised as the creation of the dataset does not separate training and testing data perfectly. While "`[A B C ? E]`" is in the training set "`[A B ? D E]`" might be in the test set. In the following we will consider further datasets and also fix the issue of a potential train-test leakage.

[1]https://huggingface.co/datasets/conceptofmind/wikitext-103-v1-clean

[2]`https://huggingface.co/distilbert-base-uncased`

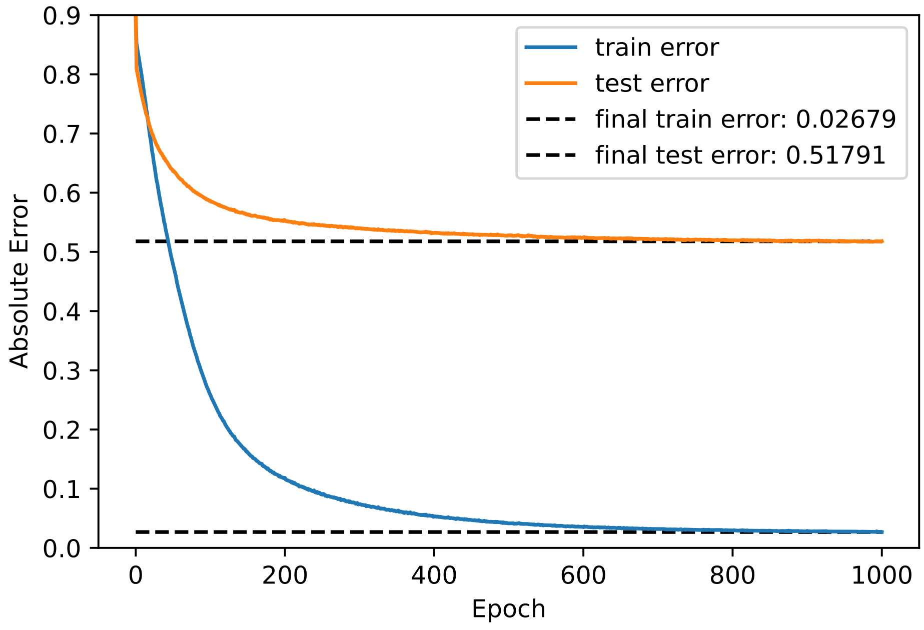

Figure 4.1: Training and test error on a dataset consisting of $\simeq 12,000$ training queries with potentially different one hot encoded answers and $\simeq 6,000$ test queries with distributional answers. We find a perfect fit of the training data and a monotonically decreasing generalisation error.

4.2 Google n-Grams

The dataset *Google n-gram* that we will use in the following is based on the Google Books n-gram Viewer[3] where the occurrence of words or phrases in books ranging from the year 1500 up to 2019 is analyzed in their relative frequency. The resulting database initially consisted of $\simeq$ five million books reaching up to the year 2012 [Michel et al., 2011] and has since been expanded with books to the year 2019.

By indexing the occurrence of n-grams from Google n-gram, the writing assistant Netspeak is able to give advice on which words are being frequently used in a certain context. The resulting dataset is stored by sorting queries of equal length alphabetically while additionally storing their frequency in the database. A search algorithm on sorted data can hence quickly find all answers to a given query and rank them by their occurrences. This allows for a word suggestion that is in line with commonly used language. However, for a larger

[3]`https://books.google.com/ngrams/`

context, the amount of data available diminishes such that suggestions might become odd or simply not available at all. In this context it could be help full to have a transformer that memorized the dataset such that the answers of Netspeak and the transformer align in the case of frequently occurring phrases. In cases where the query is poorly represented in the data due to length or other factors, the hope is that a transformer would still give good suggestions that align with "common language understanding".

4.2.1 Absolute Frequencies Dataset

To train a transformer on this indexed dataset using masked language modelling we must first convert the indexed phrases back into examples from which a transformer could learn. One simple approach would be to convert each indexed phrase `"hello world 30,765"` into exactly as many examples as the phrase occurred in the Corpus, i.e. 30,765 examples of `"hello [mask]"` and `"[mask] world"`. However, this method is completely unfeasible as, even if we limit our self to only 3-gram queries, we have 98 data files containing only 3-gram queries each containing ten million indexed queries, meaning that the total number of examples would be $\simeq 10^{11}$ if we estimate the average number of occurrences of a phrase in the corpus to be 100 (verified in Subsection 4.2.2).

One possible approach to make the training more feasible is to divide the number of occurrences for each n-gram by a constant number c. This clearly reduces the number of created examples by a factor similar to c but has the drawback that rare words might not be considered if c is chosen to large. To obtain a first grasp on how large the dataset is compared to our computational resources and how good the memorization and generalization of the transformer work, we consider the following setup: We restrict ourself to n-grams of the size three and neglect all n-grams which contain non alphabetic characters. Furthermore, we neglect queries which would have a multi token answer, restricting our self to a more simple language. Finally, we only create examples based on the last word i.e. for `"hello world 30,765"` we create 3,076 examples `"hello [mask]"` as c is chosen to be ten. As the size of the dataset still remains to large to experiment with quick training runs, the final reduction cuts the file-length to 50,000 lines, a reduction by a factor of 200.

In Table. 4.1 we see how increasing the number of considered training files increases the number of examples we are training on. For small numbers of files the relation is non-trivial as the strict criterion that needs to be fulfilled for an example to be in the training set can easily exclude a whole file as it is seen when going from one to two or from three to four files where no increase in the number of examples is seen at all. Furthermore, we find that our estimate of having around 100 examples per line is roughly verified as we

Table 4.1: Number of training examples generated from a certain number of considered 3-gram files. The examples are generated by masking the last word of a phrase and adding as many examples to the training set as their count divided by ten. The second column denotes the number of one hot encoded examples while the third column counts how many different queries are in the training set independent of the answers and how often they occur.

#Train Files	#Examples	#distr. Examples
0.01	4,684	100
1	686,556	6,818
2	686,556	6,818
3	2,072,607	13,584
4	2,072,716	13,585
5	2,072,716	13,585
6	3,635,813	19,419
20	25,455,548	70,572
40	48,437,190	136,126

consider $40 * 50,000$ lines yielding $48,437,190 * 10$ examples meaning we are at around 50 examples per line.

4.2.2 Training and Evaluation

We test the memorization of the training examples as in previous setups of Chapter 4 via the distribution of possible answers given a query. The error given the distributional answers is again measured using Equation 3.8. We also consider a "generalisation error" where we exclude a specific file from the training process and look if the predictions of the transformer start to align better with this file during the training process. The idea behind this improvement would be that the corpus has a corpus specific language whose patterns are to some degree learned during training. To find this potential generalization in dependence of the size of the training dataset, we consider the pre-trained transformer Distill-Bert[4] [Sanh et al., 2019] and train it on a varying number of files containing 3-grams, while the test-file is always excluded.

We start with pretrained-weights to get results faster and find a potential adaption to the language of the dataset. To vary the size of the dataset, we include increasingly many files alternating from one to 40 files where training with 40 is already close to what is trainable in one day on an Nvidia Ampere-100 GPU.

[4]`https://huggingface.co/distilbert-base-uncased`

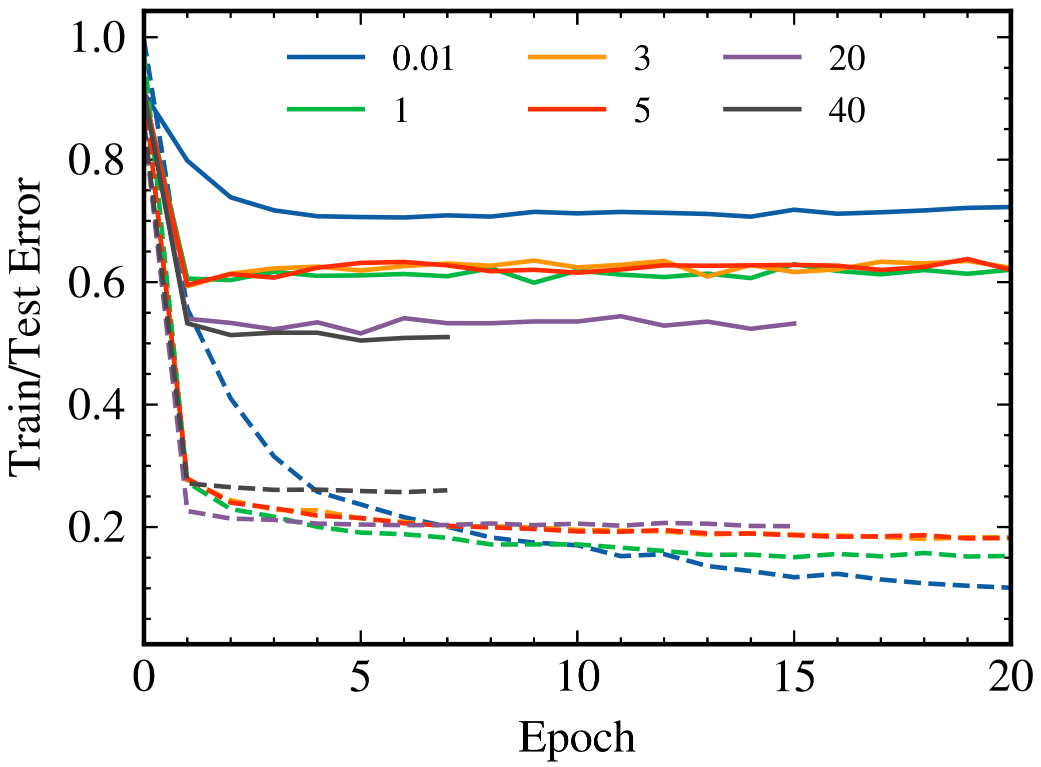

Figure 4.2: Training and test error on subsets of Google n-gram. The number indicates how many training files where considered in the training process. The testing was done one a separate file which was not considered during training. The training error is shown in dashed while the test error is a full line.

The considerations in Subsection 4.2.1 about the unequal distribution of examples in the files help us to understand the results displayed in Figure 4.2 where we show the training process of the pretrained distill-bert model when training with different numbers of files. A first finding is that for relatively small amounts of training data (see Table 4.1 for details) significant improvements on the test data (full lines) are obtained. This is most likely not due to the model learning a language similar to the one represented by Google n-gram but the model learning the preprocessing patterns of the data. Non-alphabet strings and complicated multi token words are never the correct answer meaning that just learning this pattern could give clear improvements. However, beside this potential effect, we find that increasing the amount of training data clearly reduces the generalization error. The reason for the early ending of the curves ”20” and ”40” is due to computational limitations. The shown results are in good agreement with the hypothesis that the language model learns a more general distribution of words in Google n-gram, hence reducing the test

Table 4.2: Number of training examples generated from a certain number of considered 3-gram files. The examples are created by grouping identical queries, calculating the probability distribution of the answers to then create one example per 2% probability that a given one-hot encoded answer is correct. The second column denotes the number of one hot encoded examples while the third column counts how many different queries are in the training set independent of the answers.

#Train Files	#Examples	#distr. Examples
0.01	4,928	100
1	336,699	6,818
3	669,121	13,585
5	669,121	13585
20	3,456,103	70,572
40	6,633,559	136,126

error.

4.2.3 Relative Frequencies Dataset

As our current learning method is still rather slow in learning such a large dataset we will look for a different method in the following. One simple approach would be to increase c, the number by which we divide the occurrence of each query, to reduce the computational effort by an arbitrary amount. However, this would erase rare queries and leave us with a dataset consisting only of omnipresent phrases. As already discussed, training with a distributional approach is also not an option, as this leaves us with giant sparse vectors for each query.

To combine the distributional and one-hot encoded approach, we suggest a third method which first calculates the distributional answer for each query and then "fine grains" this distributional answer into a set of one-hot encoded examples. For example if "`i love [mask]`" had the answers "`cats` (48%)", "`dogs` (50%)" and "`wolfs` (2%)", we would use a certain resolution (1% = 1 example) to convert this distribution of queries into examples. For a lower resolution (25% = 1 example) we might neglect the answer "wolfs" for the benefit of having way less examples.

This method has the advantage that the example set is not dominated by omnipresent phrases but contains a portion of everything. However, this is also its biggest draw back as it weights all queries evenly independent on the absolute occurrence in natural language.

The number of generated examples when taking 2% = 1 example while keeping the rest of the preprocessing unchanged are shown in Table 4.2.

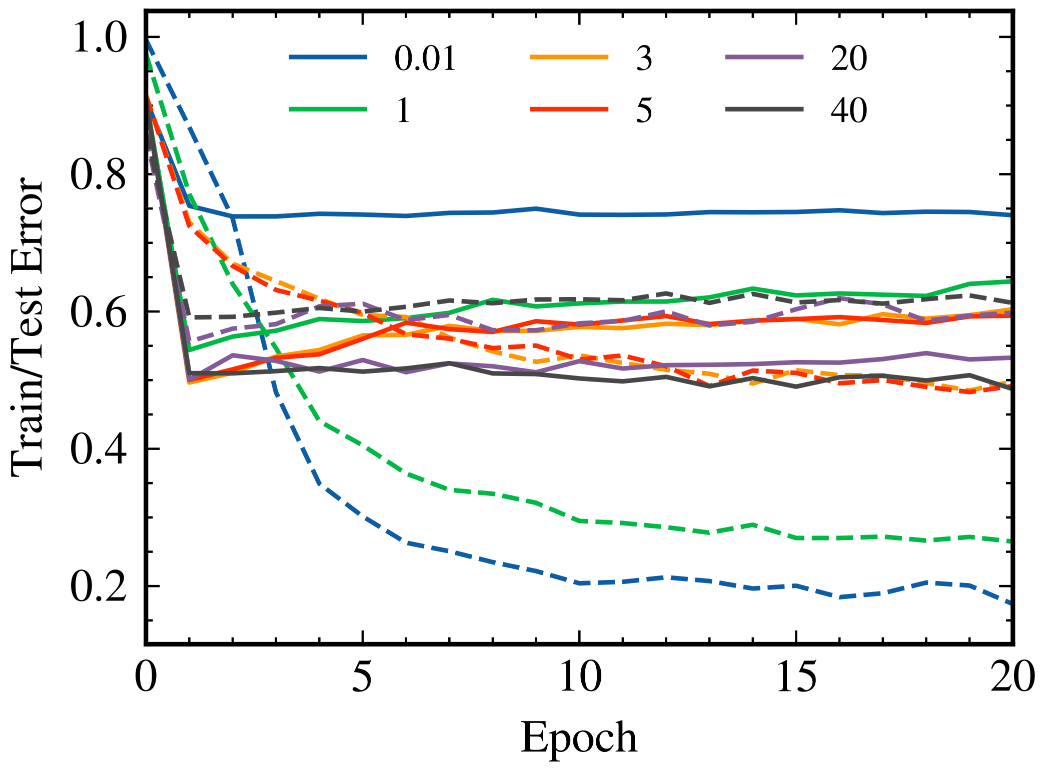

Figure 4.3: Training and test error on subsets of Google n-gram. The number indicates how many training files where considered in the training process. The testing was done one a separate file which was not considered during training. The training error is shown in dashed while the test error is a full line.

4.2.4 Training and Evaluation

The results of the training progress are displayed in Figure 4.3 where the same testing procedure as in the previous Figure 4.2 is being used. We find that:

1. The training error (dashed lines) is much higher than in the previous case which is not surprising as in the other training setup, learning a few frequently occurring phrases could cover a majority of the training set while this is not the case here anymore.

2. The more files we include the harder it is for the transformer to memorize the training data leading to a higher training error in agreement with the previous results.

3. The test error (full lines) is in all cases comparable with the error of the previous method indicating that it might no make a difference which method is being used.

4. Interestingly the test error can be even lower than the training error if the training set is large. This might possibly be due to the fact that the transformer still prefers to give ”simple” answers that occur frequently in common language. However, the training data set does not weight the ”simple queries” more than the hard ones therefore making it the harder set.

4.3 Cross-Validation on Different Datasets

To further investigate the role that the distribution of n-grams in a certain dataset plays in the test accuracy improvements, as compared to learning some structure of the training process, we will consider two more datasets in the following.

4.3.1 Datasets

The first dataset is based on the New York Times Corpus [Sandhaus, 2008] consisting of 1.8 million published articles. The second one is the Brown Corpus [Kučera and Francis, 1967] which consist of various genres published in 1961. Both Corpora are first converted into an n-gram count format i.e. all text is converted into n-grams of the following format: ”`the dog barks 43`”, where the 43 indicates the number of occurrences of this phrase in the corpus. Afterwards, two datasets are created based on their n-gram counts, following the procedure of Subsection 4.2.1. The last word of each n-gram gets masked (for the previous example ”`the dog [mask]`” and we add `count (43)` examples to the dataset with the answer being the masked token (”`barks`”). Note that we do not divide the count by a constant in this case as the datasets are already small enough.

To test the language adaption of a model to the corpus, it is trained on, we create three datasets of similar complexity based on the three Corpora Google n-gram, Brown and New York Times. Here, complexity means that we find a balance between similar amounts of different queries (”`the dog [mask]`” vs ”`a cat [mask]`”) and the total number of examples. As the smaller Brown Corpus has rather low counts on most queries as compared to Google n-gram the total number of queries in relation to the total number of examples is quite different for the two. The final dataset sizes are as follows: Google n-gram has $25,455,548$ examples, New York Times has $2,912,325$ examples and the Brown dataset has $595,694$ examples. Furthermore, we create three test sets based on the three Corpora which now contain distributional answers, e.g. the for ”`the dog [mask]`” the correct answer answers is not a single word but a set

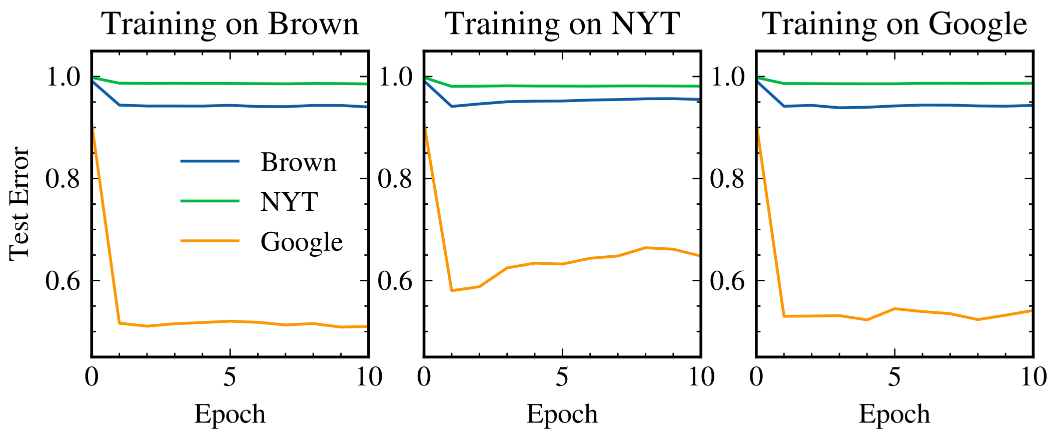

Figure 4.4: Language adaption when training a transformer model on three different datasets (Brown, New York Times, Google, left to right). We additionally keep test sets of each of these Corpora to identify a potential generalization. We find that no dataset specific language adoption occurs as the test results are not improved when testing and training a language model on the same dataset as compared to testing and training on two different ones. The most likely effect of training is the learning of the preprocessing.

of words with different probabilities corresponding to their occurrences in the Corpus.

4.3.2 Training and Evaluation

We keep the training identical to the one used when previously training Transformer models (see for example beginning Section 4.1.2). After training on one of the three datasets, we compute the error (Equation 3.8) on each of the three test sets to compare to which degree improvements are found only found on the corresponding test-set, indicating language adaption, and to which degree the improvements can be found on all test-sets, corresponding to memorization of the preprocessing.

The results can be seen in Figure 4.4 where the three different graphs show the training on the three different datasets. We find that in this case, the dataset which is being trained on only plays a minor role. The "generalization" to the Google test set when training on the Brown dataset (first graph, yellow line) is almost identical to the generalization to the Google test set when actually training on the Google train set (third graph, yellow line). Similar results are found for the other two datasets, indicating that no learning of an "underlying language distribution" has happened but as simple adoption of the transformer to the preprocessing. As the rather small improvements for the

Table 4.3: Improvements when training on the three datasets Google, Brown and New York Times for ten epochs. Looking at the columns, we find that it makes no significant difference if the language model is trained on the dataset from which the test set is created, or a different one.

Train Dataset	Google	NYT	Brown
Google	$0.9089 \rightarrow 0.5320$	$0.9981 \rightarrow 0.9867$	$0.9915 \rightarrow 0.9419$
NYT	$0.9089 \rightarrow 0.6614$	$0.9981 \rightarrow 0.9814$	$0.9915 \rightarrow 0.9566$
Brown	$0.9089 \rightarrow 0.5087$	$0.9981 \rightarrow 0.9861$	$0.9915 \rightarrow 0.9433$

Brown and New York Times dataset are hard to read of from the graphs, they are additionally provided in Table 4.3.

The reason for why the test error of Google n-gram improves so much more than the one of the other two is not completely clear to us. One potential reason might be that the test error starts considerably better than the one of the other two datasets. If, from the Transformers perspective, a set of answers can be ruled out due to "knowledge" of the preprocessing this has a much larger effect on the test error if there is already some knowledge of potentially correct answers. This can be understood if one thinks about betting on to 1000 potential outcomes but we get the hint (from preprocessing) that most of the answers are not possible but only 100 should be considered. Our chances increase by a factor of 10 for random guessing. For non random guessing it is a little more complicated but on average we have that the following behaviour: If i was already rather certain before on 20 answers we improve the error from 95% (1/20) to 50% (1/2) while someone that was much more uncertain before e.g. had 200 potentially correct answers only improved from an error of 99.5% to 95%. A much smaller absolute improvement. Together with the effect that the preprocessing simplifies the language, something that might benefit the Google test set more than the others, this might explain the much higher test accuracies.

Chapter 5

Discussion and Conclusion

In this thesis we evaluated the potential of transformer models to estimate corpus statistics. Specifically, we tried to train some transformer, to predict the probability for each word that could possibly come after a short context, such that the predicted probability matched the corpus statistics on which it was trained. This was done by first examining the theoretical training setup in Chapter 3. By analytically calculating the minimum of a classifier trained with contradicting one-hot encoded examples using the Cross-Entropy loss, we showed that the optimal prediction exactly matches the distribution of the examples. Given ten identical training inputs of which three labels give class one and seven labels give class three, the global loss is hence minimized by the prediction (1 : 30%, 3 : 70%). This allows us to train networks using one-hot encoded examples.

A second step in setting up the training with transformer models was to check whether stochastic gradient descent would be able to find a minimum which predicts the correct distribution, even when the gradients may vary strongly. We found in Section 3.2 that the minimum was indeed found for simple toy problems in multi layer perceptrons and extended the work to transformer models. Here, the training process took many epochs more than in the toy case, but eventually the distribution of the training data was perfectly mimicked.

Given the guarantee that the transformer model is able to memorize a distribution given our simple, non distributional, examples we asked the question whether a transformer that memorized a portion of corpus adapts a language which is similar to the one used in the corpus. In Section 4.1 we try to tackle this question by creating a train and a test set from a subset of 'wikitext-103-v1'. By masking words in short text-chunks we created queries ''`A [mask] B`''. For the test set we group all identical queries e.g. all queries ''`A [mask] B`'' and examine the distribution of words that are possible at the token position `''[mask]''`. We tracked the difference of the transformers prediction and the correct distribution at the `''[mask]''` position with the surprising result that the predicted distribution went from no alignment at all to a "somewhat similar" distribution (see Section 4.1 for details). Expanding the work towards the large dataset Google n-grams in Section 4.2 we again found a significant improvement on the test set when training on disjunct one-hot encoded example sets. Training with increasing portions of the dataset while keeping the test set identical yield the result that more data lead to better generalization. In 4.3 the generalization results where put to the test as we used two additional Corpora, the New York Times and the Brown Corpus for training. Training on one Corpus while testing on the other three we found that the improvements on the test set where independent of training set used, showing that the observed improvements where based on an adaption of the transformer to the preprocessing and not due to language adaption.

Given these results it remains inconclusive to which regards the transformer generalizes to the distribution of the underlying corpus when memorizing the test set. Further experiments with a less restrictive preprocessing should be done to test whether generalization can be observed. Additionally one could adopt how large language models are usually trained to reduce the effort in creating the train dataset. In our case the distribution of phrases was exactly enumerated prior to training such that the one hot encoded examples where created before the start of training based on the know distribution. However, we believe that for training on a large dataset a simple masked language modelling approach based on short inputs would also allow for an exact memorization of the distribution while greatly reducing the effort of creating the dataset. The training protocol would be to continuously go through a text, feeding the network chunks in which one word is masked which is identical to our one-hot encoded example training if done repetitively.

We conclude that the presented training methods are not ready yet to extrapolate beyond exact enumeration of n-gram distribution in applications where reliable statistics of Corpora are used. However, we set a theoretical basis on how a potentially larger transformer could be trained on a Corpus to memorize its distribution, while showing some of the difficulties in trying to define "generalization".

Bibliography

J Alammar. The illustrated transformer. `https://jalammar.github.io/illustrated-transformer/`, 2018. Accessed: 2023-04-23.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

Mubin Ul Haque, Isuru Dharmadasa, Zarrin Tasnim Sworna, Roshan Namal Rajapakse, and Hussain Ahmad. " i think this is the most disruptive technology": Exploring sentiments of chatgpt early adopters using twitter data. *arXiv preprint arXiv:2212.05856*, 2022.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

Kurt Hornik, Maxwell Stinchcombe, Halbert White, et al. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Henry Kučera and Winthrop Nelson Francis. *Computational analysis of present-day American English.* Brown university press, 1967.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K Gray, Google Books Team, Joseph P Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, et al. Quantitative analysis of culture using millions of digitized books. *science*, 331(6014):176–182, 2011.

Evan Sandhaus. The new york times annotated corpus. 2008. doi: 11272.1/AB2/GZC6PL. URL `https://catalog.ldc.upenn.edu/LDC2008T19`.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Benno Stein, Martin Potthast, and Martin Trenkmann. Retrieving customary web language to assist writers. In *European Conference on Information Retrieval*, pages 631–635. Springer, 2010.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Matti Wiegmann, Michael Völske, Benno Stein, and Martin Potthast. Language models as context-sensitive word search engines. In *Proceedings of the First Workshop on Intelligent and Interactive Writing Assistants (In2Writing 2022)*, pages 39–45, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.in2writing-1.5. URL `https://aclanthology.org/2022.in2writing-1.5`.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.