Coding a Transformer

Vemund Brynjulfsen

HiØ

ITI41820

Table of contents

1 Abstract

The transformer architecture is important in deep learning.

It handles sequence data through self-attention, which gives better results than the RNN and LSTM architecture, for language models. This is one of the reasons it is used as a building block in widely used NLP programs, like ChatGPT and BERT.

Coding a transformer will give a better understanding of how these programs work, as the algorithms will be shown in code.

In this paper, a transformer has been coded from scratch. Reading this paper will give a good understanding of the transformer architecture, both in theory and in code.

2 Introduction

This paper showcases the development of a transformer. This is an interesting project because it shows how a transformer works, by showcasing code and results for the transformer. A transformer is a deep machine-learning architecture developed by (Vaswani et al., 2017).

It handles sequence data and uses self-attention mechanisms to determine weights and spot long-range dependencies. The architecture is widely used. It is used as a building block in language models like ChatGPT and BERT.

Further theory and related work on transformers is discussed in **Theory and Related Work**. Tools and dataset is discussed in **Tuning and Experimental Results**, The results and work is discussed in **Discussion and Conclusion**.

3 Theory and Related work

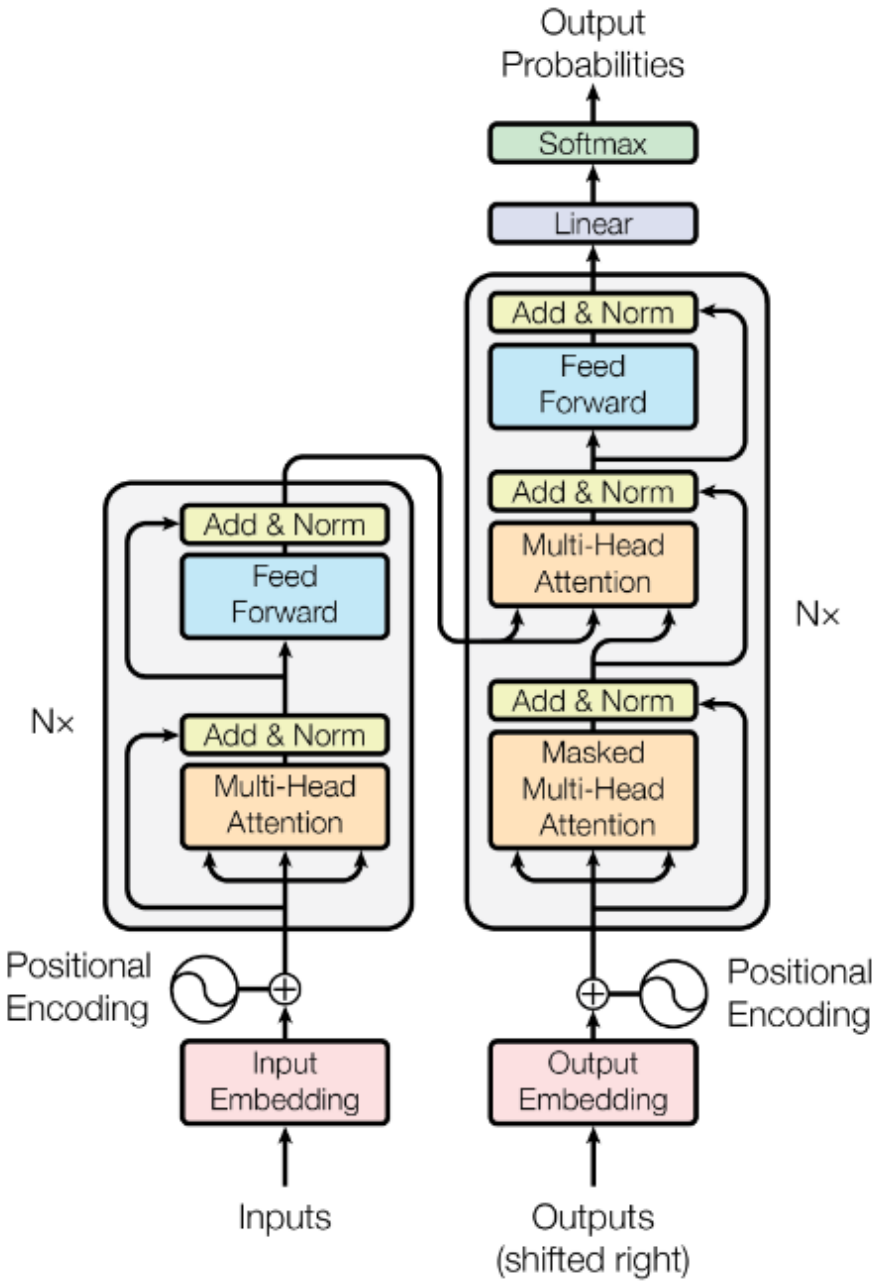

Figure 1: The Transformer model architecture (Vaswani et al., 2017)

3.1 Embedding

An embedding captures the meaning of a token. Let's say the transformer takes a sentence as an input, and produces another sentence as output. The sentence can, for example be, «This is an English sentence». The sentence is made into tokens. Tokens are pieces of broken-down test. For example, the sentence given as input might be broken up as <«this», «is», «a», «n», «eng», «lish», «sentence»>. Then we map each token to an input ID, which gives the token's position in the vocabulary. The input IDs are made into vectors called embeddings. An embedding has learnable parameters that, through training, will capture the meaning of a token.

The pipeline looks like this:

Text → Split into tokens → Map tokens to token-ids → Embedding

3.2 Positional encoding

Position encoding captures the position of a token in a sentence. Information about a token's position in a sentence, helps the model spot patterns in the sentence structure. Positional encoding is done by applying two formulas to the embeddings. One formula is used to even positions, and another to odd positions. The positional encoding isn't learnable, it's reused during training and inference.

Applied to all even positions.

$$PE(pos, 2i) = \sin\left(\frac{pos}{1000^{\frac{2i}{d_{model}}}}\right)$$

Applied to all odd positions.

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{1000^{\frac{2i}{d_{model}}}}\right)$$

3.3 Self-attention

Self-attention makes the model able to relate words to one another. It takes the positional encoded embeddings as input. This input will give information about a token's meaning and position. This is represented as a matrix. This matrix is copied into three matrices, for the query, key, and values. The query and key are multiplied, and then divided by the square root of the embedding length. Lastly, softmax is applied. This results in a score, which represents how intense the relationship is between the tokens.

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Then the score is multiplied with the values matrix. The result is an attention matrix. The multiplication of a Q and K matrix results in a matrix that tells us how strong the relation is between each word. Then the V matrix is multiplied, which makes the more important words contribute more to the attention matrix. The final attention matrix will have the same dimensions as the input embeddings. The attention matrix will provide information about each token's relation to the other tokens, in addition to the token's meanings and positions.

3.4 Multi-head attention

The input embeddings are duplicated into three matrices (Q, K, V), with the dimensions of the sequence length, and the embedding length (seq, d). Each matrix is multiplied with parameter matrices

$$(W^Q, W^K, W^V)$$

with the dimensions (d, d), which results in the matrices

$$(Q', K', V')$$

. With the dimensions (seq, d). These matrices are then split by d_k=d/h. We get smaller matrices for the number of heads specified, in the dimensions (seq, d_v=d_k). These matrices are then concatenated along the d_v dimension. This matrix is multiplied with a weight matrix (h*d_v, d), which results in the final output of the multi-head attention, with the dimensions (seq, d). In self-attention, the token's relation to the other tokens is captured. The

attention is calculated from (Q, K, V). In multi-head attention, the Q, K, and V is multiplied with weight matrices, then the resulting matrices are split into heads before calculating the attention. Each of these heads will capture a different part of a token. One might relate the token to a verb, another to a noun, and so on.

3.5 Masked multi-head attention

The output for a position should only depend on the words in the preceding positions. So we remove the influence of the tokens in front by applying masking. The mask replaces all values above the diagonal line in the matrix with negative infinity, before applying softmax. The output of the masked-multi-head attention will be used as the value input in cross-attention.

3.6 Cross self-attention

In self-attention, the key, query, and value came as single unit (the encoder). In cross self-attention, the key and query come from the encoder, while the values come from the decoder. The key and value from the encoder give the full meaning of the embeddings from the source sequence, which will tell the target sequence from the decoder, how much it should focus on each token in the source sequence. The query from the encoder represents the target sequence so far. The resultant matrix from cross self-attention will tell how much a token in the target sequence should focus on each token in the source sequence, or the similairity between each token in the source, and the target so far.

3.7 Feed Forward

The input to the feed-forward block is a tensor with the shape (batch size, seq, d). d_{ff} is the dimensionality of the feed-forward layer, and is normally larger than d. After the feed-forward block, the resulting tensor is of the shape (batch size, seq, d_{ff}). ReLU is used after the first linear layer, that introduces non-linearity into the model. After the ReLU is applied, the second linear layer transforms the tensor back into the shape (batch size, seq, d). The feedforward allows the model to learn non-linear transformations of the input. It is applied to each token in the sequence, and can be done so in parallell.

3.8 Training

The decoder-encoder architecture for a transformer takes two inputs. One input (source) goes to the encoder, and one input goes to the decoder (target). When the model makes predictions in this case, it will predict the next value from in the target, based on it's previous predictions for the target and the source.

This can be used if a model produces something based on a prompt. In that case, for each token that is predicted, the prediction will be based on previous target predictions, and the full source input.

An example is language translation. Let's say English is used as a source input, and Italian is used as a target input. To predict the next Italian word, it will generate this word based on the previous predicted Italian words, and the English sentence to translate from.

Let's say we have a batch of text examples. Each example has a text in English, and a translated sentence in Italian.

The texts are first made into tokens, and then made into token-ids. We also append some special tokens to the sentences. One special SOS (start of sentence) token is added at the beginning, and another EOS token (end of sentence) is added at the end of the text. These are for the model to learn when a sentence starts, and when it should end.

After the EOS token, PAD (padding) tokens are added. The model expects all examples to be of the same length, but it isn't nessecary that all example texts in the dataset is of the same length, or that a user always inputs a prompt of the same length. So PAD tokens are added to make all sentences have the same length.

A mask is used to cover up the padding tokens, so that the model ignores them. Not ignoring the padding tokens will bring a lot of noise into the dataset, as the model shouldn't update it's weights to learn where padding tokens are.

Such a batch will be a matrix of shape (batch size, sequence length). This batch is sent into the transformer. The embeddings contain word vectors that represents what a token means. For each example there is a two dimensional embedding of shape (sequence length, word vector length). There

is an embedding for each example, so the final datastructure is a tensor of shape (batch size, sequence length, word vector length).

Then attention is computed it is calculated how each word in the example attends to all the other words in the example. This can be done in parallell across the whole tensor.

The final linear projection layer produces logits, that holds a porbability distribution across the vocabulary.

The loss is calculated by taking the difference between the logits and the target. The targets contains for the vocabulary, which token is the correct prediction. Here cross-entropy is used to caulcate loss.

Then the loss can be used to update the weights.

3.9 Inference

Inference can be used when the model has been trained. Only input for the source is added. The target input is only a start token. The input will be processed the same way as during training. The final linear layer in the transformer will produce logits. These logits holds a porbability distribution across the vocabulary. This distribution is used to pick out the next token.

When the next token has been picked, it is added to the target input, and the we pass inn inputs to the transformer again to predict the next token. The source input is the same, but the target input now contains a SOS token, and the new token is predicted earlier.

Tokens are iterativly added to the target input, until the model predicts a EOS token.

3.10 Decoder-only

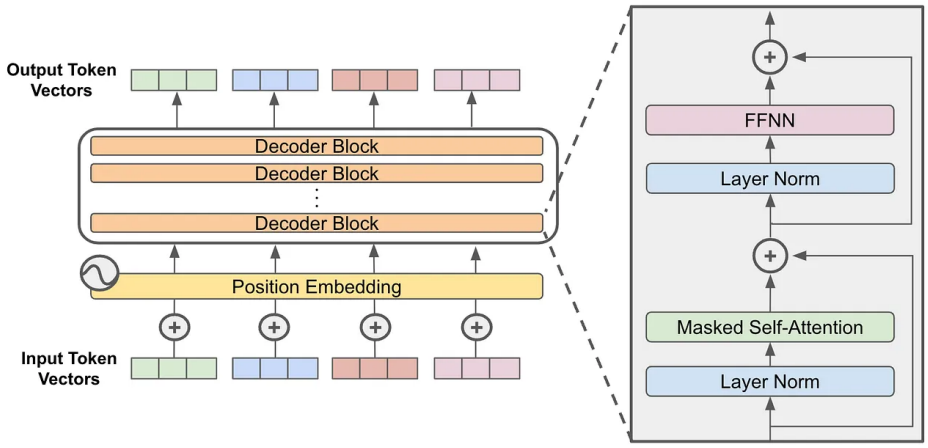

Figure 2: Structure of a decoder-only transformer model (Wolfe, 2024)

There is also a decoder-only model. This can be used to continue a sequence, that dosen't have a relating prompt to it. For example if the user has some text, and wants a model to generate the rest of that text, a decoder-only model can be used. Such a model will not take into account some source input, but only the previously predicted tokens.

With a deocer-only model, there is no cross attention, as a source input does not have to be attended to.

3.11 Parallellism

A transformer model holds millions of parameters. Because of this, a lot of data is needed to train the model. It can take a lot of computational power, expecially if it's very large. Because of this, it can be nessecary to utelize a large amount of GPUs.

For parallellism, a batch can be devided across several GPUs. So if it is needed to process a batch of size 128, the batch can be split into 8 smaller batches of size 16. Then each batch of size 16 is forward passed with each GPU. The output from each model is averaged together into a single output, that can be used to update the parameters.

4 Implementation

4.1 Framework

The transformer was coded in JAX. JAX is a functional programming framework developed by DeepMind. It is used for machine learning among other things. It provides high-performance numerical computing like NumPy with many useful functionalities for developing machine learning models.

Pytorch is a framework that provides higher-level abstractions for building a neural network. For example the architecture of a whole layer can be given in a single line.

JAX which is a lower-level framework. The programmer has to put in more work to create a neural network. The programmer has to combine arrays into matrices and tensors, and do various calculations like matrix multiplication, apply activation functions, dropout etc.

Using JAX makes it more challenging to create a transformer from scratch, But the inner workings of a transformer can be shown more clearly in code.

4.2 Metrics

This section explains different metrics used for evaluating performance.

4.2.1 Loss for opus

Loss is attained by calculating the difference between logits and a target. Here cross entropy is used. Logits is a probability distribution across the vocabulary. The target contains the ground truth, the correct next token.

4.2.2 Perplexity

PPL measures the model uncertanty when measuring the next token. It is calculated as the exponential average of the negative log likelyhood of the predicted probabilities.

$$\text{PPL} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i \mid w_1, w_2, \ldots, w_{i-1})\right)$$

4.2.3 Loss for synthetic datasets

The synthetic datasets are in the form of embeddings. Since they do not contain probabilities over vocabulary, and because the label is in the form of an embedding, the loss needs to be calculated in another way.

Mean Squared Error (MSE) is used instead to compare the embedding output to a target embedding.

It's not possible to calculate perplexity for these, as logits are needed. This is discussed further when describing the dataset in the section below.

4.3 Resources

The code was run with GPUs avalible at the HiØ GPU cluster. The cluster has several Quadro RTX 8000 GPUs avalible.

4.4 Datasets

The transformer was run on three datasets.

4.4.1 Helsinki-NLP/opus books

The opus dataset (Tiedemann, 2012) is designed for neural translation models. The dataset has examples across several languages, with the goal of translation from one language to another.

The transformer model was trained on a opus dataset for translating from English to Italian. This dataset consists of 32.332 examples. The code for the tokenizer is provided by (Sayed, 2024b). The tokenizer is trained on the text. The longest sequence in the dataset consists of ~350 tokens, and the dataset's vocabulary size is a ~22.000.

In this case, a decoder-encoder transformer was implemented to do translation tasks. The encoder will recieve input for English, while the decoder end will recieve input for Italian.

4.4.2 Synthetic datasets

The model was also trained on two synthetic datasets. A synthetic dataset is a dataset that is artifically created, for example by an algorithm, instead

of real-world examples. Both datasets have 10.000 examples

The datasets have not previously been tested on transformer models. These datasets could not be utelized like the Opus translation dataset. In the Opus translation dataset, the inputs come in the form of text. Then this text has to be preprocessed into embeddings.

But in the synthetic datasets, the data is already in the form of embeddings, This applies to the source, target, and label. Because of this, some changes were made to the model architecture.

When using these datasets, the embedding layer and final projection layer is removed. Since label is in the form of an embedding, the input data and the label needed to be of the same shape when calculating loss. So the embedding dimensions should be kept.

When calculating loss for the opus translation dataset, cross entropy is used.

But for the synthetic datasets, the mean squared error (MSE) between the precited embedding output and the label embedding was used instead.

PPL was not calculated for the synthetic datasets, as logits couldn't be made. PPL is used to measure how well a model predicts the next word in a sequence. Here logits come in, as they represent a probability distribution over the vocabulary.

4.4.3 Dictionary Translation

The dataset simulates translation from one language to another. It's expected that this dataset should be easy for a transformer to solve. The word vector length is 14, and the vocabulary size is 2. The sequence length is 128.

4.4.4 Pattern Search

This dataset is to see if a transformer made for generating text, can also process images. It's expected that this dataset should be almost impossible for transformers to process. An example has two images, one for input and one for output. The first image has a pattern that occurs in the second image. The goal of the transformer is to erase the pattern from the second input. The vocabulary size is 2 (0 for white, 1 for black), and the word vector length is also 2. The sequence length is 90.

4.4.5 Wikitext-103

The (Merity et al., 2016) wikitext-103 dataset consists of different articles from wikipedia. It has ~2 million examples. It has a far larger vocabulary than opus. The tokenizer GPT-2 was used with this dataset, which has a vocabulary size of ~50.000 tokens. In addition, it was nessecary to implement parallellism across several GPUs. This was achieved with JAX's pmap framework, where a batch can be split across several GPUs.

For the other datasets, a encoder-decoder architecture was implemented. The wikitext-103 dataset has only one input, instead of two for a source and a target. Because of this, the architecture was for this dataset changed to a decoder-only architecture. The model can be used to generate text from an input. So if part of a text is given, the model will try to predict the continuation of that text.

However, some problems related to GPU utilization was met when trying to train the model on the dataset. So results for this dataset are not ready. This is for future work.

5 Tuning and Experimental Results

5.1 Tuning

Tuning was done by ranging the learning rate at different spans, as the learning rate is the most important variable.

For Opus, the other hyperparameters, such as sequence length, embedding size, stacks, were the same as (Sayed, 2024a)'s hypterparameter values. (Sayed, 2024a) used a learning rate of 10**-4.

5.2 Helsinki-NLP/opus books

(Sayed, 2024a) has implemented a transformer from scratch using pytorch based on (Vaswani et al., 2017).

Hyperparameters: These hyperparameters are gotten from (Sayed, 2024b).

Hyperparameter	Value
Batch size	8
Epochs	20
Learning Rate	X (Varied)
Seq length	350
d_{model}	512
dimension feed forward	2048
dropout rate	0.1
amount of blocks	6

This article varies the learning rate. The learning rate is varied between 10**-2 to 10**-5

With 6 stacks there are 75.521.983 learnable parameters.

The learning rate og 10**-4 gave the best/lowest PPL of 11. A learning rate of 10**-3 also gave a good PPL result of 22.

With a learning rate is 10**-4, the PPL end up being 102, which means the model performed significantly worse.

When the learning rate was 10**-2, the PPL ended up being 1299. The model managed to learn, since the PPL went down from 2407 in the second

epoch. But the model loss seems to converge at the same rate, along with the PPL, which indicates that the model struggles to learn.

However, the models seem to struggle with new data. On the test set, the PPL is regularly far higher. (Sayed, 2024a) had a bit better performance on the training data, but also had more overfitting, in that PPL on the training set was lower, and PPL on the test set was higher.

A learning rate of 10**-5 gave the least overfitting, but then the PPL ends up at ~100.

The reason the models struggle to learn new data might be because the dataset contains too little data for the model to properly cover all potential cases.

5.3 Dictionary Translation

PPL was not used for the dictionary translation dataset. This is because logits are used to calculate PPL, but the inputs and label are all in the shape of embeddings. Only loss will be given as a metric.

The learning rate was varied between 10**-1 to 10**-4.

Hyperparameters:

Hypterparameter	Value
Batch size	8
Epochs	20
Learning rate	X (varied)
Sequence length	128
embedding size	14
feed forward layer size	2048
Number of heads	7
Stacks	1

5.4 Pattern Search

PPL was not used for the Pattern search dataset. This is because logits are used to calculate PPL, but the inputs and label are all in the shape of embeddings. Only loss will be given as a metric.

The learning rate was varied between 10**-1 to 10**-4.

Hyperparameters:

Hyperparameter	Value
Batch size	8
Epochs	20
Learning rate	X (Varied)
Sequence length	90
Embedding size	2
Feed forward size	2048
Heads	2
Dropout rate	0.1
Stacks	1

With a single stack, the model has 124.778 parameters.

6 Dicussion

The results show that the model architecture works fine and gives good results. This article's JAX implementation of the transformer gives comparable results to (Sayed, 2024a).

The transformer did not seem to be able to learn the synthetic datasets, as the loss does not decrease much at all.

On top of implementing a standard decoder-encoder model, some changes to the architecture needed to be implemented.

When utilizing the synthetic datasets, the architecture was changed to remove the embedding and final projection layer.

To accommodate for the wikitext103 dataset, a decoder-only model needed to be implemented, by removing the encoder layers. And a batch needed to be split up to be able to process a batch in parallell across multiple GPUs. But this is somewhat still a work in progress.

7 Conclusion

The performance on the opus dataset shows that the model works. It is intresting to see how the model performed on the opus translation datasets. To get better results, parallelization og the architecture needs to be implemented, to accommodate for larger datasets.

8 Future improvments

The model can be run on larger datasets. For example the wikitext-103 can be used. Currently the model can only utelize GPUs that are on the same cluster, but if the model is made to work across several nodes, then more GPUs can be utelized. Then the model can be run on larger datasets, like wikitext-103.

9 Appendix

9.1 Results

9.1.1 Helsinki-NLP/opus books

This is the result of (Sayed, 2024a)'s model:

lr = 10**-4

Epoch	avg loss	avg PPL train
Epoch 0	6.290	416.812
Epoch 1	5.602	142.980
Epoch 2	5.247	94.994
Epoch 3	4.970	68.958
Epoch 4	4.718	51.693
Epoch 5	4.483	39.500
Epoch 6	4.263	30.610
Epoch 7	4.049	23.842
Epoch 8	3.848	18.874
Epoch 9	3.652	14.991
Epoch 10	3.465	12.028
Epoch 11	3.290	9.757
Epoch 12	3.118	7.957
Epoch 13	2.957	6.570
Epoch 14	2.809	5.485
Epoch 15	2.676	4.667
Epoch 16	2.552	4.011
Epoch 17	2.444	3.517
Epoch 18	2.351	3.136
Epoch 19	2.267	2.834

Test result:

Loss: 6.686024188995361

PPL: 753.8371381110885

These are the results:

lr = 10**-2

Epoch	avg loss	avg PPL train
Epoch 0	0.655	89790
Epoch 1	0.637	2407
Epoch 2	0.649	2336
Epoch 3	0.639	2068
Epoch 4	0.592	1468
Epoch 5	0.624	1797
Epoch 6	0.600	1519
Epoch 7	0.598	1511
Epoch 8	0.584	1379
Epoch 9	0.562	1125
Epoch 10	0.562	1105
Epoch 11	0.559	1159
Epoch 12	0.633	2159
Epoch 13	0.597	1559
Epoch 14	0.580	1271
Epoch 15	0.596	1484
Epoch 16	0.584	1313
Epoch 17	0.580	1267
Epoch 18	0.590	1353
Epoch 19	0.585	1299

Test result:

Loss: 0.586

PPL: 1785

lr = 10**-3

Epoch	avg loss	avg PPL
Epoch 0	0.443	1069
Epoch 1	0.397	154
Epoch 2	0.381	125
Epoch 3	0.368	107
Epoch 4	0.358	93
Epoch 5	0.348	82
Epoch 6	0.340	74
Epoch 7	0.331	66
Epoch 8	0.323	60
Epoch 9	0.315	54
Epoch 10	0.308	49
Epoch 11	0.300	44
Epoch 12	0.293	41
Epoch 13	0.285	37
Epoch 14	0.278	34
Epoch 15	0.271	31
Epoch 16	0.264	28
Epoch 17	0.258	26
Epoch 18	0.251	24
Epoch 19	0.246	22

Test result:

Loss: 0.444

PPL: 578

lr = 10**-4

Epoch	avg loss	avg PPL
Epoch 0	0.486	2631
Epoch 1	0.413	192
Epoch 2	0.386	136
Epoch 3	0.367	106
Epoch 4	0.351	86
Epoch 5	0.336	71
Epoch 6	0.323	60
Epoch 7	0.310	51
Epoch 8	0.298	43
Epoch 9	0.286	37
Epoch 10	0.275	32
Epoch 11	0.264	28
Epoch 12	0.254	24
Epoch 13	0.243	21
Epoch 14	0.234	19
Epoch 15	0.224	17
Epoch 16	0.216	15
Epoch 17	0.207	13
Epoch 18	0.199	12
Epoch 19	0.192	11

Test result:

Loss: 0.427

PPL: 342

lr = 10**-5

Epoch	avg loss	avg PPL
Epoch 0	0.627	4003
Epoch 1	0.497	547
Epoch 2	0.474	413
Epoch 3	0.457	331
Epoch 4	0.444	280
Epoch 5	0.433	245
Epoch 6	0.425	219
Epoch 7	0.417	200
Epoch 8	0.411	184
Epoch 9	0.405	171
Epoch 10	0.400	160
Epoch 11	0.395	150
Epoch 12	0.391	142
Epoch 13	0.386	134
Epoch 14	0.382	128
Epoch 15	0.379	122
Epoch 16	0.375	116
Epoch 17	0.371	111
Epoch 18	0.368	106
Epoch 19	0.365	102

Test result:

Loss: 0.395

PPL: 186

9.1.2 Dictionary Translation

lr: 10**-1

Epoch	avg loss
Epoch 0	2.0002
Epoch 1	1.9991
Epoch 2	1.9986
Epoch 3	1.9985
Epoch 4	1.9983
Epoch 5	1.9986
Epoch 6	1.9983
Epoch 7	1.9982
Epoch 8	1.9982
Epoch 9	1.9982
Epoch 10	1.9981
Epoch 11	1.9985
Epoch 12	1.9980
Epoch 13	1.9984
Epoch 14	1.9983
Epoch 15	1.9984
Epoch 16	1.9982
Epoch 17	1.9980
Epoch 18	1.9980
Epoch 19	1.9980

Test data:

Loss: 1.9983

lr: 10**-2

Epoch	avg loss	val avg loss
Epoch 0	1.9986	1.9989
Epoch 1	1.9989	2.0002
Epoch 2	1.9982	1.9997
Epoch 3	1.9986	2.0007
Epoch 4	1.9984	2.0001
Epoch 5	1.9986	2.0004
Epoch 6	1.9984	2.0004
Epoch 7	1.9981	2.0003
Epoch 8	1.9979	2.0002
Epoch 9	1.9979	2.0000
Epoch 10	1.9978	2.0005
Epoch 11	1.9974	2.0004
Epoch 12	1.9977	2.0001
Epoch 13	1.9974	2.0000
Epoch 14	1.9973	2.0000
Epoch 15	1.9974	1.9999
Epoch 16	1.9972	1.9998
Epoch 17	1.9973	1.9997
Epoch 18	1.9973	1.9998
Epoch 19	1.9979	1.9999

Test data:

Loss: 1.9982

lr: 10**-3

Epoch	avg loss	val avg loss
Epoch 0	1.9616	1.9594
Epoch 1	1.9241	1.9248
Epoch 2	1.9023	1.9039
Epoch 3	1.8914	1.8939
Epoch 4	1.8856	1.8881
Epoch 5	1.8818	1.8845
Epoch 6	1.8757	1.8783
Epoch 7	1.8728	1.8753
Epoch 8	1.8707	1.8733
Epoch 9	1.8686	1.8711
Epoch 10	1.8661	1.8693
Epoch 11	1.8635	1.8664
Epoch 12	1.8618	1.8648
Epoch 13	1.8604	1.8636
Epoch 14	1.8592	1.8632
Epoch 15	1.8591	1.8626
Epoch 16	1.8582	1.8628
Epoch 17	1.8577	1.8621
Epoch 18	1.8576	1.8621
Epoch 19	1.8573	1.8617

Test data:

Loss: 1.8635

lr: 10**-4

Epoch	avg loss	val avg loss
Epoch 0	1.9801	1.9795
Epoch 1	1.9593	1.9599
Epoch 2	1.9481	1.9494
Epoch 3	1.9401	1.9423
Epoch 4	1.9346	1.9369
Epoch 5	1.9320	1.9345
Epoch 6	1.9297	1.9326
Epoch 7	1.9279	1.9311
Epoch 8	1.9260	1.9299
Epoch 9	1.9240	1.9282
Epoch 10	1.9218	1.9257
Epoch 11	1.9193	1.9238
Epoch 12	1.9173	1.9219
Epoch 13	1.9155	1.9198
Epoch 14	1.9134	1.9182
Epoch 15	1.9122	1.9170
Epoch 16	1.9107	1.9155
Epoch 17	1.9094	1.9144
Epoch 18	1.9084	1.9139
Epoch 19	1.9073	1.9128

Test data:

Loss: 1.9140

9.1.3 Pattern Search

lr: 10**-1

Epoch	avg loss	val avg loss
Epoch 0	1.0749	1.0745
Epoch 1	1.0749	1.0745
Epoch 2	1.0749	1.0745
Epoch 3	1.0749	1.0745
Epoch 4	1.0749	1.0745
Epoch 5	1.0749	1.0745
Epoch 6	1.0749	1.0745
Epoch 7	1.0749	1.0745
Epoch 8	1.0749	1.0745
Epoch 9	1.0749	1.0745
Epoch 10	1.0749	1.0745
Epoch 11	1.0749	1.0745
Epoch 12	1.0749	1.0745
Epoch 13	1.0749	1.0745
Epoch 14	1.0749	1.0745
Epoch 15	1.0749	1.0745
Epoch 16	1.0749	1.0745
Epoch 17	1.0749	1.0745
Epoch 18	1.0749	1.0745
Epoch 19	1.0749	1.0745

Test data:

Loss: 1.0746

lr: 10**-2

Epoch	avg loss	val avg loss
Epoch 0	1.0750	1.0745
Epoch 1	1.0749	1.0745
Epoch 2	1.0749	1.0745
Epoch 3	1.0749	1.0745
Epoch 4	1.0749	1.0745
Epoch 5	1.0749	1.0745
Epoch 6	1.0749	1.0745
Epoch 7	1.0749	1.0745
Epoch 8	1.0749	1.0745
Epoch 9	1.0749	1.0745
Epoch 10	1.0749	1.0745
Epoch 11	1.0749	1.0745
Epoch 12	1.0749	1.0745
Epoch 13	1.0749	1.0745
Epoch 14	1.0749	1.0745
Epoch 15	1.0749	1.0745
Epoch 16	1.0749	1.0745
Epoch 17	1.0749	1.0745
Epoch 18	1.0749	1.0745
Epoch 19	1.0749	1.0745

Test data:

Loss: 1.0746

lr: 10**-3

Epoch	avg loss	val avg loss
Epoch 0	1.0620	1.0632
Epoch 1	1.0599	1.0613
Epoch 2	1.0610	1.0624
Epoch 3	1.0579	1.0591
Epoch 4	1.0562	1.0574
Epoch 5	1.0510	1.0523
Epoch 6	1.0511	1.0525
Epoch 7	1.0497	1.0513
Epoch 8	1.0481	1.0497
Epoch 9	1.0479	1.0495
Epoch 10	1.0472	1.0487
Epoch 11	1.0434	1.0446
Epoch 12	1.1308	1.1327
Epoch 13	1.1752	1.1765
Epoch 14	1.1752	1.1765
Epoch 15	1.1752	1.1765
Epoch 16	1.1752	1.1765
Epoch 17	1.1752	1.1765
Epoch 18	1.1752	1.1765
Epoch 19	1.1752	1.1765

Test data: Loss: 1.1759

lr: 10**-4

Epoch	avg loss	val avg loss
Epoch 0	1.1752	1.1764
Epoch 1	1.1751	1.1764
Epoch 2	1.1752	1.1765
Epoch 3	1.1752	1.1765
Epoch 4	1.1752	1.1765
Epoch 5	1.1785	1.1792
Epoch 6	1.1892	1.1888
Epoch 7	1.1889	1.1885
Epoch 8	1.1896	1.1892
Epoch 9	1.1898	1.1894
Epoch 10	1.1896	1.1892
Epoch 11	1.1884	1.1881
Epoch 12	1.1877	1.1873
Epoch 13	1.1878	1.1874
Epoch 14	1.1884	1.1879
Epoch 15	1.1883	1.1878
Epoch 16	1.1882	1.1879
Epoch 17	1.1882	1.1878
Epoch 18	1.1890	1.1885
Epoch 19	1.1902	1.1898

Test data:

Loss: 1.1911

9.2 Module overview

The different files in the projects are used to initilize and train a transformer model on different datasets.

All the modules in the projects define different algorithms, that combined defines and algorithm for creating and using a transformer.

9.2.1 train

To run the project, these files should be run. For each dataset, there is one train file for each hyperparameter variation. So there are 4 train files for each dataset (opus and the synthetic datasets).

I have also included some of the code from (Sayed, 2024b). His project can also be run by running the train.py file.

The unfinished project for the wikitext-103 is included. For this project, the train file contains code for splitting the data across several GPUs JAX's pmap framework.

9.2.2 config

The hyperparameters are stated in the config files. For each train file, there is a config file.

9.2.3 init forward

The init forward files are for initilizing the transformer model, and forward passing data through the transformer model. In the train files, the model parameters are in a variable "params".

The synthetic init forward files differ from the init forward file for the opus dataset, in that the init forward files for the synthetic datasets have no embedding and final projection.

9.2.4 transformer

This file defines the initilization of different transformer layers, and the forward pass for the layers.

The module is used in the init forward files to construct a full model by stacking the layers on top of eachother.

9.2.5 data

This file is used to initilize and preprocess the opus dataset. Much of the code is gotten directly from (Sayed, 2024b)

9.2.6 load$_r$

These files are in the projects for the synthetic datasets. They are responsible for initilizing and preprocessing the synthetic datasets.

9.2.7 utils

This file holds utility functionality. There is for example functions for saving model weights, loading model weights, preprocessing data, etc.

9.2.8 metrics

Holds the function for calculating perplexity

9.2.9 dependencies.txt

This file holds all the nessecary dependencies for running the projects.

9.3 Data structures and algorithms

9.3.1 Params

All the parameters for a model is in the variable "params" in the train files.

9.3.2 Initilization

In the transformer.py file are algorithms for initilizing the layers of the transformer. The weights for a layer is initilized in jax arrays, and saved in a dictionary.

The init forward files use the abstractions in transformer.py, to initilize the layers together, by tacking the layers, and combining all the parameters in a final dictionary for the model.

9.3.3 Forward pass

The algorithm for forward pass is stored in the transformer.py file, and the init forward files.

The transformer.py file defines algorithms for passing data through each layer, by doing operations such as matrix multiplication between the input data and the parameters for the layer.

In the init forward files the layers are stacked on top of each other. Here is the algorithm for passing data trough for each layer, and taking the data from some layer and using it as input to the next layer.

Refrences

Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2016). Pointer sentinel mixture models.

Sayed, E. (2024a, June). Building a transformer from scratch: A step-by-step guide. https://medium.com/@sayedebad.777/building-a-transformer-from-scratch-a-step-by-step-guide-a3df0aeb7c9a

Sayed, E. (2024b, June). Training a transformer model from scratch [Published on Medium, 11 June 2024].

Tiedemann, J. (2012, May). Parallel data, tools and interfaces in OPUS. In N. Calzolari, K. Choukri, T. Declerck, M. U. Doğan, B. Maegaard, J. Mariani, A. Moreno, J. Odijk, & S. Piperidis (Eds.), *Proceedings of the eighth international conference on language resources and evaluation (LREC'12)* (pp. 2214–2218). European Language Resources Association (ELRA). http://www.lrec-conf.org/proceedings/lrec2012/pdf/463_Paper.pdf

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. https://arxiv.org/abs/1706.03762

Wolfe, C. R. (2024, March). Decoder-only transformers explained: The engine behind llms [Figure reproduced from the article. Published on Mar 04, 2024].