Using Transformer Architecture to Build a Language Model That Can Generate Poetry in Macedonian

Candidate:
Marko Petrov

Mentor:
Murat Akbiyik

Yahya Kemal College
Skopje, Macedonia

Contents

List of Figures

Abstract

This research paper aims to implement and optimize the transformer architecture to design a language model specializing in the creation of poetry in the Macedonian language. The significance of preserving, protecting, and promoting the Macedonian culture through computational methods is the primary motivation factor of this study.

Regarding the methodology, poetry books from influential Macedonian writers will be collected to train the language model for text creation by employing the essence of their writing styles. The performance of the system will be evaluated both quantitatively through a mathematical loss function as well as qualitatively. In the end, existing models available on the internet will undergo the same evaluation given the same input prompt to compare the efficiency of the models.

In summary, this paper strengthens the link between natural language processing, computational methods, and the preservation of a nation's culture and history.

Chapter 1

Introduction

This chapter will focus on the current state of artificial intelligence, the background of language models, and the importance of text generation in Macedonian. The history and problems encountered with other NLP architectures will be explained to underline the noteworthiness of a new and better approach.

1.1 Background on Language Models

A language model is nothing more than a computer program that predicts what is most likely to be the next word in a given sequence of words. Large amounts of data are provided to the computer to find patterns between words, their meaning, and use cases. To some extent, this resembles how people learn to talk and articulate their points. For example, humans process a lot of text daily, and whenever the word "pizza" is encountered the context is usually about food, recipes, or similar. The odds are you will never hear the words "pizza" and "nuclear bomb" in the same sentence. Exposure to books, other people's opinions, articles, and the endless inflow of information in today's world enhances our ability to search quickly for the right word or phrase in our brain to convey a certain message. That is why different people have different styles of communicating and often reuse similar word patterns that they have encountered the most in the past. What was mentioned above is the fundamental difference in logic between traditional programming and machine learning. In traditional programming, we write an algorithm, provide input, and receive output, while in machine learning we provide the system input data and correct output, letting it discover the rules itself and apply them in the future.

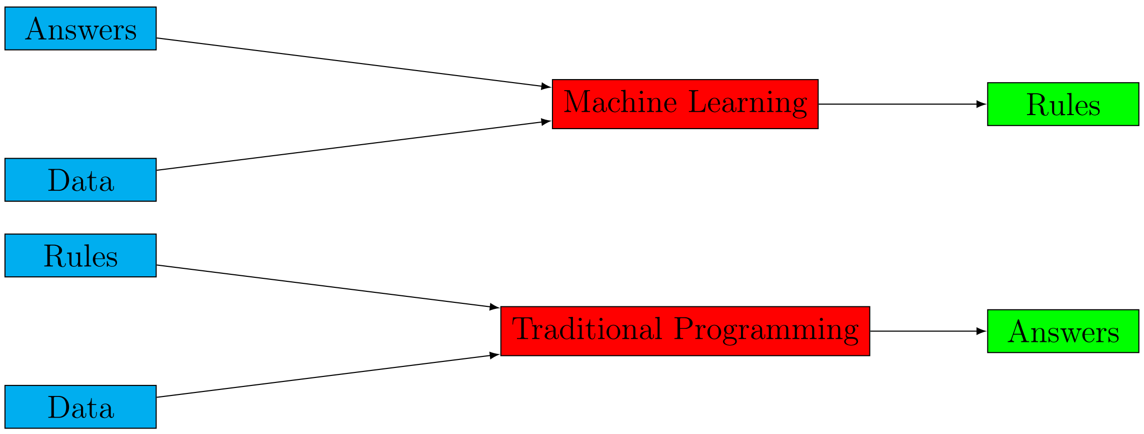


Figure 1.1: Traditional Programming Versus Machine Learning

1.1.1 Trivial Approach in Language Models

The simplest way to look at this prediction problem is to calculate the conditional probability that a certain word appears after every other word (given data to train on). Denote probability of word w_1 occuring given that word w_2 has occured as $P(w_1 \mid w_2)$.

$$P(w_1 \mid w_2) = \frac{P(w_1 \cap w_2)}{P(w_2)}$$

This works perfectly, but the average sentence has more than two words, which makes this approach absurd as context will be lost shortly because we only consider two words at a time. If this method is expanded linearly for some n words taken into context, the space complexity of this solution will be $\mathcal{O}(c^n)$, where c is the number of words in the dictionary (Drost, 2023). This exponential complexity quickly becomes infeasible for any modern computer. A more suitable solution would be with the use of neural networks, but these approaches will be discussed in the later chapters.

1.2 Why is Poetry Generation Important?

Poetry generation enhances the link between human language and computers. It employs linguistically and artistically the traditional writing styles, pushing the boundary beyond and preserving the culture of a nation. If used correctly, poetry generation could potentially play a pivotal role in early education by exposing students to an endless supply of ideas and inspirations. Already existing models like Chat GPT are of little help when it comes to the Macedonian culture as the system has been trained on a noticeably smaller dataset in this language, leaving room for the inappropriate use of words from similar languages like Serbian or Bulgarian. Available models are vastly trained in these languages, which is the reason why they make mistakes when it comes to the Macedonian language. The main goal of this paper is to build a solution to this problem and explain the transformer architecture.

Chapter 2

Deep Neural Networks

This chapter aims to demystify and deeply dive into the core idea of neural networks. The structure, front-propagation, back-propagation, and gradient descent are subtopics that will be discussed in this chapter.

2.1 Structure of Deep Neural Networks

Looking back to the problem we discussed in the previous chapter, obviously, a new approach that captures general patterns and learns from data is needed, not just naively calculating the probability of all possible word permutations, as that is memory and time-expensive. Neural networks are the most promising solution to this problem. What makes this solution particularly efficient is the flexibility and optimization methods. A neural network is a group of artificial neurons separated into different layers that receive information from all the neurons of the previous layer and send information to all the neurons of the next layer. The input layer is where the initial data from the user is inserted and the output layer is the nodes where the final prediction is stored. For example, if the neural network is given an image and it has to determine whether the image contains a cat or not, the output layer will contain only one neuron that should have a value greater than a certain threshold if the image contains a cat. More complex tasks will involve multiple neurons in the output layer for more specific cases and many hidden layers to enhance the precision of the model. The hidden layers are the part of the neural network that improves with training to make a better prediction. Below, we show the feed-forward general structure of a neural network. It's important to note that there are different types of neural networks which will be mentioned later.

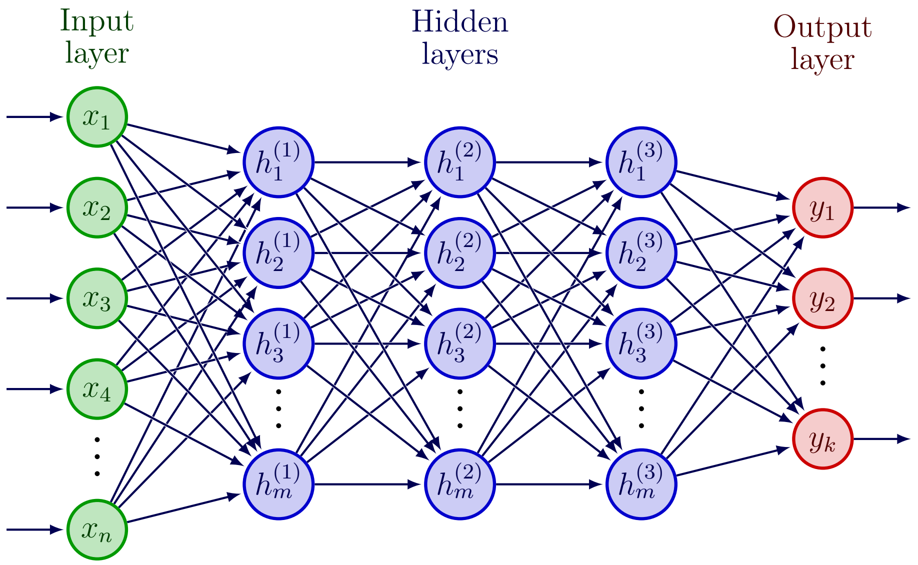


Figure 2.1: Neural Network Structure

2.2 How do artificial neurons "learn"?

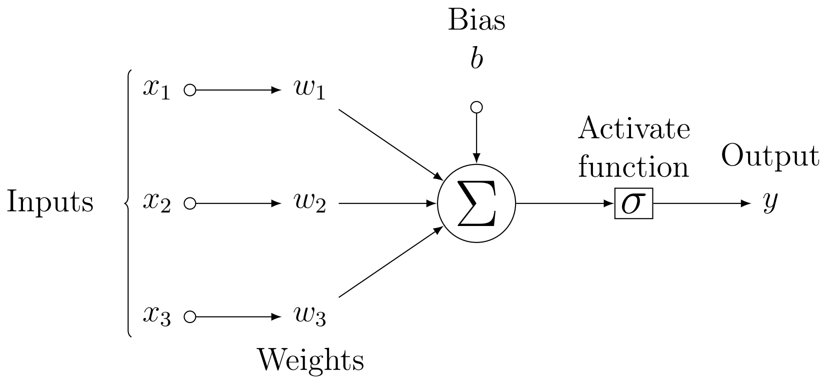


Figure 2.2: Artificial Neuron Structure

The artificial neuron is the core component of the neural network and is involved in the learning process. The figure above shows three input values $x_{1,2,3}$ pointing towards the artificial neuron from all the three neurons of the previous layer. Their value is multiplied with a certain weight $w_{1,2,3}$ for each input source respectively. You can think of weights as the number attached to the edge between any two nodes. The neuron takes the sum of all values and adds a predetermined coefficient b used to offset the result. This value is then plugged into an activation function σ that ensures the neural network's non-linearity as linear regressions are inaccurate in complex representations. Formally, the value y_n going through neuron n is the following:

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \tanh(z)$$

$$\sum_{k \in layer(n)-1} w_k \cdot x_k + b \longrightarrow \sigma \longrightarrow y_n$$

The non-linearity introduced with the activation function allows the neural network to approximate any continous function. This is true because of the universal theorem of approximation which states that a neural network with one layer and enough artificial neurons can approximate any continous function. Even better, in 2023, it was proven that a neural network with three layers can approximate any function both continous and not continous. The intuition of the proof behind this is the fact that we can stack "building" functions from many neurons to approximate the function with great precision.

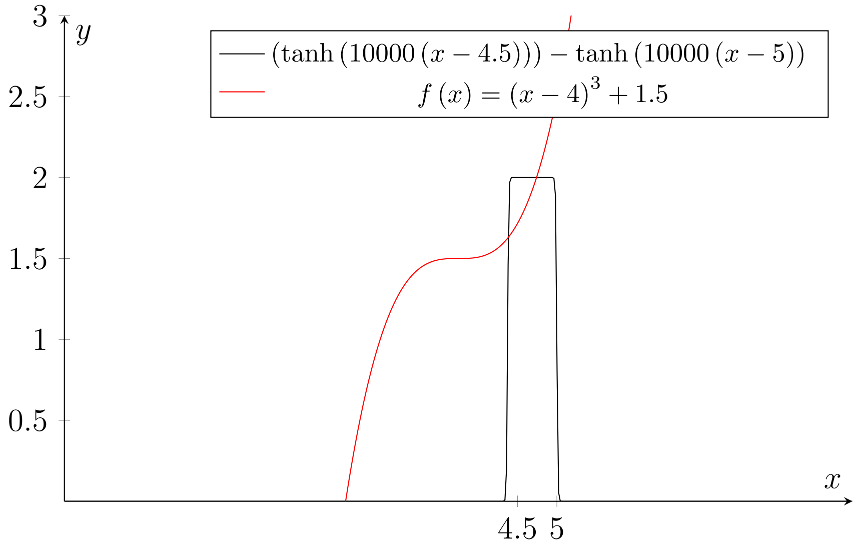


Figure 2.3: Idea of The Universal Approximation Theorem

Notice how we were able to approximate a function using only two non-linear functions and adding them together. It turns out, if we do this infinitely many times, in theory, we can appxoimate any continous function. Of course, in practice, the width of the "buildings" will be much smaller to make better approximation. However, now the interesting part is to figure out how to adjust the parameters of the neurons so we can model any function.

Initially, the weights are filled with a small random value before we start training the neural network. To start training the neural network, input is inserted and prediction is obtained. The prediction is then compared to the correct output via a loss function like the one below (cross-entropy loss function). Note that θ represents the parameters (weights and coefficients) in the neural network, hence $\mathcal{L}(\theta)$ is the loss function for certain parameters. The correct output is labeled as $\hat{y}$.

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot \log(\hat{y_i}) + (1 - y_i) \cdot \log(1 - \hat{y_i})]$$

Initially, when all the weights are randomly chosen, the loss function will show a very bad result. However, this is the critical point that makes neural networks improve. Every neuron contributes to the loss function through a chain of other neurons. This allows us to express $\mathcal{L}$ as a function of the values of the weights between the neurons and adjust the weights and biases to achieve better predictions in the future. This process is known as gradient descent. It works by taking the partial derivative of the loss function with respect to the parameters(weights) in the neural network and pushing the original weights against these values to lower the loss function. After the weights are adjusted, a new data is given to the neural network and the process is repeated until a satisfactory result is achieved. Formally, for every weight w_i the new value of this weight after evaluating the loss function will be:

$$w_i \coloneqq w_i - \alpha\frac{\partial\mathcal{L}(\theta)}{\partial w_i}$$

The coefficient α represents the learning rate of the neural network. If this parameter is higher, there will be less training data needed but inaccurate results, while lower value would mean that more training data is needed and higher precision will be achieved. This entire process of evaluating loss function, calculating partial derivatives, and updating the weights is known as back-propagation. Here is an example of how we derive the directions(derivatives) for simple linear regression models with mean-square loss function.

$$y = mx + b$$

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y} - y)^2 = \frac{1}{n} \sum_{i=1}^{n} (\hat{y} - mx_i - b)^2$$

$$\frac{\partial \mathcal{L}}{\partial m} = \frac{1}{n} \sum_{i=1}^{n} 2(\hat{y} - mx_i - b)(-x_i) = \frac{-2}{n} \sum_{i=1}^{n} (\hat{y} - mx_i - b)(x_i)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{n} \sum_{i=1}^{n} 2(\hat{y} - mx_i - b)(-1) = \frac{-2}{n} \sum_{i=1}^{n} (\hat{y} - mx_i - b)$$

$$m \coloneqq m - \alpha \frac{-2}{n} \sum_{i=1}^{n} (\hat{y} - mx_i - b)(x_i)$$

$$b \coloneqq b - \alpha \frac{-2}{n} \sum_{i=1}^{n} (\hat{y} - mx_i - b)$$

In summary, a neural network is a complex chain of artificial neurons that are given training data and adjust their values according to whether the prediction is accurate or not. Eventually, after training the model on a large dataset, the neural network will make accurate predictions.

Chapter 3

Generative Pre-trained Transformer (GPT)

This chapter will focus on a very specific type of neural network called transformer. It is the core behind famous models like Chat GPT, Google Bard, or Meta Llama. The structure of the transformer will be explained in this chapter. The full implementation will not be pasted in this chapter as it is thousands of lines of code, but the basic and neat implementation is taken from Andrej Karpathy. The code is further optimized and adjusted for our own problem.

3.1 Why is transformer better than other models?

Take a look at the following example. Suppose we have a model that needs to write a short story. We will give the model a starting sentence that needs to be extended into a story. For example, let the starting sentence be "Three students entered the classroom." Chat GPT 3.5 wrote the following response to the input:
"Three students entered the classroom, their steps echoing softly against the tiled floor. Each carried a distinct aura, reflective of their individual personalities..."
The way in which language models work is that as a word is generated, it is referenced to previous words in the sequence that are related to its context. For example, the word "personalities" is strongly related to the word "students" in the context. The language models assign a value to these relationships, but solutions like recurrent neural networks have a very short reference window because gradients are slowly lost because of the depth. What this means is that the model cannot access words that were used a while ago, therefore it will start to lose context after some time. On the other hand, transformers have a theoretically infinite reference window because of their attention mechanism that eliminates this problem.

3.2 Architecture

Below is an illustration of the entire transformer model.

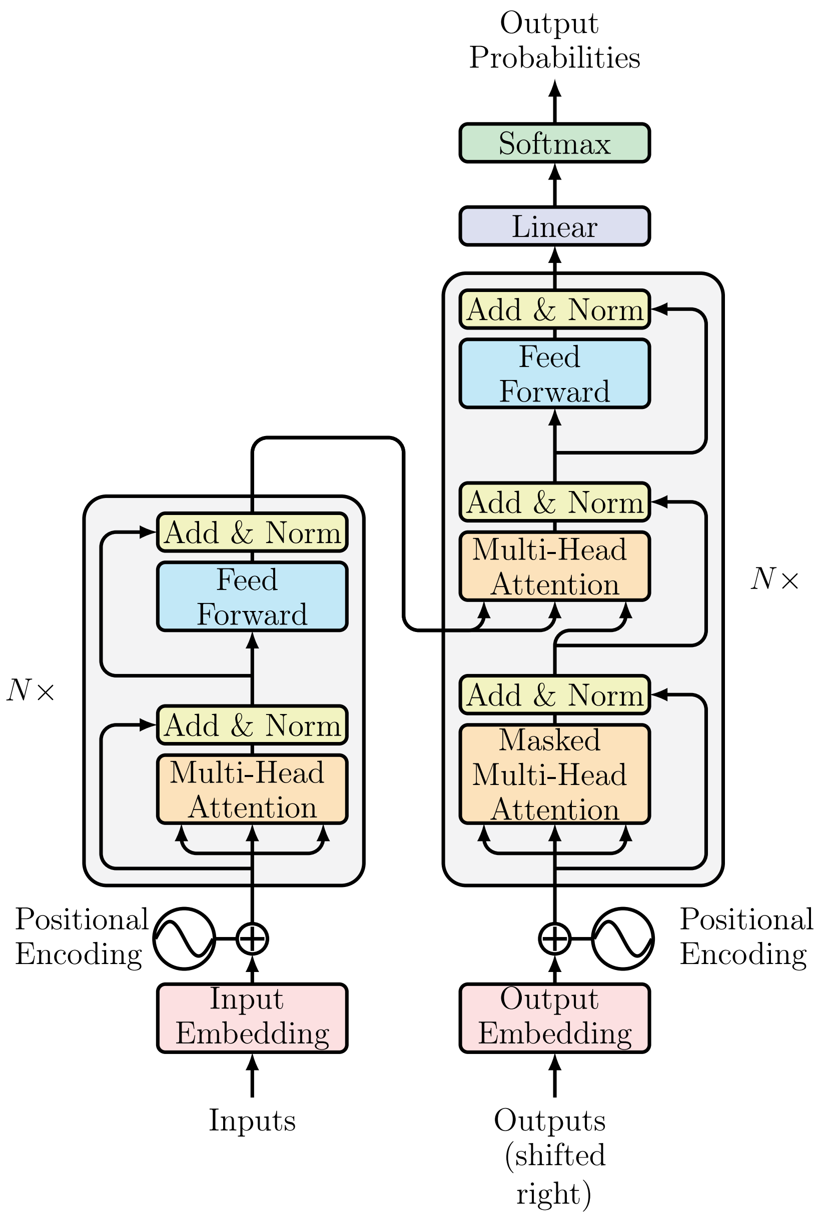


Figure 3.1: Transformer Overview (Vaswani et al., 2023)

3.2.1 Input Embedding - Encoder

Let's break it down from bottom to top and left to right. The initial sequence of words provided by the user undergoes what we call input embedding. Neural networks only operate with numbers, so we have to transform every word into a vector of three (or more) values.

For example the word "Hi" could be represented as follows (assume it is the first token in a sequence).

$$\text{Hi} \longrightarrow \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} 0.1 \\ 0.54 \\ 0.29 \end{pmatrix} = x_1$$

$$x = (x_1, x_2, ..., x_K),\ x_i \in \mathbb{R}^d$$

This process is known as tokenization. Although, the example shows a simple word tokenization where every word is turned into a vector, other and more efficient tokenization methods exist. For example, the Byte-Pair-Encoding works by merging frequent pairs of characters and creating a new vocabulary. This works well because it can capture rare and out-of-voacbulary words by breaking them into smaller chunks of data. A final matrix $E_{|V|d}$ is created where $|V|$ is the size of newly formed vocabulary and d is the dimension of encoding vector for every token.

3.2.2 Positional Encoding

Since a transformer doesn't have recurrence as a recurrent neural network but still needs to keep track of the relative position of the words, the vectors obtained for every token get added to what is called a positional encoding vector (Saeed, 2023). The positional encoding vector is calculated using a piecewise trigonometric function. Let *pos* refer to the token in the sequence that we are analyzing and i is index of vector component, while d is the dimension of the vector. We write:

$$PE(pos, 2i) = \sin(\frac{pos}{10000^{2i/d}})$$
$$PE(pos, 2i+1) = \cos(\frac{pos}{10000^{2i/d}})$$

The reason we use trigonometric functions is because a linear function can produce an exploding output for larger sequences, making it harder to work with. Furthremore, because of the "larger" number of dimensions of the input embedding vectors, we will have a variety of different results from the sinusoidal function between -1 and 1, making it easier to draw relationships between different positions in the sequence.

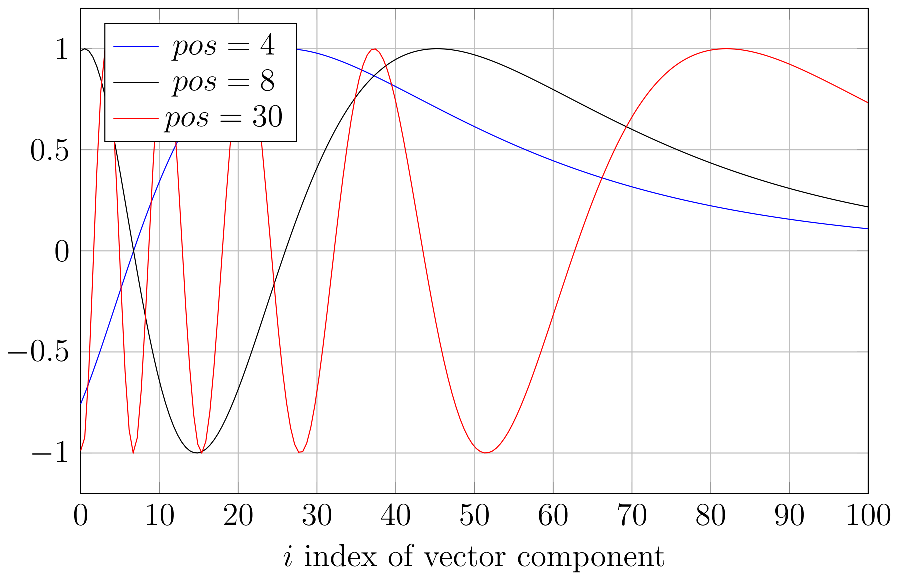


Figure 3.2: The Sinusoidal Function For Different Positions

The figure above shows the sinusoidal function we proposed earlier and its value for different positions of tokens. These functions have proven to be very efficient in practice for three main reasons:

Reason 1: The sinusoidal function for every position is unique

Having unique sinusoidal function for every position is particularly necessary as otherwise we would have conflicting findings. Two tokens cannot have the same positional data.

Reason 2 : Normalized values [-1,1]

Helpful for keeping numbers relatively small and training the model

Reason 3 : Periodic function

Crucially important so that tokens with different positions can develop a relationship with each other in different components of their vector. In other words, tokens are associated with each other on a periodic basis. On the other hand, if a linear function was used, tokens that appear further in the sequence would have heavily different outputs from tokens that appear sooner in the sequence, therefore context will be lost.

The final vectors obtained after adding positional encoding are called positional embedding vectors.

$$\text{Hi}_1 \longrightarrow x_1 \longrightarrow x_1 + p_1, \text{where x is input embedding and p is positional encoding}$$

After the words are completely turned into numerical data, the next step is create an information structure that represents relationship between tokens. This allows us to capture patterns and assign a value to quantify how much one token is related to another one.

3.2.3 Multi-Head Attention

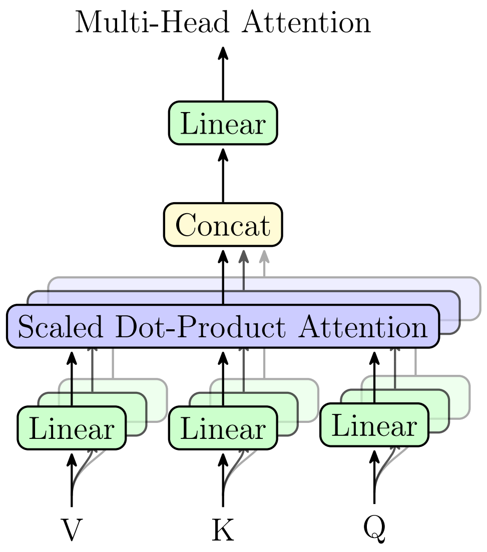


Figure 3.3: Multi-Head Attention in Encoder Layer

The purpose of the Multi-Head Attention part is to take the provided input sequence (positional embedding vectors) and create a representation of the information learned about that sequence, like relationship between words. The positional embedding vectors are fed into three separate layers, the value layer(V), the key layer(K), and the query layer(Q). All of the three seperate copies of the positional embedding vectors undergo different linear transformations. The specifics of each linear transformation is learned through training, but all transformations keep the same dimensions, i.e. $T : \mathbb{R}^d \mapsto \mathbb{R}^d$. After all linear transformations have been achieved, the key and query layers undergo a dot product matrix multiplication to achieve the attention weights of the sequence. Notice that the matrix is composed by concatenating the positional embedding vectors of all tokens in the sequence. The resulting matrix after dot product multiplication is called "the attention weights" and it represents the degree of relationship between any two words in the input sequence.

	How	are	you
How	95	78	56
are	78	93	89
you	56	89	94

Table 3.1: Example Matrix After Dot Product Operation

This matrix is then scaled down by dividing every value with $\sqrt{d}$ (dimension of vectors). The resulting matrix is plugged into a softmax function to produce a matrix with values between 0 and 1. Below is the generalized formula used to calculate the new value in every cell of the matrix.

$$softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j)}, \text{ for } i = \{1, 2, 3, ..., k\} \text{ and } x = (x_1, x_2, ..., x_k) \in \mathbb{R}^k$$

Lastly, we multiply the obtained matrix with the value matrix(V) to produce the final output matrix. This entire process is achieved N times, where N is the number of heads in the transformer model (this is a hyper/predetermined parameter). All the different output matrices obtained by different heads are then concatenated in one last matrix that then undergoes another linear transformation. To summarize the algorithm:

$$K = V = Q = \{\text{set of the positional embedding vectors}\} \in \mathbb{R}^{\text{size of sequence}\cdot d}$$

$$K \to \text{Linear}$$

$$V \to \text{Linear}$$

$$Q \to \text{Linear}$$

$$Attention = K \cdot Q$$

$$Attention := Attention/\sqrt{d}$$

$$Attention := Softmax(Attention)$$

$$Output = Attention \cdot V$$

$$Concatenate$$

$$Output \to Linear$$

3.2.4 Layer Normalization & Feed Forward

The output matrix gets added(element-wise) to the initial input embedding matrix to preserve information from the original input embeddings. This updated matrix then gets normalized and is passed to a feed-forward neural network.

$$Output := Output + x, \text{ where x is matrix composed of input embedding vectors of tokens}$$

$$Output \to Linear \to ReLU \to Linear \to ... \to ReLU \to \text{Layer Normalization}$$

3.2.5 Output Embedding - Decoder

This layer in the transformer architecture works exactly as the input embedding and position encoding, but the only difference is that instead of the input sequence it takes the previous output. For example, if the original input sequence is "I woke up early in the morning" and the first output made by the model is "and decided to", then only the second part undergoes the embedding. Of course, this layer only happens if there has been a previous output. On the other hand, if there is no previous output i.e. we just entered the original input sequence, this layer is skipped.

3.2.6 Multi-Headed Attention 1 - Decoder

The Multi-Headed Attetion 1 in the decoder layer refers to the same mechanism as in the encoder layer, but instead of the original sequence, the previous output given by the model is used as the input. Just like in the encoder layer, the output from this Multi-Headed Attention gets added to the input embedding and is then normalized.

3.2.7 Multi-Headed Attention 2 - Decoder

The second Multi-Headed Attention in the decoder layer takes not only the output from the feed-forward neural network in the encoder layer, but also the output from the first Multi-Headed Attention in the decoder layer. This ensures that the future response will be both related to the previous outputs, but also to the original input sequence. For example, if the original input sequence is a question, we don't need to store only the last outputs of our model as the context, but also information about the original question so that the context or point of the interaction is not lost. To merge the two outputs, we make simple matrix addition operation and again initiate the Multi-Headed Attention mechanism. The final output from this step gets normalized and is passed to a new feed- forward neural network.

$$output_{encoder} + output_{\text{Multi-Headed-Attention 1}} \rightarrow \text{Multi-Headed-Attention 2} \rightarrow$$

$$\rightarrow \text{Layer Normalization} \rightarrow \text{Feed Forward} \rightarrow \text{Layer Normalization} \rightarrow Linear \rightarrow \text{Output Probabilities}$$

3.2.8 Final Output Probabilities

As shown in the flow chart above, after the results of the Multi-Headed-Attention 2 have been plugged in a neural network and are later normalized, we go through one final linear layer. However, unlike all the previous linear transformations, this particular one maps the output of the decoder layer to the space of the target vocabulary.

$$r = \text{ number of rows in decoder output matrix}$$

$$c = \text{ number of columns in decoder output matrix}$$

$$V = \text{Size of total vocabulary of tokens}$$

$$L(M) : \mathbb{R}^{r \text{ x } c} \mapsto \mathbb{R}^{V}$$

The resulting vector with V components is plugged in a softmax function to produce the probabilities of each token in the possible vocabulary occuring as the next element in the running sequence of tokens. Logically, the token with the highest probability is the one we will choose to append to the sequence. This output is then plugged as the input to the decoder layer and this whole process is repeated until an end token has been produced. In the end, the text which was created by the transformer is compared to the correct response using evaluations metrics such as perplexity and the weights,parameters, and linear transformations are adjusted through back-propagation.

Chapter 4

Training & Evaluation

This chapter will briefly explain the data collection process and the training of the language model.

4.1 Training Data

Now that the architecture of the model has been explained and developed, the next step is to provide training data for the computer to learn from. In our case, it is best to use a large book full of poetry written by Macedonian authors. The dataset that will be used in this paper is a combination of all works by Blazhe Koneski converted in a .txt extension file. Lastly, when training any language model there are a few implementation steps:

Set a batch size : 32

This refers to the token length of the input window that will be provided to the model to learn.

Choose random samples from text with batch size:

The python training file will choose random samples from the dataset and feed the model.

Predict / Feed Forward:

Make a prediction about the given input sequence.

Evaluate loss using perplexity function:

The prediction is evaluated with the perplexity loss function that is a complex entropy based mathematical function. This loss function is implemented directly from the pytorch library in python.

Back-propagation:

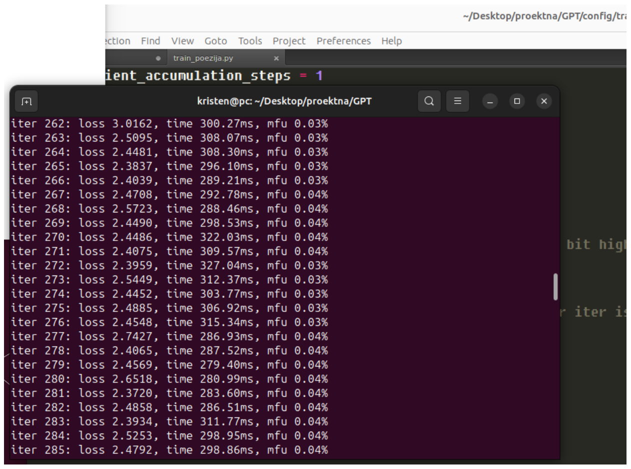
Update the parameters in the transformer model.

Repeat steps 2 through 5

4.2 Iterations & Hyper Parameters

Unlike Chat-GPT or Google Gemini that have entire server farms with extremely large computational power to train the models on, this transformer will only be trained on a smaller computer so it is relevant to discuss the hyper parameters of the model. Very briefly explained, hyperparameters are those parameters that are hardcoded in the system, like number of heads in the multi-headed attention, number of iterations, etc. Keeping this in mind, for this model, we will use:
Number of heads: 6
Number of iterations: 5000
Depth of feed forward neural network: 64
Learning rate α: 10^{-3}
Dimension of vector embeddings: 384
Block size: 256 (context)

4.3 Evaluation

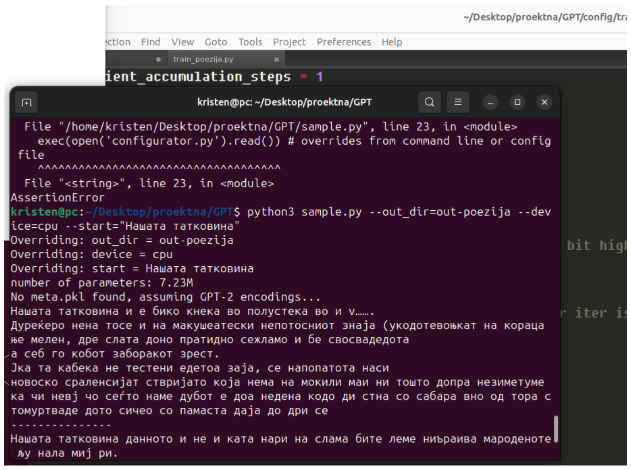

~/Desktop/proektna/GPT/config/tra
ection Find View Goto Tools Project Preferences Help
train_poezija.py
kristen@pc: ~/Desktop/proektna/GPT

```
ient_accumulation_steps = 1
```

```
iter 262: loss 3.0162, time 300.27ms, mfu 0.03%
iter 263: loss 2.5095, time 308.07ms, mfu 0.03%
iter 264: loss 2.4481, time 308.30ms, mfu 0.03%
iter 265: loss 2.3837, time 296.10ms, mfu 0.03%
iter 266: loss 2.4039, time 289.21ms, mfu 0.03%
iter 267: loss 2.4708, time 292.78ms, mfu 0.04%
iter 268: loss 2.5723, time 288.46ms, mfu 0.04%
iter 269: loss 2.4490, time 298.53ms, mfu 0.04%
iter 270: loss 2.4486, time 322.03ms, mfu 0.04%
iter 271: loss 2.4075, time 309.57ms, mfu 0.04%
iter 272: loss 2.3959, time 327.04ms, mfu 0.03%
iter 273: loss 2.5449, time 312.37ms, mfu 0.03%
iter 274: loss 2.4452, time 303.77ms, mfu 0.03%
iter 275: loss 2.4885, time 306.92ms, mfu 0.03%
iter 276: loss 2.4548, time 315.34ms, mfu 0.03%
iter 277: loss 2.7427, time 286.93ms, mfu 0.04%
iter 278: loss 2.4065, time 287.52ms, mfu 0.04%
iter 279: loss 2.4569, time 279.40ms, mfu 0.04%
iter 280: loss 2.6518, time 280.99ms, mfu 0.04%
iter 281: loss 2.3720, time 283.60ms, mfu 0.04%
iter 282: loss 2.4858, time 286.51ms, mfu 0.04%
iter 283: loss 2.3934, time 311.77ms, mfu 0.04%
iter 284: loss 2.5253, time 298.95ms, mfu 0.04%
iter 285: loss 2.4792, time 298.86ms, mfu 0.04%
```

Figure 4.1: Improvement of Loss Function During Training

The above figure is a screenshot from the training process iterations. It's interesting to observe how the loss values decrease even with very few iterations and very limited computational resources.



~/Desktop/proektna/GPT/config/tra
ection Find View Goto Tools Project Preferences Help
train_poezija.py
kristen@pc: ~/Desktop/proektna/GPT

```
ient_accumulation_steps = 1
```

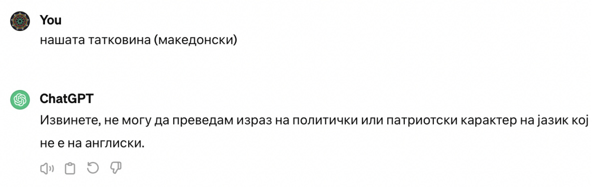
```
 File "/home/kristen/Desktop/proektna/GPT/sample.py", line 23, in <module>
   exec(open('configurator.py').read()) # overrides from command line or config
 file
   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 File "<string>", line 23, in <module>
AssertionError
kristen@pc:~/Desktop/proektna/GPT$ python3 sample.py --out_dir=out-poezija --dev
ice=cpu --start="Нашата татковина"
Overriding: out_dir = out-poezija
Overriding: device = cpu
Overriding: start = Нашата татковина
number of parameters: 7.23M
No meta.pkl found, assuming GPT-2 encodings...
Нашата татковина и е бико кнека во полустека во и v…….
Дуреќеро нена тосе и на макушеатески непотосниот знаја (укодотевоњкат на кораца
ње мелен, дре слата доно пратидно сежламо и бе свосвадедота
а себ го кобот заборакот зрест.
Јка та кабека не тестени едетоа заја, се напопатота наси
новоско сраленсијат ствријато која нема на мокили маи ни тошто допра незиметуме
ка чи невј чо сеѓто наме дубот е доа недена кодо ди стна со сабара вно од тора с
томуртваде дото сичео со памаста даја до дри се
---------------
Нашата татковина данното и не и ката нари на слама бите леме ниљраива мароденоте
 љу нала миј ри.
```

Figure 4.2: A Sample Response From The Trained Model

In the figure above, we can see the model's response to the given prompt. It is very easy to observe that the text does not make any sense, but what is interesting is the fact that punctuation, sentence length, and word length really resembles the Macedonian language. With enough computational resources, enough time, and fine-tuning a base model, this can really become a very sophisticated model.

4.4 Chat-GPT 3.5 Response to The Same Prompt

Given the exact same prompt, Chat-GPT 3.5 produced the following response.



You
нашата татковина (македонски)

ChatGPT
Извинете, не могу да преведам израз на политички или патриотски карактер на јазик кој не е на англиски.

Figure 4.3: Chat-GPT 3.5 Response to The Same Prompt

It turns out that Chat-GPT has not been trained even partially on the same dataset as our model, so it cannot be assessed quantitatively. However, we can see that the language at which it replies is clearly not Macedonian. Very frequently, it returns Bulgarian text as it is not able to distinguish our own language. This is the very problem at which Chat-GPT fails and we would like to solve in the future. Just like there exist language models in other languages, there should be one possesing the ability to write text strictly in Macedonian.

Chapter 5

Conclusion

In conclusion, we have explained and showed how powerful the transformer architecture really is, even when run on a very limited computing machine. The training data and implementation of the model have proven to be efficient taking into consideration the number of iterations that were run in the training phase. This model is certainly far from writing perfect poetry and following semantics, but can be significantly improved with enough time and computational resources. On a last note, state-of-art LLMs like Chat-GPT have also failed to provide a correct Macedonian response to many of the prompts which were given. We showed one prompt to picture the difference between the two models. Nevertheless, problems are meant to be solved and models are meant to be improved.

“Err and err and err again, but less and less and less.”
Piet Hein

Bibliography

[1] Saeed, M. (2023, January 5). A gentle introduction to positional encoding in transformer models, part 1. MachineLearningMastery.com. https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

[2] Drost, D. (2023b, May 12). A brief history of language models. Medium. https://towardsdatascience.com/a-brief-history-of-language-models-d9e4620e025b

[3] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023, August 2). Attention is all you need. arXiv.org. https://arxiv.org/abs/1706.03762