Architecture and Applications of Auto-regressive Generative Transformer Models

Mateusz Jaszczuk

December 18, 2024

Abstract

In recent years, Auto-Regressive Generative Transformer Models were one of the most important research topic in the engineering computing field. Models allow us to predict sequences of tokens based on the training datasets, which can be used in Large Language Model, Natural Language Processing or Machine Translation. Another potential application of the auto-regressive transformer is in the flied of robotics. Model can be train to predict task forces on the robot exerted by humans based on the analysis of their motion. Therefore, in this work, I plan to explore the architecture of auto-regressive transformer model and learn about how they can be applied in Large Language Models. I want this work to be a learning project, which will allow me to understand model architecture and, in the future, apply it in my own robotics research.

1 Introduction

In this project, I plan to explore the model architecture of auto-regressive transformer models. I will apply the model in Large Language Model trained with three different datasets, to create a framework which can generate sequence of tokens in auto-regressive manner using given context. I will be focusing on my work on recreating approach shown in the landmark paper that introduced concept of Self-Attention, "Attention is All You Need" [1].

Motivation for this work is to learn about model structure and understand all components of the transformer block. Auto-regressive models, beyond large Language models, have a large spectrum of possible applications. One that I am particularly interested in is applying the models in the field of robotics. When we consider robot to human interaction, it is important for the robot to anticipate the human's intentions - task forces that someone may exert on the robot. To reinforce the adaptive control, we might apply an auto-regressive model, which will aim to predict possible distribution of task forces within the next couple time steps. This information will be feed into adaptive controller as the prior knowledge, and might help stabilize the control while making sure that the robot will meet the task objectives.

To learn how to apply auto-regressive model to robotics, I plan to explore this architecture in this project by braking down and analyzing all the components of the transformer network. By this exercise, I want to understand all the math behind the model, so in the future, I will be able to apply the model to my own research.

In the project, I will go though the logic behind creating transformer model able to generate new tokens in sequence as well as additional steps that are required to stabilize the training process. I will evaluate the model using three different test datasets, and perform additional evaluation, like training the model without Layer Normalization and Residual Connections, to examine why they are important in the architecture. I will also go over additional steps that are needed to create a Chat Bot after pre-training step, as well as explain possible implementations of the transformer architecture in the filed of robotics.

2 Existing Work

Current state-of-the-art auto-regressive models achieve impressive results in many fields of engineering and computer science. Most noticeable one is ChatGPT, an online chat bot which is capable of answering questions based on given prompt [2]. ChatGPT is an example of auto-regressive pre-trained transformer, which follows the same logic that I will explain in my work. However, I will only explain the pre-training stage of the transformer, while also fine-tuning stage is required to achieve result similar to the product by OpenAI. Nevertheless, in this work I will teach the model how to speak, without teaching it how to answer questions, which is still interesting part of the framework.

Another application of transformers was shown in the paper "Attention is All You Need" [1] is Machine Translation. In such application, we create an encoder-decoder transformer which is able to translate the machine input into statistical information or language. This application can also be extended to translating text from language to another language.

Similar application of transformer model was implemented in the robotics field to allow robots learn grasping motions from videos of humans [3]. Such studies are object of research by Prof. Antonio Loquercio from the University of Pennsylvania Electrical and Systems Engineering department. During a gust lecture in Introduction to Robotics class that I attended, Professor explained how he applies transformer models to learn grasping motions for the robots by showing the framework different videos of humans grasping objects. This application shows that we can reliably apply transformer frameworks in robot learning.

To summarize, there are numerous different applications of transformer models in the world. We can basically apply the framework to any task, where we have a history of sequences, and we want the model to predict next token based on the history that it was exposed to. Therefore, I believe this framework can be applicable in predicting humans behavior in robot to human interactions.

3 Methodology

The framework of auto-regressive predictive transformer networks is a combination of various neural network techniques, such as multi-layer perception, layer norm, or residual neural networks, which are applied in sequence. What makes the transformer architecture unique and able to predict the data sequences is the Self-Attention mechanism, which was first introduced in [1]. This combined creates a framework where attention is followed by computation, which allows the model to effectively predict the sequences of tokens, either in machine translation, Large-Language Models, or in any other applications. In this section, I will break down the auto-regressive model architecture and explain the information flow within the network.

In this project, I will focus on creating auto-regressive transformer to analyze the input text and predict tokens (another text) based on that. I begin this process with data tokenization, which is translation of text into tokens, which will be further analyzed by the transformer block. Then, I create position and token embedding table, which will be used by the self-attention block to learn dependencies between different tokens. Further, we pass this information to Feed Forward Neural Network, where model will learn patterns in the input. Finally, we map the transformer predictions from the embedding dimension back to vocabulary size of our tokenizer to compute probabilities of observing particular next token in the sequence.

Project focuses on architecture called encoder only transformer. This implies that we will allow the model to learn only from the previous tokens in sequence, allowing the network to successively predict future tokens. Differences between encoder and decoder transformers will be explained further in the report. High-level framework overview for the project is shown in Figure 1. In this section of the report, I will break down this architecture and explain functionality of each of the blocks.

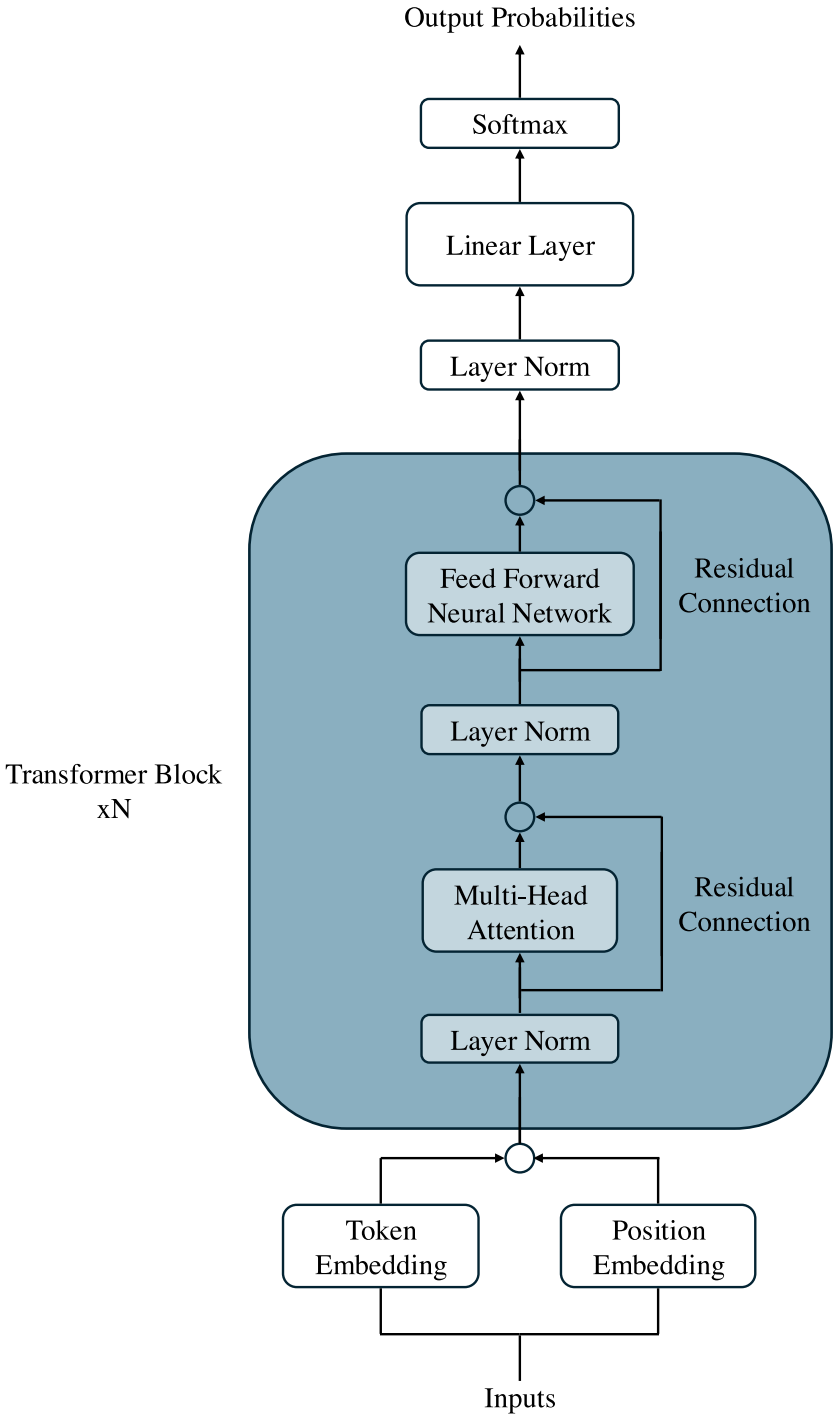

Figure 1: Project framework overview.

3.1 Data Tokenization

In order to process any data using the transformer networks, we need to tokenize it - translate input, in our case text, to a list of integers - tokens. Tokenazation can be performed differently, based on the application. In this framework, I will implement two main tokenization methods:

- **Character-Level Encoding** - This method breaks down every single character in the text into separate tokens.
- **GPT-2 Encoding** - Byte Pair encoding method, which breaks down sub-pieces of words into separate tokens, while also allowing character level encoding. Byte Pair Encoding method is widely use in Natural Language Processing and was first introduced in [4].

Character-level tokenization is a simple and straight-forward way of encoding text into token sequences. It takes a string, which is a series of characters, and transforms it into token list, which is series of integers, as shown in the example below.

$$\text{Thank you!} \xrightarrow{encode} \begin{bmatrix} 42 & 59 & 52 & 65 & 62 & 1 & 76 & 66 & 72 & 2 \end{bmatrix} \xrightarrow{decode} \text{Thank you!} \tag{1}$$

Advantage of this method is it's clear simplicity. Tokenizer is straight-forward by translating every single string element into an integer - same applies to special characters, like exclamation mark, space, or starting a new line. This method, however, enforces the Language-Model to predict new letters in every single workds, which requires a bigger network to achieve similar results. It also makes the process more computationally expensive as we need 10 tokens to encode "Thank you" message.

GPT-2 type encoding focuses on encoding sub-pieces of words, instead of individual characters. Comapring to character-level tokenization, which in our example ahs a vocabulary size of 85, it has a vocabulary size of 50257. This allows the model to analyze the same text using shorter sequences of tokoens, as shown in the example below.

$$\text{Thank you!} \xrightarrow{encode} \begin{bmatrix} 10449 & 34 & 0 \end{bmatrix} \xrightarrow{decode} \text{Thank you!} \tag{2}$$

Clear advantage of the GPT-2 encoding is the fact that it encoded our message using only 3 tokens, instead of 10. This means that in our network to analyze a given piece of text, it will need shorter sequence of tokens, which reduces model computational cost and allows us to feasibly create larger models. Disadvantage of this approach, however, might come when we deal with a very specific data sets, like text written in Early-Modern English. Because the vocabulary might not include some words from 400 years ago, tokenizer might still need to break it down to character level, which reduces the computational advantage and may further confuse the model.

In this project, I will mainly use character-level tokenization, as it allows easier and less complicated model tuning while working with different training datasets. However, in the evaluation section, I will compare the performance of both tokenization methods side by side using the same training text.

It is also worth mentioning that similar tokenization approaches can be applied to inputs other than text. We can, for example, use the autoregressive-transformer model to predict the next frame of a video using a sequence of pictures as input. In this case, we can tokenize our data by encoding different pixels in the network.

3.2 Token and Position Embedding

We need to perform the embedding process to allow the model to predict the next token in the sequence. It creates the embedding tables, which are storing the information about each token in sequence. To generate meaningful text, we need:

- **Token Embedding** - Embedding which allows the model to understand the meaning of the token (either a character or a word).
- **Position Embedding** - Embedding allows understanding particular tokens in the context of other tokens in the sequence.

By embedding process, we create embedding tables, which are the spatial information about token meaning and position. We begin by defining the number of embedding dimensions in our network, which is the vector size that will store the information about the token.

For token embedding, we create an embedding table between the vocabulary size and the embedding dimension. This will result in an embedding table for each token that will store the information about the token's meaning in terms of all vocabulary in the tokenizer and embedding dimension. It is worth mentioning that the number of embedding dimensions can be adjusted as the model's hyperparameter. A higher number of embedding dimensions may allow the model to learn more complex token meanings; however, it may also lead to overfitting.

Position embedding is a similar process, which instead of embedding the vocabulary size, uses all tokens in the sequence of training data. This process allows to store the positional information for each token in the context of all tokens that happened before. As a result, we are able to predict token not only on its meaning, but also position in the context of other tokens.

$$E = E_{token} + E_{position} \tag{3}$$

To combine this information, we add token and position embedding tables, which are further fed into the transformer block as input. All parameters inside the embedding table are trainable, which means that the model can learn dependencies during the training process.

Combining token and position embedding is essential in all applications of auto-regressive transformers. We need to make sure that the output of the model is meaningful in the context of previous tokens. Beyond LLMs, this is also vital in robotics applications, where is we want to predict the task force exerted by human based on their previous actions, we need to make sure that our transformer will understand the context well.

3.3 Self-Attention

The most interesting part of transformer architecture and the most significant contribution made in [1] is Self-Attention. It is a mechanism that allows different tokens in the sequence to learn information from different tokens around them. By aggregating this information, the model can learn from data and predict the next sequential tokens.

In the self-attention block model, we take embedded inputs and quantify relative information for each token based on the token information and information in tokens around it. We create three vectors to mathematically represent this concept: query, key, and value. in self-attention, for every token, we emit a key, which quantifies question information that this token is looking for in the data around. Key vector is an answer to query - it poses information that about tokens around the token of interest. By computing the dot product between the query and the key, we obtain a matrix that quantifies each token's affinities - information from which tokens we want this token to learn. The higher the affinities between tokens, the more tokens are related to one another. Because our embedded input has information about tokens values and position, the self-attention model can successfully account for both of these.

After computing attention weights, we need to normalize them by the square root of key vector size. This ensures that no attention scores will have very high values, which will be problematic for the softmax function to deal with (it may converge to spikes in data). Normalization also stabilizes the training process.

Further, we normalize the attention scores using the softmax function. This will cap all attention scores between 0 and 1, making sure they all sum to 1. Applying softmax to the probability distribution (attention scores) allows us to obtain a meaningful and easy-to-interpret distribution of attention weights. By applying softmax, we ensure that none of the values will converge to a very high or very low value, which would bias the attention process; as a result we allow the model to focus on meaningful information that can be observed from tokens in the sequence.

Finally, we compute final attention scores by aggregating the attention weights by a value vector. This will quantify how much of the information observed from the data will be actually used in the model - defining the importance of the data. This all results in self-attention equation as shown below:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_K}})V \tag{4}$$

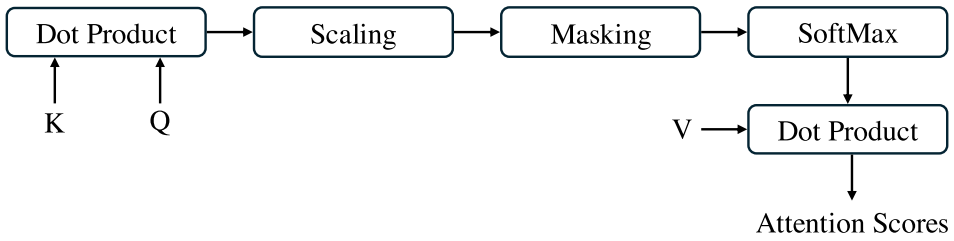

Figure 2: Self-Attention overview.

where Q, K, and V are query, key, and value vectors, respectively, and d_K is the length of the key vector. This attention score is used in the transformer block to compute the affinities between different tokens - tokens communicate with one another and share information. Note that all query, key, and value are trainable parameters, which means that model itself will determine what information and at which importance it wants to learn from different tokens, and how much it wants to apply it.

The method described above allows the model to learn from all tokens around in the batch, both these before and after. This method is applicable in, for example, machine translation. However, if we want to create an auto-regressive model, we need to restrict the attention process to learning only from previous tokens - attention cannot be seen in the future. To achieve this, we create a masking block, as indicated in the figure above.

If we want to restrict the self-attention to learn only from the past, we need to consider only the self-attention scores that are dependent only on the tokens that happened before. To achieve this, we mask the output of dot product between queries and keys (which is a square matrix) with lower triangular matrix, masking the upper triangular matrix to negative infinities. When we compute a softmax of such a masked matrix, upper-triangular part of the matrix will be set to 0 (softmax output for negative infinity), while lower-triangular part of the matrix will be properly normalized with the softmax. As a result, our attention scores will be only based on the previous tokens - all attention to future tokens will be set to 0 - our model will be interested in learning only from the past tokens. Schematics of such masking is shown below.

$$\begin{bmatrix} 0.0880 & 0.2976 & 0.3876 \\ 0.3238 & 0.8666 & 0.7062 \\ 0.9651 & 0.3739 & 0.6863 \end{bmatrix} \xrightarrow{Masking} \begin{bmatrix} 0.0880 & -\infty & -\infty \\ 0.3238 & 0.8666 & -\infty \\ 0.9651 & 0.3739 & 0.6863 \end{bmatrix} \xrightarrow{Softmax} \begin{bmatrix} 1.0000 & 00000 & 00000 \\ 0.3675 & 0.6325 & 0.0000 \\ 0.4328 & 0.2396 & 0.3275 \end{bmatrix} \tag{5}$$

Using the approach shown in the equation below, we can mask complex self-attention matrices to force the model to learn only from the past tokens. This creates a self-attention architecture which is implemented in this project.

Note that in different applications than auto-regressive models, we can implement self-attention without masking, which can be applied, for example, in machine translation. Our architecture is called a decoder-only transformer while removing masking will give us an encoder-only transformer. We can also implement an encoder-decoder transformer, which will be further explained later in the report.

3.4 Multi-head Attention

In order to make the model more robust and reduce the computational time required to process the self-attention, we add Multi-Head Attention to the transformer. Multi-head attention is simply multiple heads of self-attention running in parallel. This concept allows the model to explore the data more complexly, catching different features from the context with different heads of self-attention. Also, due to the usage of the CPU, we are able to run all self-attention heads in parallel, which reduces the computational time required.

In multi-head attention, we call multiple self-attention layers with adjusted embedding dimensions—we divide the number of embedding dimensions by the number of self-attention heads. This process makes the model more computationally feasible. Then, we allow the model to run all self-attention heads in parallel and concatenate the results. This creates a tensor of self-attention scores that are dependent on different features in the data - some of the heads might be picking up the local

context, while others can focus on the global context. At the end, we apply a linear transformation to adjust the size of multi-head attention to have an output of a number of embedding dimensions (exact as input). This allows us to pass the data to the Feed-Forward Neural Network and then repeat the transformer block sequentially. Framework of the multi-head self-attention is shown in Figure 3.

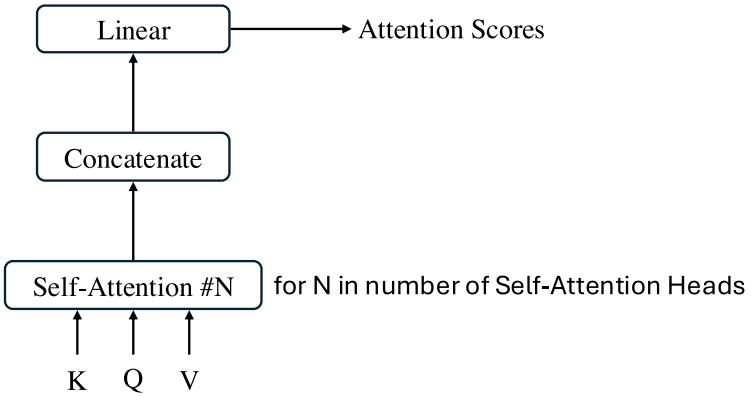

Figure 3: Multi-Head Attention overview.

The advantage of that method is that we allow the model to learn more independently, which results in better results - we allow different self-attention heads to learn different features from the context, for example, long or short dependencies. We also reduce the computational time required to get attention scores, which in total makes the framework more robust.

3.5 Feed Forward Neural Network

Transformer architecture uniqueness is combining attention with computation. In previous sections, I described the attention mechanisms, which allow the model to learn dependencies between tokens. To allow our model to learn more complex features from the data, we must introduce non-linear operations - self-attention only focuses on linear transformations from the data, which will be the computation part of the transformer block. We need to add an additional component into the transformer block, which is a simple Feed-Forward Neural Network that includes non-linear transformation, ReLU. This allows the model to learn from global and local patterns more complexly, dramatically improving model performance.

In this application, we will introduce a simple Multi-Layer Perception Neural Network, as it was introduced in the original transformer work [1]. In this architecture, we take the input, which is the embedding table output from multi-head attention block, and apply a Neural Network with two hidden layers, as shown in Figure 4.

In the network, we take input of size of embedding table dimension, apply linear transformation to size of NN layer, follow that by a ReLU non-linearity, and transform to output of the original size, as shown in equation below.

$$Output_{FF-NN}(x) = ReLU(xW_1 + b_1)W_2 + b_2 \tag{6}$$

This architecture is a simple way of introducing complexity and non-linearity to the model, which allows the transformer to learn complex patterns from the data. Due to simple architecture, which results in output of the same dimension as the input, we can use this network in the transformer block - final output dimension of the transformer will be the same as its input dimension, which allows us to call the transformer block sequentially.

The network-tuning process requires adjusting all hyperparameters of this neural network. Results with different hyperparameters (FWD—dimension of hidden layer in Feed-Forward Neural Network) will be presented in the Evaluation section. A general rule is to keep the dimension of hidden layers four times bigger than the embedding dimension of input.

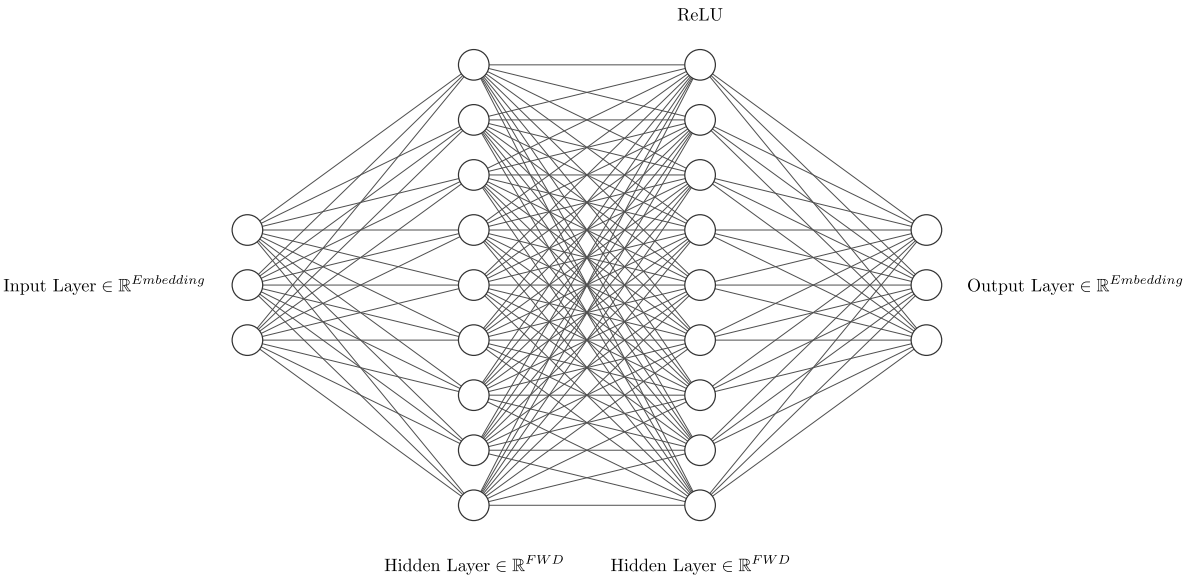

Figure 4: Schematics of Feed-Forward Neural Network.

3.6 Layer Normalization

As we developed the transformer model, we reached the moment that our network had close to a million parameters, considered a large Deep Neural Network. There are several challenges with training such algorithms, which, if not addressed properly, can lead to gradient diminishing or training instability.

Potential solution to this problem is adding Layer Normalization, also called LayerNorm, first introduced in [5]. This concept assumes adding additional LayerNorm layer to Neural Network, which normalizes the output to zero mean unit variance. This allows the framework to deal with several issues that are listed below.

- **Training Instability** - At the initial phases of training, inputs to different transformer block might have different scales from one another due to weight update rule. This may lead to very large gradients in the update process, which can further make the training process very unstable. Applying LayerNorm reduces all inputs to zero mean, unit variance, which effectively stabilizes the training process.
- **Gradient Diminishing** - Due to different scales of parameters in the networks, some gradients in the training process might become very large or very small, which leads to gradient explosion or diminishing. As we might be loosing some of the gradient values, the optimization process will become very inefficient. normalizing parameters effectively prevents that issue.
- **Softmax Convergence** - As we are using softmax function to compute final output probabilities, we need to be careful output inputs we give to the function. If softmax is given values with very high variances, it might converge to the spike in data, which will not be the correct representation. Therefore, we also need to normalize the softmax input in the final layer after the transformed blocks.

Mathematica formulation of the LayerNorm, which was introduced in [5], assumes normalizing the inputs to the function to zero mean and unit variance, as shown in the equation below.

$$LayerNorm(x) = \frac{x - \mu}{\sigma}\gamma + \beta \tag{7}$$

Basically, we normalize all the inputs of the function about their mean and standard deviation. To allow parameters achieve higher or lower values in the long term, we introduce γ and β parameters, which are scale and short parameters. These are trainable, and will be optimized in the training loop,

which allows the model to learn true distributions of the data in a stable manner, preventing all risks mentioned above.

In the auto-regressive framework, we apply the LayerNorm before each transformer sub-block, Multi-Head Attention and Feed-Forward NN, as well as before calculating softmax probabilities after leaving transformer blocks. This approach guarantees that we will prevent gradient diminishing and training instability withing transformer, as well as incorrect convergence of softmax probabilities at the end of the network.

3.7 Residual Connections

As mentioned in the previous section, gradient diminishing is a large problem in training large Deep Neural Networks. Gradients can become very small during the back-propagation process, which makes it very difficult for certain layers to optimize their particular parameters. An additional step to prevent that from happening is adding Residual Connections, concept first introduced as ResNet (Residual Neural Network) in [6]. Concept assumes that we will add input of the layer to its output, making the network learn the residual between them. This effectively prevents gradient diminishing; it is important, however, to make sure that layer to which we apply the Residual Connection is having the same size of output as input. Equation below shows application of Residual Connection to Multi-Head Attention block.

$$Output = x + MultiHeadAttention(x) \tag{8}$$

Adding a residual connection, also called a skip connection, is a very simple and effective way of preventing gradient diminishing. It also does not require any trainable parameters, which means it does not have a high computational cost itself. Usually, it is applied alongside with LayerNorm, which creates a robust structure that allows effective training of Deep Neural Networks.

$$Output = LayerNorm(x + MultiHeadAttention(x)) \tag{9}$$

Equation above is a mathematical representation of adding a residual path to Multi-Head Attention block alongside with LayerNorm. As shown, it is a simple operation which does not introduce model complexity, but improves the overall training performance.

3.8 Transformer Block

The main component of the auto-regressive transformer model is the transformer block itself. As introduced in the landmark paper [1], the transformer is a Neural Network block in which we compute attention followed by computation. Attention allows us to analyze affinities between different tokens and understand their meaning and context in the sequence. Computation, conducted by the Feed-Forward neural Network, allows the model to learn patterns from data and understand what the next token prediction can be in a non-linear manner.

In a transformer, we first normalize the input using Layernorm and then pass it to Multi-Head attention, which outputs attention scores. We then add a residual connection, normalize it again, and pass it as an input to the Feed-Forward Neural Network. Neural Network processes the data within the given architecture. We then add another residual connection and output the data from the transformer block. The framework architecture of the transformer block is shown in Figure 5.

In both Multi-Head Attention and Feed-Forward Neural Network, we ensure that outputs are of the same spatial dimensions as input and that the whole transformer block possesses the same quality. This means that we can call the block sequentially multiple times, which allows the auto-regressive model to learn more complex features from the data.

3.9 Model Prediction and Loss Computation

After we get output from the last transformer block, we apply a final LayerNorm to normalize model parameters and make the training process more stable. Then, we apply the final Linear Layer, mapping the outputs from the number of embedding dimensions to vocabulary size. This allows us to get a meaningful output in terms of all possible tokens that can happen in the sequence. Finally, we

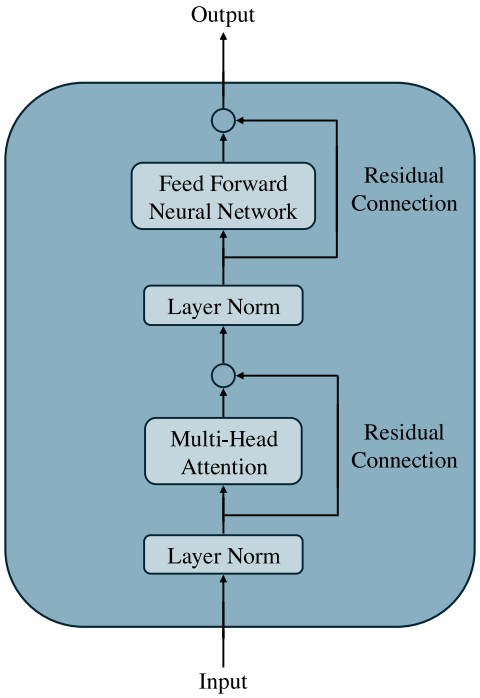

Figure 5: Transformer block overview.

apply the softmax function to compute probabilities of any given token being next in the sequence, as shown in the equation below, where x is the model output after last linear fully-connected layer.

$$Output\ Probabilities = Softmax(x) \tag{10}$$

In the generation process, we sample from these probability distributions over the vocabulary size, which gives us the most probable next token. We apply this to the generated block, predicting tokens one by one. The new sequence is passed to the generation block as the new context, and the process is repeated.

To properly evaluate model training performance, we calculate the loss function, which in our application is the cross-entropy loss, as shown in the equation below.

$$Loss = -\sum_{i=1}^{\text{vocab size}} y\ log(\hat{y}) \tag{11}$$

In this equation, we quantify the mismatch between true label y and predicted probabilities of output $\hat{y}$ for all classes, which in our case is the vocabulary size of the encoder. This equation allows us to simply quantify the error between model prediction and the actual accurate label. We can obtain gradients used in the optimization process by taking the gradient of this loss function using automatic differentiation.

To compute the gradients of the loss function, we use the concept of back propagation. We first performed the forward pass, where we passed all inputs through the network. Then, we compute gradients with respect to the final layer, and using the chain rule, we back propagate them throughout the network to compute gradients on all parameters. These gradients are then optimized and used for the weight update.

To optimize the model parameters, I used the ADAM Optimizer (Adaptive Moment Estimation), which is a powerful estimator introduced in [7]. The optimizer uses an adaptive learning rate by computing momentum - based on a weighted average of past gradients. Optimizers can effectively accelerate the learning process in the early training stages while preventing oscillations close to model convergence. The ADAM optimizer is a popular, robust method used to train most modern neural networks.

3.10 Scaling the Model and Dropout

To create a large auto-regressive model that will be able to learn from large datasets, we need to scale it up by adjusting the model hyperparameters. Depending on type of encoding, we need to select a batch block size appropriately, so the model can analyze the dataset in a computationally feasible manner. In this process, we make sure that our model is complex enough that it will be able to capture a lot of complex features from the data.

The potential issue with adding complexity to our model is the risk of overfitting. This means that our training process will optimize weights based on the training data in a way that it works only for the particular batches that the model was exposed to. In other words, train loss of the model will drop significantly below validation loss, and validation loss might even diverge during the training process.

To mitigate that risk, I applied dropout, the concept introduced initially in [8]. Dropout is a random drop of connections inside the Neural Network during the training process - different connections are being dropped for each interaction. This allows the model to learn in more robust way, without focusing on features that are exclusively present in the train data. Figure 6 shows an example of Dropout applied to a simple Neural Network with one hidden layer. Note that we do not drop these connections anymore in the generation process - dropout is only used to prevent overfitting in training.

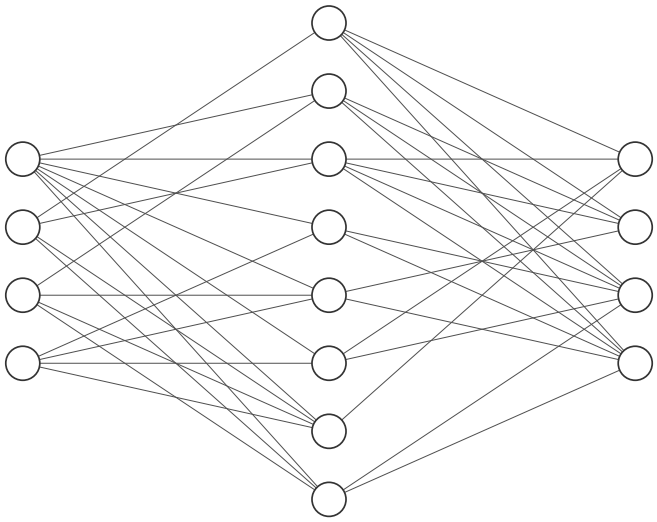

Figure 6: Neural Network with dropped connections - Dropout.

Using dropout, we are able to train larger neural Networks minimizing the risk of overfitting. In my model, I applied dropout three times - after each Single-Head Attention block, after Multi-Head Attention, ad after Feed-Forward neural Network. It allowed me to train the model on over 1,000,000 parameters without significant over fitting.

3.11 Framework Summary

To summarize the model framework, we first encode the input dataset using given tokoenizer. Then, we create embedding table, based on position and token embeddings of the input data. Further, we pass the data through multiple self-attention blocks, and then pass it to feed-forward neural network. We repeat this process (transformer) multiple times and finally output the data that is being transformed into probability distributions over vocabulary size of the tokenizer.

In the process, we apply LayerNorm and residual connections to stabilize the training process and prevent gradient diminishing. We also include dropout to prevent the network from overfitting.

This framework allows us to create the auto-regressive transformer, which can generate infinite text based on the training dataset. We can apply this framework to any given dataset, or apply it to different applications, like machine translation.

Framework described in the Methodology section follows the flowchart shown in Figure 1. To further elaborate on this work, we can add cross-attention block, which allows creating an encoder-decoder transformer.

4 Evaluation

To evaluate the performance of the auto-regressive transformer I applied in this project, I will train the network using three different texts and compare the output and losses. I will also conduct additional experiments with the framework, like training the model with different tokenizers, with and without residual connections and LayerNorm, as well as with Single-Head Attention only. I want to examine model behavior under different conditions to learn why particular blocks in the framework are important and how they influence the predicted output.

4.1 Model Testing with Different Datasets

To ensure the model's capability to predict new text based on any given text, I evaluated performance using three different datasets: TinyShakespeare - a collection of all known works of Shakespeare; Robinson Crusoe - a novel by Daniel Defoe, and Forest Gump - a movie script directed by Robert Zemeckis. These datasets are collection of texts from different ages and of different lengths, which will provide a meaningful representation of inputs to model evaluation.

All models are trained at Google Coolab Research Platform using A100 GPUs. This is most powerful computation resource available at the Coolab, which is necessary to train models of such a complexity.

4.1.1 Test Case 1 - TinyShakespeare

The first test case of the model is an auto-regressive transformer trained on TinyShakespeare. The challenge of this data set is the fact that it was written in Early-Modern English, which might be confusing for modern tokenizers, like GPT-2. Therefore, in this evaluation, I choose character-level encoding. To train the model. Model hyperparameters and training parameters chosen for this model are shown in Table 1.

Model Parameter	**Value**
Training Iterations	6,000
Learning Rate	3e-4
Model Parameters	14,335,553
Encoding Type	Char
Transformer Blocks	8
Self-Attention Heads	8
Embedding Dimension	384
Batch Size	64
Context Block Length	256
Feed-Forward NN Layer Dimension	1,536
Dropout Rate	0.3
Training Time	17:21 min

Table 1: Model parameters for auto-regressive transformer training on TinyShakespeare dataset with character-level tokenization.

Using the training parameters shown in the table above, I successfully trained the auto-regressive model using TinyShakespeare dataset. As shown in Figure 7, both training and validation losses converged to values below 1.5, which suggests that model was trained properly.

Figure 8 shows example output of 500 generated tokens using the auto-regressive model trained on the TinyShakespeare. As we can observe, model successfully learned from the dataset, and was able to generate Shakespeare-like output, which is meaningful and consistence. There are grammar errors in the generated text, which suggests that tokenizer was not effective and that the model training could have been further improved. Nevertheless, the output is a validation that the auto-regressive transformer was indeed trained to generate text based on the given input dataset, which is a promising result about model capabilities.

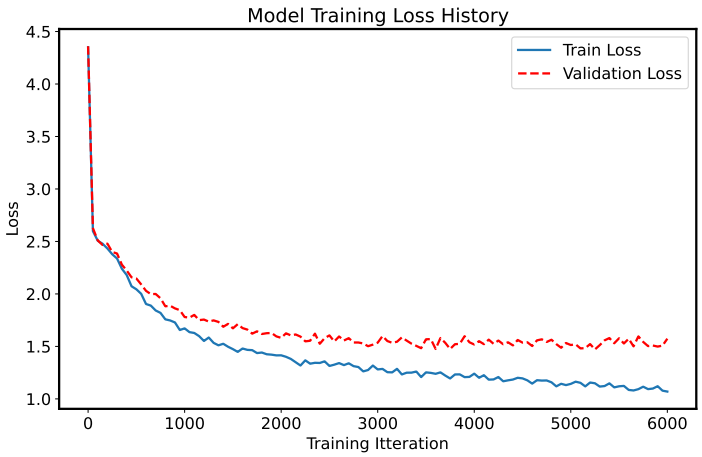

Figure 7: Train and validation loss histories for training auto-regressive transformer model on TinyShakespeare dataset.

```
### Model Prediction ###

QUEEN MARGARET:
A cursed for haste, keep fatal in pity.
Qut let me to heaven, we repeal Clarence,
And pardon 't: Richard canst thou do-fwell
The quarten maid through the presence of his form.
3 KING HENRY VI

GLOUCESTER:
His noble kniegrow, man! what should he lay so?

KING EDWARD IV:
then at myself the quarrel of wrongs;
My brother of? Myself had wrings twaits in our peace, canst
A letter us to Wortho's with thy breath.

KING HENRY VI:
Thyself, for this curses the fillence have cursed
```

Figure 8: Model output when trained on TinyShakespeare dataset.

4.1.2 Test Case 2 - Robinson Crusoe

For second evaluation dataset, I choose the novel "Robinson Crusoe." Comparing to previous dataset, this one is a novel rather than drama, written in english from different period of time. For training the model I choose the same hyperparameters as shown in Table 1. Figure 10 shows the train and validation loss from the model training process. Since the model architecture was the same, it took model similar time to conclude the training process.

```
### Model Prediction ###


me well; you may but second civilize."

Upon this our greatest refrence, like together, into any occast,
inward, that was awakening, and to sleep, beyond all the secret six
pounds which came to my master. I wished my little money with myself, and
there great lieup, a kencease, for thee, swlepy-con, cakin, and made
my great erprotion, but half-plown along our bullets up to his servantance,
I was guilty to see it; and so that these fire assured me more
in my head.

Friday was so much scere, as I n
```

Figure 9: Model output when trained on Robinson Crusoe dataset.

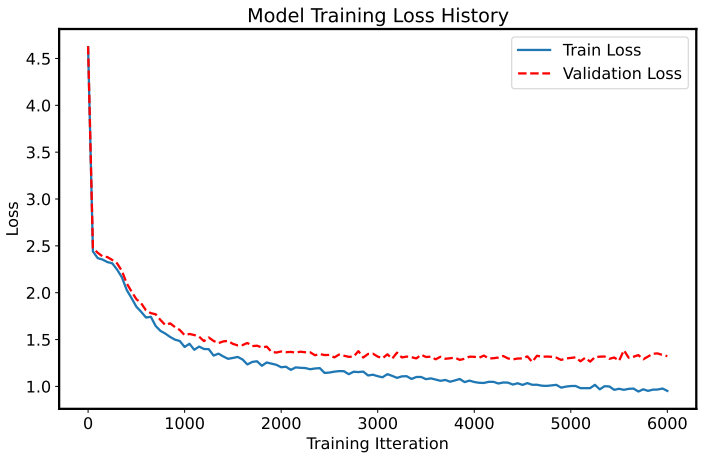

Figure 10: Train and validation loss histories for training auto-regressive transformer model on Robinson Crusoe dataset.

As we can see in the Figure 9, output generated by the model is a reasonable text similar to the original novel. This proves that our auto-regressive model can generate the text on any given input, no matter what language or structure of the text is. Model train and validation losses both converged, which suggests that the training process was successful.

4.1.3 Test Case 3 - Forest Gump

Final dataset on which I will evaluate the model will be the script from the "Forest Gump" movie. It was written in Modern English; it challenge, however, is its length that is about 10 times smaller than the previous two datasets. When training using the same parameters as used in previous two cases, our model is heavily over parametrized, which causes significant overfitting. Figure 11 shows two loss curves that clearly diverged from one another, which suggests that the model overfitted.

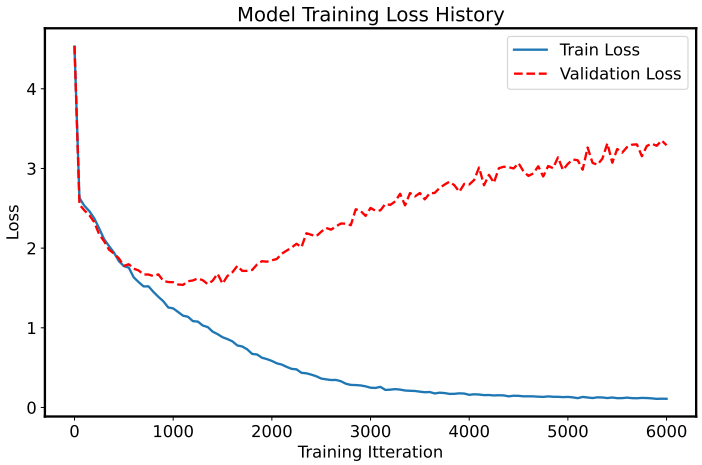

Figure 11: Train and validation loss histories for training auto-regressive transformer model on Forest Gump dataset. We can see that loss curves diverged from one another.

To fix this issue, we need to redesign the model by changing some of the hyperparameters, effectively making the network smaller, this will prevent overfitting and actually allow the model to learn from the data. Update parameters used in the second training attempt are shown in the Table 2.

As we can see in the Figure 12, train and validation losses did not diverge anymore after shrinking the model size. This means that we were able to properly train the model on the new dataset. This shows how important it is to make sure that the training process of the auto-regressive transformer is adjusted to the dataset that we are working on. Changing this parameters effectively made the

Model Parameter	Value
Training Iterations	6,000
Learning Rate	3e-4
Model Parameters	2,006,454
Encoding Type	Char
Transformer Blocks	6
Self-Attention Heads	6
Embedding Dimension	164
Batch Size	64
Context Block Length	256
Feed-Forward NN Layer Dimension	656
Dropout Rate	0.3
Training Time	5:54 min

Table 2: Model parameters for auto-regressive transformer training on Forest Gump dataset with character-level tokenization.

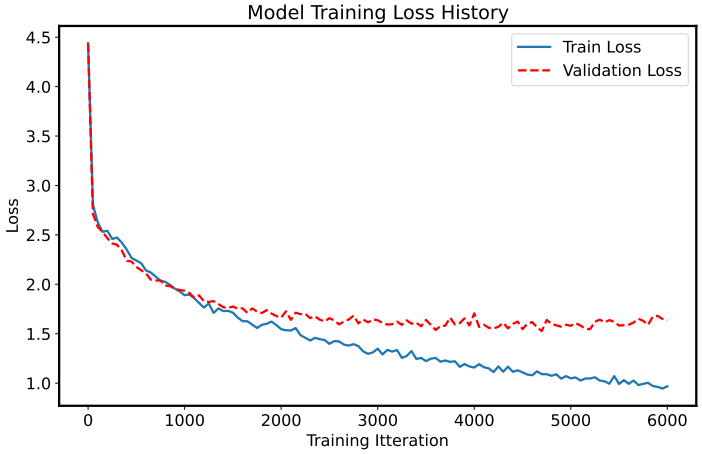

Figure 12: Train and validation loss histories for training auto-regressive transformer model on Forest Gump dataset with updated hyperparameters. We can see that loss curves did not diverge from one another anymore.

```
### Model Prediction ###


for Shre's it was scrory.

LT. DAN
Happy New York, 'chepted, you coming
fried a happrethest wounded soldier.

Lt. Dan, guit out the criplegaing?

AThe captanch unical survach ontor the ground. Lt. Dan looks at the
othe Nerkader "Gump," over the call by of
she whococolate try to the raduse. Forrest young
Welly gid in from
his new.

LT. DAN
I hard this is out! Yeah, dem now you were,
get I was ain't meenj.
What am? An, Mrill rimpomate in a ma.

Forrest come of the legs on the grazing cearr.
```

Figure 13: Model output when trained on Forest Gump dataset.

network about 6 times smaller in terms of model parameters. Choosing the right parameters might be difficult and require a lot of tuning. When model is shrinked too much, it will not train at all, so

we need to be careful adjusting the parameters. Example of model generated output based on Forest Gump dataset is shown in Figure 13.

4.2 Prediction Starting from a Fixed Prompt

Auto-regressive predictive transformer models, like ChatGPT, are being used to predict future sequences of token given the context, which is user prompt. Similar, but less impressive, result can be achieved using the model developed in this framework.

In this section, instead of starting the model with an empty context - new line, I will initialize and encode context to force the model to generate future predictions in the provided context.

I will evaluate this using the same trained model as in section 5.1.1, which is a transformer pre-trained on TinyShakespeare. I provide model with a prompt "ROMEO," which is the name of one of the characters in Shakespeare's drama. I expect generated text to feature another character from the same drama, which is "Romeo and Juliet."

```
### Model Prediction ###

ROMEO:
A noble fearing, 'tis play let thee the way.
This saying is too late to the complace earth.

DUKE OF YORK:
Speak not.

DUCHESS OF YORK:
Stay, not when me not beasts, I mean thee:
His mother pity hath the sent that while,
Or he their successed 'gainst me with thy slave.'

DUCHESS OF YORK:
Ah, mock, my peace, and ere the present plain,
Do I with him right out to us so prevent me:
I was a diaJu welcome.

MONTAGUE:
I sin't doth another,--

JULIET:
If that adverse they live should prove in hope
```

Figure 14: Model output when starting with a fixed context prompt.

As we can see in the model output, after forcing the model to start with the fixed context, the model generated output that is similar to one of the original author's texts. This means that model can accurately understand the context information and generate text based on that.

Although our model was successfully able to predict next tokens using given context, there are additional steps to create a full GPT chat bot, like ChatGPT. This is called the Fine-Tuning stage. The process adjusts the model to generate text that satisfies provided prompts by applying a custom reward function. The overview of this process will be provided further in the report.

4.3 Comparison of Different Encoding Methods

In the Methodology section, I evaluated two encoding methods that I will evaluate in the model: character-level encoding and GPT-2 encoding. In this section, I will compare the performance of the two methods based on the base line model trained on the TinyShakespeare dataset.

Since GPT-2 encoding is tokenizing text on word and character level, we can improve the network by decreasing the context block size for each batch and making the whole network larger. This comes from the fact that the model will not need to analyze the long sequences of characters, but will focus on sub-word tokens, which is more effective and efficient approach. Model hyperparameters chosen for the training are shown in the Table 3.

Using GPT-2 encoding, I was able to train a larger auto-regressive transformer model within shorter amount of time, due to smaller batch of data that was necessary for the training. More complex encoding allow us to analyze larger datasets without the need to break them down to character level. Figure 15 shows the loss history of the trained model. As we can see, both loss curves converged to a similar value. Although loss values are not as low as these for the character level tokenization,

Model Parameter	Value
Training Iterations	6,000
Learning Rate	3e-4
Model Parameters	11,168,97
Encoding Type	GPT-2
Transformer Blocks	10
Self-Attention Heads	10
Embedding Dimension	96
Batch Size	64
Context Block Length	64
Feed-Forward NN Layer Dimension	576
Dropout Rate	0.3
Training Time	12:08 min

Table 3: Model parameters for auto-regressive transformer training on TinyShakespeare dataset with GPT-2 encoding.

they cannot be directly compared as loss itself will be evaluated differently with different encoding methods. I also attached a plot of relative improvement, shown in Figure 17, comparing two models, which shows that it is similar for both models. Although character-level tokenization results may seem better comparing to GPT-2 encoding, it is a result of different model architecture, which makes it hard to compare methods directly based on losses.

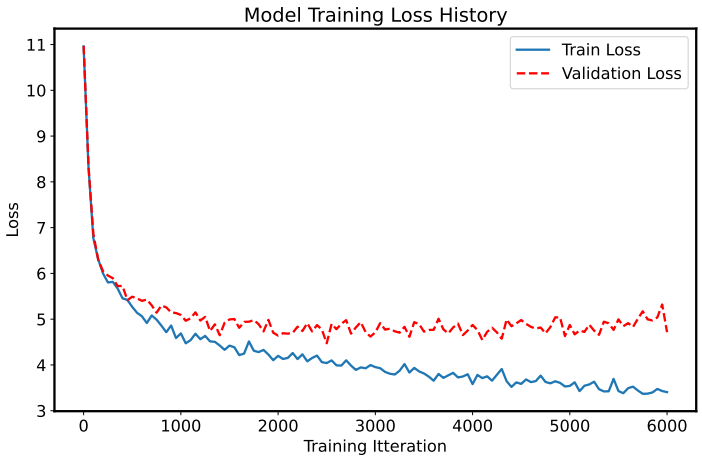

Figure 15: Training loss history for auto-regressive transformer model with GPT-2 encoding.

```
### Model Prediction ###

LEONTES:
It in ask the Duke of stand
Your honour! Would Great my Creator,
The law we do bring of yew clear us, not uprightam:
The tribunes are down, she hangs together,
That, sow'd my care my friend old acts.
Bestians on, a command!

JOHN OF GAUNT:
'This remember, for me too rough, and like a while;
And at God and on victory was nymphaged dangerous.

CLIFFORD:
So easily heard of the limits Richard slain!

TYRREL:
King son, loving of his friend's unjust depending are;
He's lightness, hoping, the blest, and all night
The disguised of worldhire rhybering clouds
```

Figure 16: Model generated output using GPT-2 encoding.

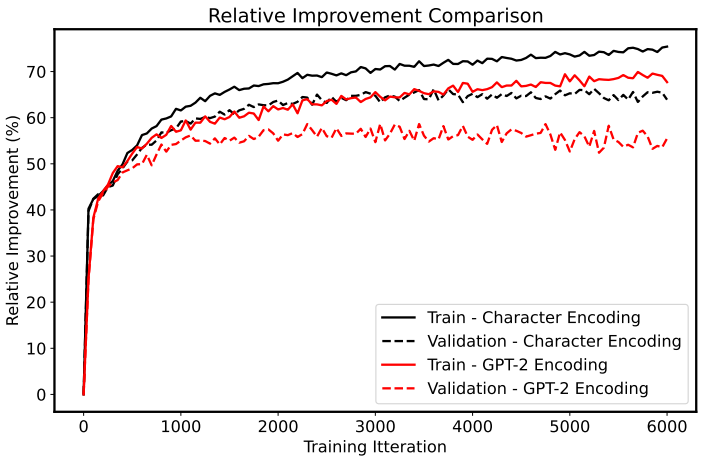

Figure 17: Relative improvement for character-level and GPT-2 encoding methods. Plot compares the value of the loss to its initial value, where 100% is achieving 0 loss.

Using more complex encoding methods, like GPT-2, is important in creating Large Language Models, like ChatGPT. They allow to create large and complex models with millions of parameters, while keeping the model computationally feasible. Also, model does not focus on recreating individual words using letter, but sentences using chunks of words. This makes the Byte Pair Encoding incomparably more effective in most applications.

4.4 Self-attention vs Multi-Head Attention

To further evaluate the architecture of the model, I will train the exact same model as in previous cases, but only using Single-Head Self Attention. This forces the model to develop a single head attention, which is designed to capture all information about affinities between different tokens in the sequence.

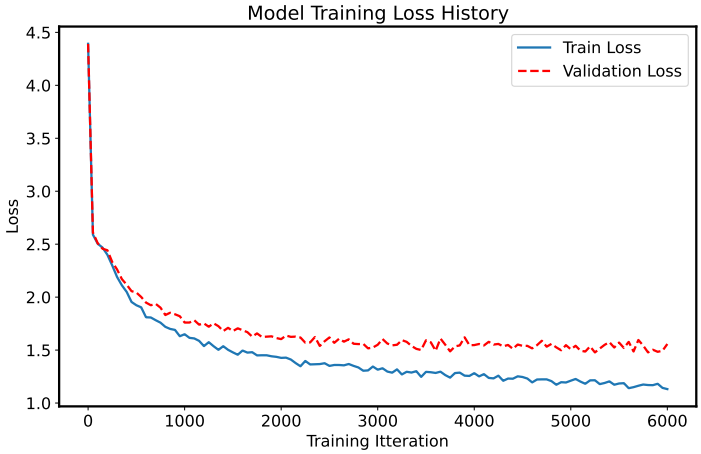

Figure 18: Model loss history when trained with Single-Head Self-Attention.

As shown in Figure 18, loss history for both validation and train datasets is very similar to the baseline model. This is caused by the fact that our training dataset is very small and uniform, which makes it possible for single head attention to compute proper affinities between tokens. Applying Multi-Head Attention, however, is very important in complex models, where capturing context from multiple perspectives is crucial for model performance.

4.5 Training the Model without LayerNorm and Residual Connections

In my framework, I mentioned to improvements that makes the network stable: LayerNorm and Residual Connections. Purpose of both is to stabilize the training process and prevent gradient diminishing throughout back propagation. In this section, I will attempt to train the baseline model on TinyShakespeare dataset without LayerNorm, and then without Residual Connections in the transformer block. The purpose of that is to examine the effect of these two stabilization techniques on the model training performance.

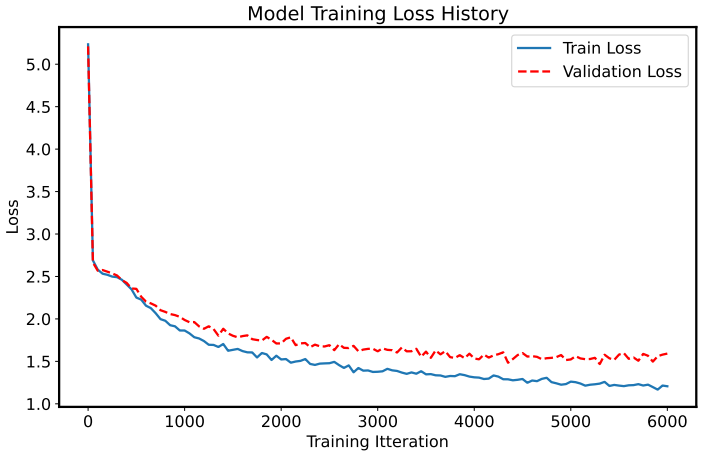

Figure 19: Model loss history when trained without LayerNorm.

As shown in the Figure 19, the model still converged properly without adding the LayerNorm its performance, however, is slightly worst comparing to baseline model (both losses are about 0.1 higher comparing to base model). Adding LayerNorm in this example is not crucial for training this network, but still stabilizes it and ensures proper convergence, while preventing incorrect convergence of softmax probabilities. LayerNorm effect is especially important when training larger Deep Neural Networks.

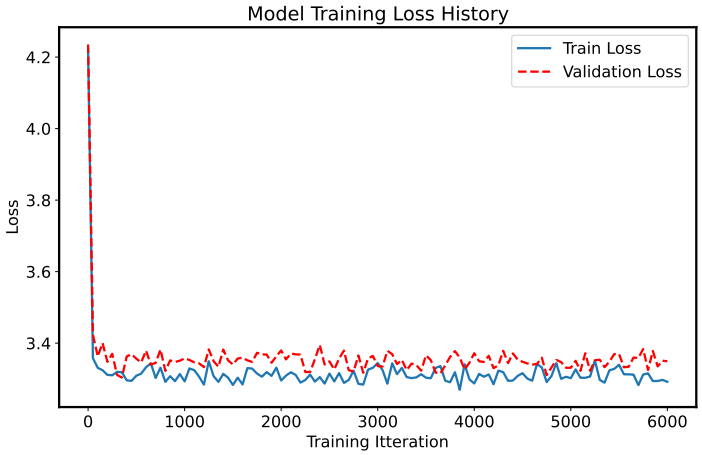

Figure 20: Model loss history when trained without Residual Connections.

Figure 20 shows loss history for the model trained without residual connections inside the transformer block. As shown in the Figure, model failed to converge and optimization failed due to gradient diminishing in Deep Neural Network during back propagation. This shows importance of creating these skip connections - it is a simple method that does not add any parameters to the model but heavily stabilizes the training process, ensuring that model will be able to generate meaningful output.

4.6 Possible Grammar Improvements to Large Language Models

Although most of the generated text from different datasets shown in this Evaluation is making sense in the context of the input, we can notice numerous grammar mistakes on the way. This is due to a probabilistic nature of the model. We can try to resolve this by making the prediction model more robust, or introducing additional loss terms that are penalizing incorrect outputs.

Another way of fixing this problem is to implement a language tool, which can correct the grammar mistakes for us. Example of such package in python is language-tool. It can be implemented onto generated text to correct any grammar mistakes that were observed in the generated output. This is a simple and effective way of improving model generation performance. Below I show an example of implementing the tool on a line of Forest Gump like text generated during the evaluation.

Original text: AThe captanch unical survach ontor the ground.

Corrected text: The captaincy unital surface onto the ground.

As shown above, the language tool can be used to improve the grammar errors from the auto-regressive generator. This way we can fix errors in the output without the need to add complexity to our models. Similar tools are implemented in most of modern Large Language Models.

4.7 Issues with Scaling Up the Model

In order to create effective auto-regressive transformer model, we need to make sure it is of a right scale. Scaling up the model is challenging process, as we are balancing between under parametrized models, when the NN is not able to catch features in the data, and over parameterized model, where NN is over fitting. In this work, I encountered this challenge while adjusting the model to work for Forest Gump dataset. Adjusting proper hyperparameters needs to be carefully done before the training process, accounting for unique features that dataset has.

Another challenge with scaling up Deep Neural Networks is computational expense of training the networks. Baseline model run in this work is having over 14 millions parameters - processing this over 6,000 iterations requires a large GPU, which is expensive. Therefore, training the model itself might be a challenge, as obtaining reliable computational resources for model evaluating might be beyond scope of a project.

4.8 Project Code

Code used in the project was based on the GitHub repository created by Andrej Karpathy as part of tutorial linked by Professor in the example projects for the class - "Let's build GPT: from scratch, in code, spelled out." Code I used in this project is my version of the example code that I rewrote for the purpose of this project. It is still the same structure and logic as the one explained by the author of the tutorial.

Code used for the project, along with all train datasets, was uploaded to the GitHub repository attached below. https://github.com/mjaszczukpenn/ENM5310-Final-Project—Mateusz-Jaszczuk

4.9 Summary of Results

To summarize the results, I was able to train the auto-regressive transformer model and evaluate it in three different datasets. Since the transformer was decoder-only, I recreated a model that could complete text given any starting prompt.

I also tested different variations of transformer architecture, removing LayerNorm and Residual Connections. After experimenting with the framework, I learned that large Deep Neural Networks are very unstable in their nature - we need to be really careful to adjust model hyperparameters properly and to prevent gradient diminishing in the training process.

Most important lesson learned from the evaluation and testing was understanding all key components of the transformer network and learning ability to apply the model to real-life datasets.

5 Advanced Features and Applications of Transformers

5.1 Fine Tuning Stage of LLMs

In order to develop a Large Language Model Chat Bot, similar to ChatGPT, we need to perform fine tuning stage of the LLM. In this work, I focused on pre-training stage of the transformer, which can be interpreted as teaching model how to speak. It cannot, however, answer the questions yet - it can only generate text by completing next tokens in the given context.

Pre-training stage that was developed in this report is of a similar structure to the one demonstrated in GPT-3 technical report by OpenAI [9]. They key difference in model architecture is the model size. Our transformer has a total of 14 million parameters, while GPT-3 model extends to 175 billions. Beyond that, pre-trained transformer by OpenAI is trained on 300 billion tokens, which is way beyond the training we performed in this project. Nevertheless, model architecture of the auto-regressive transformer is the same and follows the structure proposed by [1].

Key difference that makes ChatGPT an excellent chatbot is fine tuning stage of the model. During the process, we collect demonstrations for the model, where we show example of questions with expected answers. Then, we train another model with a custom reward function, that quantifies how well model was able to address different prompts. Finally, we utilize reinforcement learning techniques to optimize the model to fit the desired policy.

Fine tuning stage of auto-regressive model is more difficult than creating pre-trained transformer itself. It needs to be tailored to desired policy and may require a lot of attention in tuning. In robotics applications, however, pre-trained transformer which can complete sequences of predicted user inputs to make the robot aware of potential task forces might be sufficient - model will provide robot with distribution of probabilities of next task forces, which would be used for model Inverse Dynamics adjustments.

5.2 LLaMa Encoding

Another interesting improvement to Large Language Models was introduced by MetaAI ad is called Large Language Model Meta AI (LLaMa). This model uses different type of token embedding, which is called Rotary positional embeddings (RoPE) [10].

For tokenization, LLaMa model utilizes similar Byte Pair Encoding as the GPT-2 model - it converts text into tokens based on sub word pieces. Key difference is application of Rotary Position Embedding Comparing to Absolute Position Embedding used by the GPT-2 model.

Rotary Position Embedding applies the rotation to token embedding based on their position is sequence. This approach allows the model to capture relative positions of tokens in sequence, rather their absolute positions. This approach is particularly effective in analyzing long sequences of data, where we don't want to bias the prediction. This embedding method is probably most robust embedding method currently used in Large Language Models and Natural Language Processing. Unfortunately, due to other class projects, I was not able to implement this method into my model. However, I plan to try to explore it further in the future research - I think that using Rotary Embedding method can be useful in robotics when analyzing long sequences of motions, making sure that the model can capture relative dependencies between tokens, rather than focusing on absolute position in the batch.

5.3 Encoder-Decoder Transformers

An aspect of transformer that I mentioned in the Methodology section was that model I developed is a decoder-only transformer. This means, that model is trained to be auto-regressive with lower-triangular masking - it has no ability to communicate with future tokens. It makes it ideal use for Large Language Models. If we removed the masking, we would have encoder transformer, which would allow the model to analyze the full context - tokens both before and after the token of interest.

Encoder-decoder architecture is another sort of transformer, commonly used in machine translation. This architecture is used when the desired output is of a different format then input. We can then encode the input and apply cross-attention block, which will allow the model to translate between two different formats. This architecture of encoder-decoder transformer can be used to for example, translate sentences from english to another language, or to translate machine actions to text output.

Original landmark paper [1] implemented cross-attention block, as it focused on the applications in machine translation.

5.4 Possible Implementations of Auto-regressive Models in the Robotics Field

One of the biggest challenge in robot to human interactions is robot's ability to understand human's intentions. Robot's adaptive controller cannot simply adapt to varying Inverse Dynamics of the system, as it can be an effect of changing robot's inertia, or applying task force. These two factors needs to be distinguished in order to allow safe interactions.

One approach h is to apply perception system to the robot that will make it aware of human's actions. I think this approach can be extended by the perception system input being feed to the transformer (decoder only, or encoder-decoder, depending on perception system input definition). Using that approach, robot would be able to predict next time step possible task force probabilities. Note that these can be kept as distribution of probabilities, as they will make robot aware of possible task forces exerted by a person on a manipulator. Using this prior information, when robot will detect an disturbance, it by dynamics exploration it would be able to understand if this comes from applying extra mass or torque to the robot (external disturbance) or from human interaction. Using this information, robot should be able to adapt and safely co manipulate different objects with humans.

This application of auto-regressive model is something I would like to apply to my robotics research and was my main motivation on pursuing this topic of Final Project for the class.

6 Conclusion

To summarize, auto-regressive transformer models are powerful tool that can be utilized to predict sequences of tokens based on the previous train data. Models are very sensitive to choice of hyperparameters and require adding LayerNorm and Residual Connections to stabilize the training process. For all test cases, my model was able to predict next tokens for given training text and context, which were all making sense. Model can be further tuned and modified to increase the performance; however, output is satisfactory to prove that the model works well.

My main takeaway message from the project is how well auto-regressive transformers can predict output probabilities in the sequence of generated tokens. I believe this can be applied in the robotics problems, were humans try to co manipulate objects with robots. Transformer models will allow robots to anticipate human's intentions, which will result in both better and safer control performance.

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[2] OpenAI. *ChatGPT (December 18 version)*. `https://openai.com/chatgpt`. Accessed: 2024-12-18. 2024.

[3] Himanshu Gaurav Singh, Antonio Loquercio, Carmelo Sferrazza, Jane Wu, Haozhi Qi, Pieter Abbeel, and Jitendra Malik. "Hand-Object Interaction Pretraining from Videos". In: *arXiv preprint arXiv:2024.09* (Sept. 2024).

[4] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL 2016)*. 2016, pp. 1715–1725. DOI: `10.18653/v1/P16-1162`. URL: `https://arxiv.org/abs/1508.07909`.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[6] 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 (CVPR)*. 2016, pp. 770–778.

[7] Diederik P Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations (ICLR)* (2015). URL: `https://arxiv.org/abs/1412.6980`.

[8] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A simple way to prevent neural networks from overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958.

[9] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *arXiv preprint arXiv:2005.14165* (2020). arXiv: `2005.14165 [cs.CL]`.

[10] Juncheng Su, Yu Lu, Shengfeng Pan, Liang Wen, and Yunfeng Liu. "RoFormer: Enhanced Transformer with Rotary Position Embedding". In: *arXiv preprint arXiv:2104.09864* (2021).