Abstract

Examining text generation with transformers

Samvit Jatia

2023

This paper provides comprehensive background on the transformer architecture for sequence modeling and transduction tasks. We elaborate the overall model comprising encoder and decoder stacks, with a focus on the decoder. Key components including multi-headed self-attention and feed forward layers are explained. We describe scaled dot-product attention enabling every sequence position to relate to every other for global context modeling. Multi-headed attention provides multiple representation subspaces for richer relationships. Positional encodings inject order information into the parallel computation. Additional elements like residual connections and layer normalization are also detailed. These innovations provide transformers with greater ability to capture long-range dependencies critical for language understanding, without reliance on recurrence. Transformers have achieved state-of-the-art results over predecessors on tasks with complex context relationships. The paper offers an in-depth reference on the transformers and constituent mechanisms powering their rise as a general advance for sequential data modeling across language, vision and beyond.

Examining text generation with transformers

A Capstone Project
Presented to the Faculty of Computer Science
of
Ashoka University
in partial fulfillment of the requirements for the Degree of
Postgraduate Diploma in Advanced Studies and Research

by
Samvit Jatia

Advisor: Debayan Gupta

December, 2023

Contents

List of Figures **iv**

1. Introduction **1**

2. Background **3**

2.1. RNN vs CNN vs Transformer 3

2.2. Transformer Model 6

2.2.1. High Level Overview 6

2.2.2. Self Attention 8

2.2.3. Multi-Head Attention 11

2.2.4. Positional Encoding 12

2.3. Experimental Setups 15

2.3.1. Training Data 15

2.3.2. Hyperparameters 16

2.3.3. Evaluation Metric 17

2.3.4. Experiment Result 18

3. Conclusion **21**

Bibliography **22**

List of Figures

2.1. The control flow of a standard RNN 4
2.2. The control flow of a standard CNN 5
2.3. Architecture of different neural network in a NMT 6
2.4. The transformer model 7
2.5. Intuition behind Self Attention 8
2.6. Scaled Product Attention 10
2.7. Multi-Head Attention 11
2.8. Multi-Head Attention Flowchart 12
2.9. Positional Encoding 13
2.10. Variable Loss 20

Chapter 1

Introduction

Recurrent neural networks, long short-term memory and gated recurrent neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and transduction problems such as language modeling and machine translation. Numerous efforts have since continued to push the boundaries of recurrent language models and encoder-decoder architectures. Recurrent models typically factor computation along the symbol positions of the input and output sequences. Aligning the positions to steps in computation time, they generate a sequence of hidden states h_t, as a function of the previous hidden state h_{t-1} and the input for position t. This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples. Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences. In all but a few cases, however, such attention mechanisms are used in conjunction with a recurrent network. The Transformer proposes a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output. The Transformer allows for

significantly more parallelization and can reach a new state of the art in translation and text generation quality.

Chapter 2

Background

2.1 RNN vs CNN vs Transformer

Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and Transformers represent pivotal architectures in the field of deep learning. RNNs excel in sequential data processing, CNNs in spatial pattern recognition, and Transformers, through attention mechanisms, have demonstrated superior performance in capturing long-range dependencies. The transformative capabilities of Transformers make them particularly adept at numerous tasks, outshining both RNNs and CNNs in certain domains. To better understand how the transformer differentiates itself, this part of the paper will compare all three models and highlight key elements solved by the implementation of the transformer.

RNN

RNNs are stateful networks that change as new inputs are fed to them, and each state has a direct connection only to the previous state. Recurrent Neural Network (RNN) is designed to make use of sequential information. The output y_t at time

step t depends not only on its present input x_t but also the entire history of inputs $x_0, x_1, \ldots, x_{t-1}$ from the previous moments. In order to remember the past, RNN introduces hidden states to act as the memory of the network. The hidden state h_t at time step t captures information from all previous time steps. It is calculated based on the input at the current time step x_t and the previously hidden state h_{t-1} through a single tanh layer as the activation function.

However, RNN is difficult to solve long-term dependence in practice, which means the information slowly disappears as the number of layers in the neural network increases. Using an LSTM is a possible solution; however, it still has limitations since its output is mostly based on previous states.

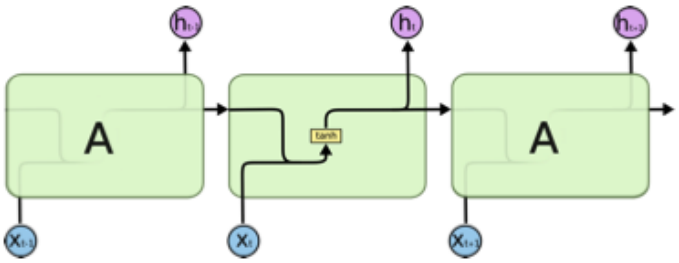

Figure 2.1: The control flow of a standard RNN

CNN

Convolutional Neural Networks (CNNs) are specialized neural networks designed for processing grid-structured data, particularly images. Unlike RNNs, which excel at capturing sequential dependencies, CNNs are tailored to recognize local patterns in spatially organized information. The core architecture includes convolutional layers, where filters traverse the input grid, identifying specific features in different regions. Pooling layers subsequently reduce spatial dimensions, maintaining critical information while enhancing computational efficiency. Fully connected layers follow, linking

neurons across different layers for final classification or regressions.

However similar to RNNs they face challenges in capturing long-term dependencies in sequences, primarily because their architecture isn't optimized for these tasks due their fixed receptive fields and sequential structure.

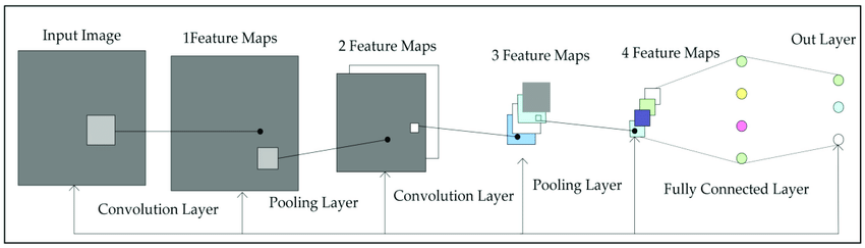

Figure 2.2: The control flow of a standard CNN

Transformer

Transformers rely heavily on self-attention networks. Each token is connected to any other token in the same sentence directly via self-attention. Moreover, Transformers feature attention networks with multiple attention heads. Multi-head attention is more fine-grained, compared to conventional 1-head attention mechanisms. Any two tokens are connected directly: the path length between the first and the fifth tokens is 1. Similar to CNNs, positional information is also preserved in positional embeddings.

Unlike CNNs and RNNs transformers capture long-term dependencies well primarily due to their innovative architecture, specifically the self-attention mechanism. The self-attention mechanism in transformers allows each element in the sequence to attend to all other elements, assigning different attention weights to them. This enables the model to weigh the importance of each element in relation to others, facilitating

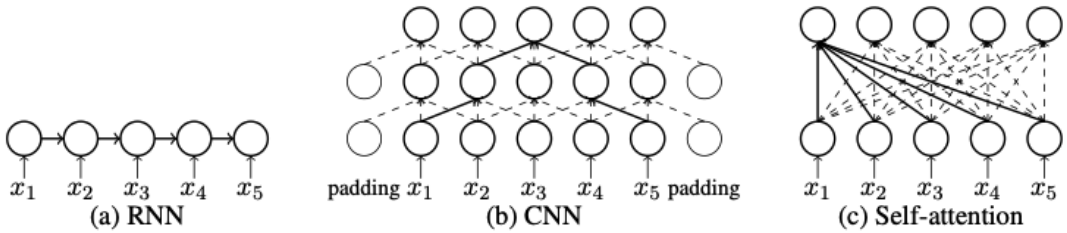

Figure 2.3: Architecture of different neural network in a NMT

the capture of long-range dependencies.

Long-term dependencies enable models to understand the context of a sequence over extended periods. This is crucial in tasks where the meaning or interpretation of a particular element depends on the context of preceding elements.

2.2 Transformer Model

2.2.1 High Level Overview

The transformer model consists of an input layer, an encoder, a decoder and an output layer At a high level, the model encoder processes an input sequence of L tokens into a L $\times$ d dense representation, progressively re-encoding each token using the context of all the sequence. Given this representation, a decoder auto-regressively generates a new sequence, one token at a time. For the course of this project, we will be focusing only on the decoder aspect of the Transformer as we are performing sequence generation rather than seq-2-seq translation. The decoder consists of a stack of 6 identical layers that are each composed of three sublayers:

1. The first sublayer receives the previous output of the decoder stack (or a fresh input), augments it with positional information, and implements multi-head

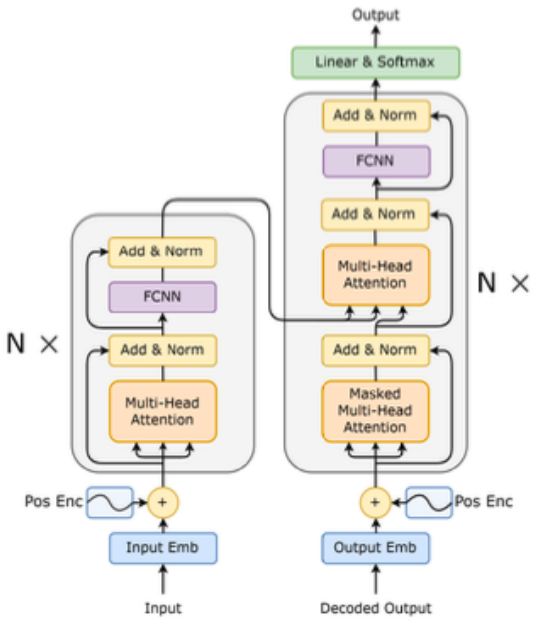

Figure 2.4: The transformer model

self-attention over it.

2. The second layer implements a multi-head self-attention mechanism with h number of heads that receive a (different) linearly projected version of the queries, keys, and values, each to produce h outputs in parallel that are then used to generate a final result. This multi-head mechanism receives the queries from the previous decoder sublayer and the keys and values from the output of the encoder or from an independent input vector. This allows the decoder to attend to all the words in the input sequence.

3. The third layer implements a fully connected feed-forward network consisting of two linear transformations with Rectified Linear Unit (ReLU) activation in between.

The six layers of the Transformer decoder apply the same linear transformations to all the words in the input sequence, but each layer employs different weight (W)

and bias (b) parameters to do so.

Furthermore, each of these two sublayers has a residual connection around it.

Each sublayer is also succeeded by a normalization layer, layernorm, which normalizes the sum computed between the sublayer input, x, and the output generated by the sublayer itself, sublayer(x):

$$\text{layernorm}(x + \text{sublayer}(x))$$

An important consideration to keep in mind is that the Transformer architecture cannot inherently capture any information about the relative positions of the words in the sequence since it does not make use of recurrence. This information has to be injected by introducing positional encodings to the input embeddings.

2.2.2 Self Attention

Self-attention is an attention mechanism which has the ability to represent relationships between words in a sentence.

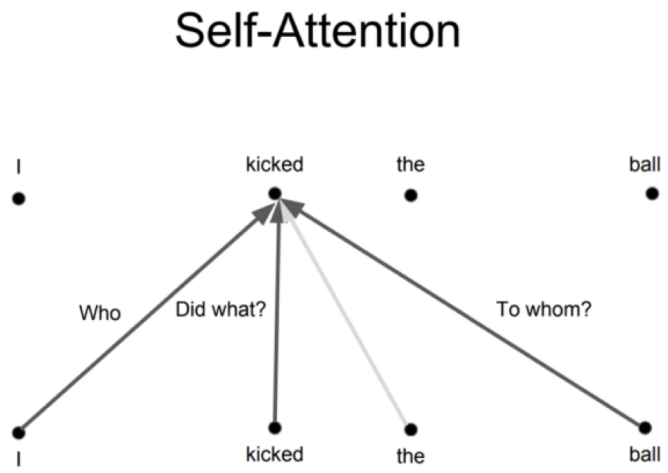

Figure 2.5: Intuition behind Self Attention

Figure-(5) shows an example of self-attention mechanism. The model is able to look at the other words in the input sequence - "I", "kicked", "ball" - to get a better understanding of the certain word "kicked" by answering three questions - "Who", "Did what", "To Whom" - respectively. The attention mechanism used by Vaswani et al. (2017) is dot-product attention, which can be described by the following equation.

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

In a self-attention layer, its input vector is transformed into a key (K), query (Q) and value (V). All the tokens in the input vector parellely and independently produce a key and a Query. K and Q are linear modules of the dimension (Batch, Time, headsize).

At first, the score (S) is calculated to determine the amount of focus to place on the other words in a sequence while encoding the current word. This score is calculated using the dot product between the query and key vectors ($S = Q \cdot K^T$). It is normalized ($S = S/\sqrt{d_k}$) to stabilize the gradients.

Unlike the encoder, the decoder is designed in such a way that it attends to only preceding words. Hence, the prediction for a word at position i can only depend on the known outputs for the words that come before it in the sequence. In the attention mechanism, we achieve this by introducing a mask over the values produced by matrix S. This masking is implemented by suppressing the matrix values that would otherwise correspond to illegal connections.

$$\text{mask}(QK^T) = \text{mask}\left(\begin{bmatrix} e_{11} & e_{12} & \dots & e_{1n} \\ e_{21} & e_{22} & \dots & e_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m1} & e_{m2} & \dots & e_{mn} \end{bmatrix}\right) = \begin{bmatrix} e_{11} & -\infty & \dots & -\infty \\ e_{21} & e_{22} & \dots & -\infty \\ \vdots & \vdots & \ddots & \vdots \\ e_{m1} & e_{m2} & \dots & e_{mn} \end{bmatrix}.$$

Following gradient stabalisation, the softmax function takes the scores and transforms them into probabilities. It exponentiates each score, making larger scores more significant, and then normalizes them so they add up to 1. This way, we get a set of values representing the relevance of the current word to others, where higher values mean more attention.

Each word has an associated value vector V. The attention scores act as weights, determining how much emphasis each word should receive. Multiplying the scores by V results in a weighted sum, emphasizing words with higher scores and de-emphasizing those with lower scores. This step ensures that important words contribute more to the final representation, while less relevant words have a smaller impact.

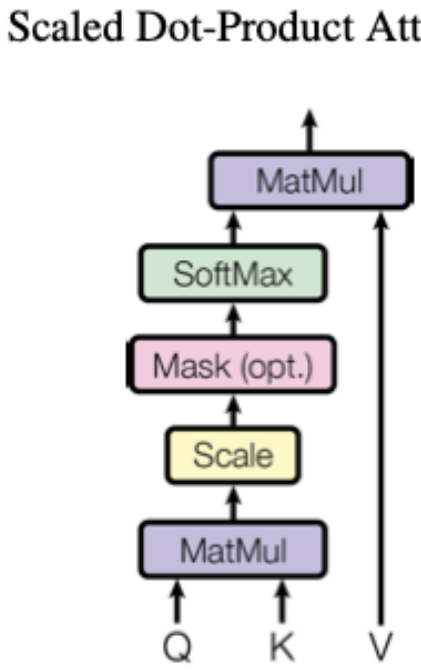

Figure 2.6: Scaled Product Attention

2.2.3 Multi-Head Attention

The self-attention mechanism proposed by Vaswani et al. [2017] has an additional feature called multi-head attention (MHA). It helps to improve the performance in two ways: by augmenting the network's ability to focus on multiple positions and by giving distinct representational subspaces to each word. For example, if there are eight heads, eight sets of K, Q and V matrices exist, each representing a unique representational subspace. They are concatenated before passing through the feedforward network.

Instead of a single attention weighted sum of the values, the Multi-Head Attention computes multiple attention weighted sums to capture various aspects of the input. Through simple splicing of multiple independent attentions, we can obtain information on different sub-spaces.

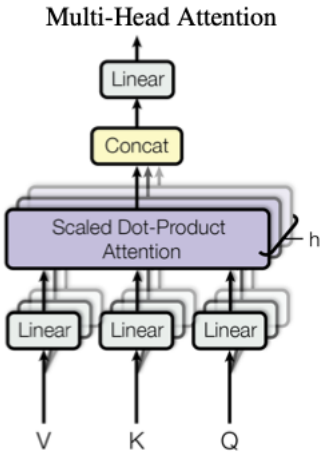

Figure 2.7: Multi-Head Attention

To learn diverse representations, each head is a unique linear transformation of the input representation as query, key, and value. Then the Scaled-Dot Attention is calculated h times in parallel, making it so-called Multi-Headed. Outputs are then concatenated. Finally, one single linear transformation is applied, as showed in Figure

8. Multi-Head Self Attention can learn related information from different representation sub-spaces because each of these query, key, value sets is randomly initialized.

The key characteristic of NLP tasks is the order of the words in a sequence. However, the operations we have discussed till now are permutation invariant. Positional embedding vectors are added to each input embedding vector to address this issue. These vectors help the model estimate the position of each word in a sequence in the projection space (i.e.,K/Q/V).

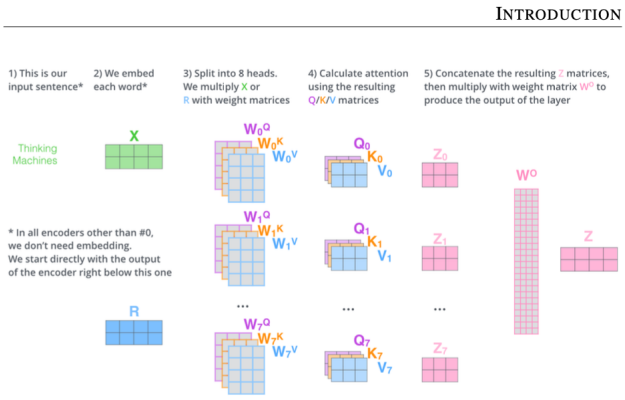

Figure 2.8: Multi-Head Attention Flowchart

2.2.4 Positional Encoding

One of the reasons why Transformer is so powerful today is because of its ability to compute in parallel and therefore, scale really well. When computing a sequential data in parallel, the relative positions within the sequence data is lost. This is why traditional Recursive Neural Networks are computed sequentially. However, Positional Encoding allows sequential data to be computed in parallel without losing the positional information.

Positional Encoding works like Word Embedding but for positions and with a fixed (not learned) function. Like word embedding, positional encoding takes in the position of the token and outputs an encoded embedding that represents the relative

position of the token within the sequence. This way, the individual tokens themself know their relative position even when extracted from the sequence.

The author suggests two possible forms of positional encoding: Learned and fixed. The paper uses an implementation of fixed positional encoding, however we go with former. Further we'll explain both forms of encoding and reasons as to why we used learned encoding.

In the original Transformer paper, the sinusoidal positional encoding scheme is defined as follows:

- Each position *pos* in the sequence is assigned a positional encoding vector.
- Each dimension PE_i of this vector is calculated using sinusoidal functions:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{(2i/d_{model})}}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{(2i/d_{model})}}\right)$$

- This results in each dimension having a different wavelength, forming a geometric progression, allowing the model to generalize to longer sequence lengths.

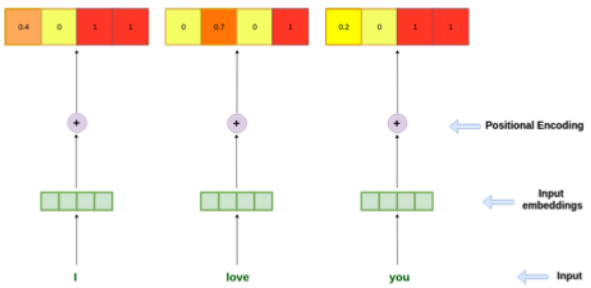

Figure 2.9: Positional Encoding

Learned position Embedding

Traditional neural networks like RNNs and LSTMs inherently understand sequence order because they process elements sequentially. However, models like Transformers process all elements of a sequence simultaneously and hence do not have a sense of order.

To remedy this, Transformers use positional encodings to give the model information about the position of each word in a sentence

Learned positional embeddings, Instead of using a predefined formula, the model learns the positional embeddings during training. These embeddings are initialized randomly and then adjusted through backpropagation, just like other learnable parameters in the model. This is done by creating two seperate embeddings:

1) Token Embeddings - We use an embedding layer to convert token indices to embeddings, which is standard in transformer models.

2) 2. Positional Embeddings: You're also using an embedding layer for positional encodings. This means the positional information is learned during training rather than being fixed (as in the sinusoidal approach).

We combine these two embeddings causing each token to contain both semantic and positional information.

We chose to use this form of positional encoding due to its Adaptability. Learned positional embeddings exhibit a dynamic nature, adept at capturing intricate patterns and dependencies within the training data. This adaptability proves crucial for tasks featuring varying sequence lengths or intricate positional relationships. In contrast, static embeddings, being fixed, lack adaptability and may result in suboptimal performance when faced with complex or data-dependent sequence patterns.

However this method does have a drawback. Learned positional embeddings introduce additional parameters, potentially elevating overall model complexity. Conversely, Static embeddings, unaltered during training, incur no additional parameter overhead, offering a benefit in scenarios where model size or computational resources

are constraints. This increased parameter overhead could be a consideration in situations prioritizing model size or computational efficiency.

2.3 Experimental Setups

2.3.1 Training Data

Introduced in the 2017 paper "Generating Wikipedia by Summarizing Long Sequences", the Tiny Shakespeare text dataset contains over 40,000 lines extracted and condensed from the legendary plays of William Shakespeare. Despite its modest size, Tiny Shakespeare packs considerable complexity. With a vocabulary of over 5,000 unique words arranged into sequences exhibiting intricate long-range dependencies, understanding Tiny Shakespeare thoroughly tests a model's language mastery.

The verbose vocabulary swimming in Shakespearean diction provides great diversity. Proper nouns are frequently mentioned with a distance between references, requiring awareness of prior context. Speech patterns intertwine nefariously as the bard interleaves dialogue between characters. All this occurs fluidly through Shakespeare's iconic iambic pentameter style.

With just over 15kb of textual data, Tiny Shakespeare offers enough complexity to sufficiently train miniature generative models while remaining simple enough to iterate on quickly. These dual properties establish Tiny Shakespeare's utility for rapidly prototyping and analyzing context-aware natural language systems.

Relevance for training transformers

The complex word patterns, references and speech changes present over extended sequences in Tiny Shakespeare naturally underscore the value of long-range modeling - precisely the strength of the Transformer architecture. Transformers eschew recur-

rence and convolution for direct connectivity between all input and output positions through self-attention. Tiny Shakespeare exercises this self-attention machinery to relate signals distant in semantic space.

Multi-headed self-attention grants transformers diversity in representations. Tiny Shakespeare's rich vocabulary spanning characters, eras and emotions offers plenty to capture. The interleaved dialogue provides cues for learning specialized attention heads. Properly tracking speech requires keen awareness of passage positions for the transformer decoder - enabled precisely by Tiny Shakespeare's positional intricacies combined with positional encodings.

In summary, Tiny Shakespeare provides the perfect blend of intricacy and concision to rigorously exercise a transformer's capabilities on core language challenges like contextual references, speech patterns and positional reasoning - making it incredibly well-suited for training and analyzing context-aware natural language models based on the transformer architecture.

2.3.2 Hyperparameters

Batch Size The number of independent sequences processed simultaneously during training - 16.

Block Size Maximum context length considered for predictions - 128.

Maximum Iterations Total number of training steps - 5000.

Evaluation Interval Frequency at which this model is evaluated during training - 500.

Learning Rate 3×10^{-4}.

Evaluation Iterations Number of iterations before the model is evaluated - 200.

Embedding Size Dimensionality of word embedding used in Transformer - 162.

Number of Attention Heads The number of parallel self-attention mechanisms in the Transformer model - 6.

Number of Layers 6.

Dropout The probability of dropping out neurons during training, preventing overfitting - 0.2.

The transformer model is highly sensitive to hyper-parameters. Hyper-parameters such as number of layers in decoder/encoder, word embedding size, feed-forward hidden size, number of attention heads... are tuned in the setting of each model.

2.3.3 Evaluation Metric

When a probability distribution has either very high or very low probabilities for certain events, it has low entropy as there is low uncertainty about what events may occur. In contrast, a uniform distribution with equal probabilities for all events has maximum entropy as there is maximum uncertainty in the outcomes.

Text generation models assign probabilities to predict the next token based on the context. We want the model's predicted distribution to match the actual token distribution in the dataset. If they match perfectly, the loss entropy will be minimized indicating high quality text generation.

Mathematically, for the model distribution P and true data distribution Q over the vocabulary tokens, the loss entropy is:

$$L(Q, P) = -\sum_{x \in Vocab} Q(x) \log P(x) \tag{2.1}$$

Here, x refers to tokens in vocabulary $Vocab$. $Q(x)$ is the probability of token x based on true distribution. $P(x)$ is model-assigned probability for x.

By minimizing L, we make the model likelihood P approach the ground truth

likelihood Q forcing the model to assign high probabilities to tokens that actually occur more frequently.

Loss entropy advantages over metrics like Perplexity:

1. Independent of vocabulary size
2. Accounts for model overconfidence on unlikely tokens
3. Penalizes models that poorly calibrate predictions

Tracking loss entropy during training and using it to compare model performance allows better judgment of text generation quality. Lower entropy correlates to better contextual understanding and language modelling.

2.3.4 Experiment Result

With the transformer model trained on the Tiny Shakespeare dataset, we performed text generation to gauge the quality and coherence produced in practice. As a canonical benchmark containing intricate language structures, reproduction of Shakespeare poses a formidable challenge.

In this section, we assess model performance by analyzing generated samples juxtaposed to the original corpus. Both quantitative metrics and qualitative aspects based on domain expertise and human evaluation are utilized for a comprehensive perspective.

We primarily report results using loss entropy instead of the more conventional perplexity measure. As discussed in Section 5.2, loss entropy has favorable properties like invariance to vocabulary size and the ability to capture overconfidence—thus aligning better with human judgment of generation quality. Lower entropy correlates to better language modeling.

The key research questions guiding the analysis are:

1. Can the model accurately capture the style, diction, and syntax distinctive of Shakespeare across generated passages?

2. How effectively can it reproduce coherent, extended snippets exhibiting clear narrative structure?

Here is a snippet of the input fed to the transformer and its generation.

Original BRUTUS: Come: Half all Cominius' honours are to Marcius. Though Marcius earned them not, and all his faults To Marcius shall be honours, though indeed In aught he merit not

Generative LUCENTIO: He is last afford: make him diseably to London, Take him great Hastings, boldness in his natic keeps, To oftragn lost me ready glust through the house. Why chose that I dares it be a Montague.

To answer the questions posed above, the text captures the style, diction, and syntax distinctive of Shakespeare across generated passages. However, The model struggles to reproduce coherent, extended snippets exhibiting a clear narrative structure. The generated text appears disjointed and lacks a clear storyline. We believe these errors come due to the low hyperparameters used. As and when we scale it up the texts start to become more and more coherent.

Variable loss We view a decreasing varible loss. In an ideal scenario, as the model learns and adjusts its parameters, the loss should steadily decrease, indicating improved alignment between the predicted and actual values. A decreasing variable loss is indicative of a more stable and converging learning process, where the model is effectively adapting to the training data. This trend is crucial for achieving reliable and consistent model performance, allowing practitioners to better interpret the optimization process and make informed decisions about the model's effectiveness and generalization to new, unseen data.

Figure 2.10: Variable Loss

Chapter 3

Conclusion

In conclusion, this thesis undertook an extensive exploration of the Transformer model, delving into its architecture, mechanisms, and advancements. While briefly comparing with traditional models like RNN and CNN, the focus remained on Transformer's unique attention mechanism and its ability to capture long-range dependencies, making it particularly well-suited for sequential tasks. The analysis highlighted the transformative impact of Transformers across various natural language processing applications. The experimentation phase, centered on text generation, provided practical insights into the model's creative potential. The results underscored the Transformer's proficiency in generating coherent and contextually relevant text, showcasing its prowess in creative applications.

Bibliography

1. Vaswani, A., et al. "Attention Is All You Need."

2. Truong Thinh Nguyen. Machine Translation with transformers. Institut fu r Maschinelle Sprachverarbeitung, 2019.

3. Xie, Z. (2018). "Neural Text Generation: A Practical Guide." Retrieved March 22, 2018, from

4. Yorsh Uladzislau. *The Study of Linear Self-Attention Mechanism in Transformer*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

5. Vaishnav, M. (2023). "Exploring the role of (self-)attention in cognitive and computer vision architecture." Doctoral thesis, Université Toulouse 3 Paul Sabatier

6. Ramachandran, P., Parmar, N., Vaswani, A., Bello, I., Levskaya, A., & Shlens, J. (Year). "Stand-Alone Self-Attention in Vision Models." Google Research, Brain Team