2023-04-07

Language Modeling Using Image Representations of Natural Language

Seong Eun Cho
Brigham Young University

Language Modeling Using Image Representations of Natural Language

Seong Eun Cho

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Tyler Jarvis, Chair
Nancy Fulda
Jared Whitehead

Department of Mathematics

Brigham Young University

ABSTRACT

Language Modeling Using Image Representations of Natural Language

Seong Eun Cho
Department of Mathematics, BYU
Master of Science

This thesis presents training of an end-to-end autoencoder model using the transformer [46], with an encoder that can encode sentences into fixed-length latent vectors and a decoder that can reconstruct the sentences using image representations. Encoding and decoding sentences to and from these image representations are central to the model design. This method allows new sentences to be generated by traversing the Euclidean space, which makes vector arithmetic possible using sentences. Machines excel in dealing with concrete numbers and calculations, but do not possess an innate infrastructure designed to help them understand abstract concepts like natural language. In order for a machine to process language, scaffolding must be provided wherein the abstract concept becomes concrete. The main objective of this research is to provide such scaffolding so that machines can process human language in an intuitive manner.

Keywords: machine learning, deep learning, natural language processing, language modeling, autoencoder, transformer, attention, Jacobian, matrix calculus

Acknowledgements

I first want to express my gratitude to my committee members, Dr. Fulda and Dr. Whitehead, for being supportive and encouraging of my work, and for always being ready to discuss the direction of my research. I would like to give special thanks to my advisor Dr. Jarvis, who was always willing to help and never gave up on me even when I was struggling. I would also like to give special thanks to my wife Hyejin, who has always believed in my ability to accomplish great things and has helped me to write this thesis. Without their support, I might not have been able to complete this degree. I would like to recognize my family whom, although not always helpful, found their own ways to motivate me. A big thanks to Taylor, who was my personal therapist whenever I needed, and Charles for remaining my best friend. I would also like to acknowledge Do for sharing effective tips on staying focused, and Johnny and Yejin for providing me with much needed socialization. Finally, I acknowledge Woojoo for giving me my daily dopamine and Duyu for his emotional support. I am lucky to be surrounded by such incredible individuals.

Contents

List of Figures

Chapter 1. Introduction

Language is one of the most defining traits of intelligence. The ability to communicate is often used to distinguish between intelligent and non-intelligent life, and sophisticated language is what sets humans apart from other species. Likewise, one of the hallmarks of intelligence in machines will be their ability to use language as we do.

Throughout the past decade, deep learning has revolutionized the field of machine learning and artificial intelligence. Not only has deep learning surpassed other machine learning methods on simple classification or prediction tasks, it has become the state-of-the-art model for many different tasks. Furthermore, deep learning has found success achieving tasks that were previously unthinkable. In computer vision, for example, Generative Adversarial Networks [14] have found a way to generate hyper realistic human faces. AlphaGo [42], a reinforcement learning model, was able to defeat a world champion in the game of Go.

Deep learning has also made significant progress in the field of Natural Language Processing. Previous conversational models relied heavily on hard coded output responses and recognizing input key words. Conversely, language models built on deep learning are not restricted by such limitations. Recently, large language models like GPT [34] have shown a near unlimited range in providing targeted, elaborate, human-like responses. ChatGPT [40], in particular, has gained popularity as people experienced firsthand the depth and breadth of its capabilities. Even so, there are many necessary improvements to be made before language models can be truly human-like. The pinnacle of artificial intelligence will be when machines can freely speak and communicate with us and with other machines.

1.1 Motivation

What does it mean for a machine to process language as humans do? Machines excel in dealing with concrete numbers and calculations. They do not, however, possess an innate infrastructure designed to help them understand abstract concepts like natural language. In order for a machine to process language, scaffolding must be provided wherein the abstract concept becomes concrete. Word2Vec [28], a word embedding algorithm showed that trained word embeddings can have vector-like properties. With these properties, similar words are clustered in the latent space and simple algebraic operations can be performed using the latent vectors. A famous example of this shows vec(`king`) - vec(`man`) + vec(`woman`) = vec(`queen`), which is an intuitive result for us. By embedding natural language into concrete vectors, we mimic the manner in which humans process language for machines.

Word embedding proved to be successful, but difficulties arise when the embedding is scaled to sentences. While the English vocabulary is finite, vocabulary combinations allow for infinite sentence possibilities. Within the field of Natural Language Processing, this is an ongoing area of research, and much has been done to address the challenge of embedding sentences. Many of the current methodologies will be presented in section 2.8.

The motivation for this research comes from a desire to create an image representation of natural language. Much of human language is used to describe objects of the real world. In a sense, language and images are different ways of describing the world. Another motivation is to encode language into latent variables with vector-like properties. Using the latent vectors, machines could develop a way to understand human language and also find ways to communicate. The goal of this research is to design an encoder that can encode sentences into latent vectors of such space and a decoder that can use these vectors to generate new sentences using image representations.

1.2 Approach

The current state-of-the-art transformer [46] encoder model encodes sentences while preserving the length of each sentence. One of the primary challenges focused on in this research is discovering a method to find an optimal method to transform the variable-length encodings of the transformer into a fixed-length vector while preserving all the necessary properties and information that it contains. Another challenge is that the transformer decoder relies on attention. A single fixed-length vector is not suitable to be used as an input for the transformer decoder's attention mechanism.

The methodology presented in this research is to design an end-to-end autoencoder model by simultaneously training an encoder and a decoder, both of which utilizes the transformer architecture. The encoder first encodes the input sentence into a fixed-length vector, after which an image representation is generated from this vector. Then, the transformer decoder attends on this image to reconstruct the original sentence. Encoding and decoding sentences to and from these image representations are central to the model design. This method allows new sentences to be generated by traversing in the latent space, which makes vector arithmetic, similar to those shown in Word2Vec, possible with sentences as well.

Chapter 2. Background

This chapter provides necessary background on deep learning and the different types of learning objectives it can have. As this research focuses on building an autoencoder, there is an extensive discussion about autoencoders and their advantages, as well as different types of autoencoders and their respective strengths. Following, a brief overview of natural language processing is described along with a discussion of recurrent neural networks and the rise of the attention mechanism. A mathematical description of the transformer model is shown, after which relevant works concerning language modeling and the different approaches to the problem are presented.

2.1 Deep Learning

Deep learning is a subset of machine learning where a deep neural network model is trained for a specific objective with a given dataset. Deep neural network architecture is made possible through many layers of linear transformation with parameters $\boldsymbol{\theta}$ and non-linear activation functions. By using non-linear activation functions at each layer, a deep enough neural network model can learn to approximate any function. This is part of the universal approximation theorem [12, 23, 32].

Given a data space $\mathcal{X}$ where each datapoint is a tuple $(x_i, y_i) \in \mathcal{X}$, a neural network model f is tasked with predicting the target label y_i given x_i as follows: $f(x_i \mid \boldsymbol{\theta}) = \hat{y}_i$. A cost function C is chosen for each task to give a metric to determine how close or similar the prediction is to the target. The model learns by optimizing the objective, which is to minimize the cost $C(y_i, \hat{y}_i)$ by altering the values of $\boldsymbol{\theta}$. This is done by using calculus, where the gradient of C with respect to $\boldsymbol{\theta}$ at each layer shows how the values of the parameters should change in order to lower the cost. With a sufficiently large dataset, suitable architecture, and a feasible objective, a deep neural network model can be trained to perform any given task.

2.2 Supervised, Unsupervised, and Semi-supervised Learning

Machine learning problems can be largely categorized into supervised learning and unsupervised learning. Supervised learning, as the name suggests, is where the machine learns through human supervision. This is accomplished through the use of labeled datasets; for each data point $(x_i, y_i) \in \mathcal{X}$, the target y_i is a desired output that has been labeled by a human. The goal of a supervised learning task is to be able to predict the labeled target given the input, ultimately being able to generalize to new unlabeled inputs. Examples of supervised tasks include image recognition, where the dataset consists of image label pairs; and speech recognition, where the input data is an audio clip of human speech with its text transcription as the label.

In unsupervised learning, the data is not humanly labeled. Instead, the target y_i may be part of the data, generated by an algorithm, or may even be an abstract concept. Unsupervised learning has the advantage in that the data available is nearly unlimited; it takes a great deal of resources and man power to curate a large labeled dataset, whereas unlabeled data is abundant. However, the correct training scheme for a given task is not obvious and is frequently a very difficult problem to solve. But when a solution is found, the results are often astounding.

Semi-supervised learning attempts to take advantage of the best of both learning schemes. Often in semi-supervised learning, a model is first trained unsupervised to learn important patterns about the data and is then fine-tuned using a relatively smaller amount of labeled data in order to perform specific tasks. Many impressive works done with deep learning comes from unsupervised and semi-supervised learning.

2.3 Autoencoders

One unsupervised learning method that has found success with deep learning is an autoencoder, which is a training method where the target is identical to the input, or $y_i = x_i$, and

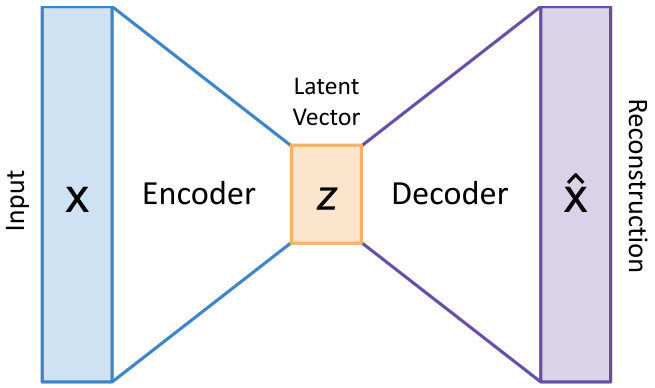


Figure 2.1: Autoencoder design

the training objective of the model is simply to reconstruct the given input. Let $\mathcal{X}$ be the data space and $\mathcal{Z}$ be the encoding space. An auto-encoder consists of two parts: an encoder $E : \mathcal{X} \to \mathcal{Z}$ that encodes a given data $x \in \mathcal{X}$ into a latent variable $z \in \mathcal{Z}$ and a decoder $D : \mathcal{Z} \to \mathcal{X}$ that reconstructs the input from the encoded representation. It is important for $\mathcal{Z}$ to be a lower dimensional space than $\mathcal{X}$ so that the autoencoder model does not reduce to an identity function. This is often referred to as the *bottleneck*.

Autoencoders may be trained for many different reasons. Due to the bottleneck, the encoder must learn to identify important aspects of the data in order to compress the data while still preserving information. This can help the autoencoder model identify deep relationships among data that are not immediately obvious. If the data space $\mathcal{X}$ and the latent space $\mathcal{Z}$ are from the same domain, say a Euclidean space where $X = \mathbb{R}^m$ and $Z = \mathbb{R}^n$ for some $n < m$, then autoencoders can act as a dimensionality reduction algorithm. Often in deep learning however, the data space may be from a completely different domain than the latent space. For example, the inputs may be images or text white the latent space is Euclidean.

Different types of autoencoders were developed in order to meet the needs of various tasks. One such type is denoising autoencoder (DAE) [3, 47], which tries to improve representation by adding an additional transformation to the input data. Let $T : \mathcal{X} \to \mathcal{X}$ be a noise function randomly sampled from a family of noise functions μ_T. Given an input $x \in \mathcal{X}$, a noisy version $T(x)$ is generated as the input for an autoencoder. This noise may be random

noise added to an image or an audio clip, or masking of certain words in text. The decoder is then tasked with reconstructing the input without the added noise. A trained DAE can not only act as a noise remover, but is more robust in encoding input representations because of its ability to better identify key features of the input data.

Variational autoencoder (VAE) [21] is another type of autoencoder that incorporates variational Bayesian methods during its training. The goal of a VAE is not only to encode and reconstruct, but to sample from a distribution imposed by a prior — often a standard Gaussian. Given an input x, the encoder is tasked with producing a mean μ_x and a variance σ_x. A latent variable is then sampled from $z \sim \mathcal{N}(\mu_x, \sigma_x)$, and the decoder reconstructs x using z. The objective of a VAE includes minimizing the difference between the sample distribution $\mathcal{N}(\mu_x, \sigma_x)$ and the prior distribution. This causes the distribution of the latent variables to match the prior. Once trained, a new random variable can be sampled from a known prior distribution, from which the decoder can generate a new object from the otherwise complex and abstract data distribution.

2.4 Natural Language Processing

Let $\mathcal{L}$ be the space of a natural language (i.e. English), and let $\mathcal{V}$ be the set of vocabulary in $\mathcal{L}$. An element $\mathbf{s} \in \mathcal{L}$ is a sentence in that language and can be described as $\mathbf{s} = (\tau_1, \tau_2, ..., \tau_\ell)$ with $\tau_i \in \mathcal{V}$, representing a sentence as an ordered ℓ long list of tokens from the vocabulary set. An element in $\mathcal{L}$ could also be multiple sentences or an entire document, but for the ease of describing and training models, the space will be limited to single sentences.

In order to input a sentence into a machine learning model, it must first be embedded as vectors. This is done using an embedding model that maps each token into a single d-dimensional vector, $E : \mathcal{V} \to \mathbb{R}^d$, where d is usually referred to as the dimension of the model. In practice, this is done by creating a parameter matrix W_E of size $|\mathcal{V}| \times d$ that acts as a lookup table. Each unique token $\tau \in \mathcal{V}$ is assigned a unique index $\mathtt{k} = 1, 2, \ldots, |\mathcal{V}|$ by using an index map $\mathtt{idx} : \mathcal{V} \to \{k \in \mathbb{Z}^+ : k \leq |\mathcal{V}|\}$. Using these tools, the embedding model

can be defined as $E(\tau \mid \boldsymbol{\theta}) := (W_E)^{\mathsf{T}}_{\mathtt{idx}(\tau)}$ where $\boldsymbol{\theta} = (W_E)$. Applying the embedding model to an entire sentence can be defined as an element-wise operation:

$$\begin{aligned} E(\mathbf{s}) &:= (E(\tau_1), E(\tau_2), \ldots, E(\tau_\ell)) \\ &= \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_\ell \end{bmatrix}^{\mathsf{T}} \\ &= \mathbf{X} \in \mathbb{R}^{\ell \times d} \quad . \end{aligned} \tag{2.1}$$

The embedded sentence $\mathbf{X}$ can then be described either as a sequence of d-dimensional vectors with length ℓ or a single matrix of shape $\ell \times d$.

2.5 Recurrent Neural Network

The embedded input vectors in $\mathbf{X}$ are nothing more than a dissociated sequence of words. Meaning in a sentence is not found in the words but from the words in relation to each other. Given $\mathbf{X}$, the goal of a language model is to extract the meaning of the sentence by associating the meaning of the words together as a whole. Recurrent neural networks (RNNs) initially found great success in this domain. The RNN function $\boldsymbol{\rho} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$ with parameters $\boldsymbol{\theta}$ is defined recursively as:

$$\boldsymbol{\rho}(x_t, h_{t-1} \mid \boldsymbol{\theta}) = h_t \tag{2.2}$$

where x_t represents the input vector at time t and the h vectors are the hidden states produced by the model. Hence, given a sequence of inputs $(x_1, x_2, \ldots, x_T)$ and an initial hidden state h_0, $\boldsymbol{\rho}$ produces the output sequence of hidden states $(h_1, h_2, \ldots, h_T)$. These hidden states act as the encoded memory of all the data that has been input so far. The last hidden state vector h_T can then be used to perform various tasks, such as predicting the next data point or extracting information from the data.

In theory, RNNs should be able to keep track of arbitrary long-term dependencies in the input sequences. In practice, however, RNNs have a hard time learning with long input sequences because of the *vanishing gradient* problem [29, 17]. The problem arises from hyperbolic tangent (tanh) function, which is used as the activation function for RNNs, and

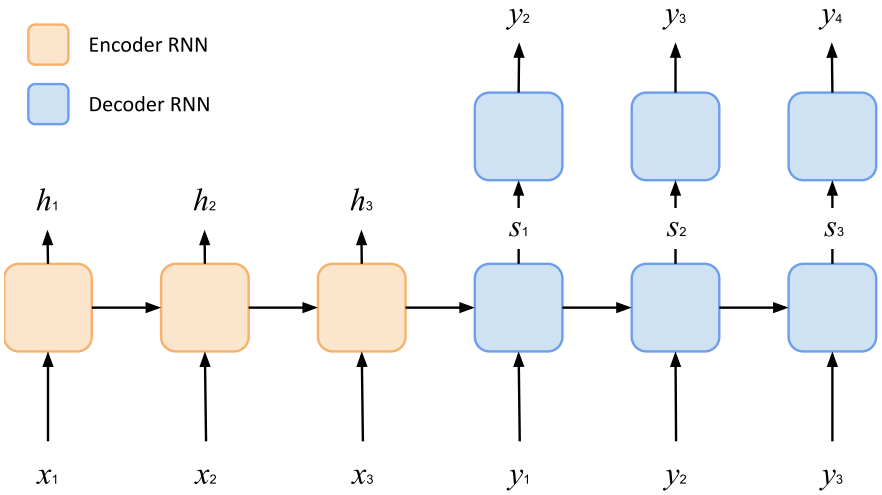


Figure 2.2: Encoder decoder RNN diagram

the image of its derivative that is bounded between 0 and 1. This means that as the gradients gather during back-propagation [38], the output values of $\tanh'(x)$ repeatedly get multiplied, causing the gradients to quickly shrink to zero. As a result, learning will only occur with the last few values of the input sequence and will fail to consider the earlier values.

Different architectures have been designed in order to overcome the short-term memory problem with RNNs. Two of the most widely used RNN architectures are the Long Short-Term Memory (LSTM) [18] and the Gated Recurrent Unit (GRU) [10]. Given an input vector x_t and a hidden state h_{t-1}, the LSTM model is designed as:

$$\begin{aligned} f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\ i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\ \tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{2.3}$$

where $\boldsymbol{\theta} = \big(W_k, U_k, b_k; k \in \{f, i, o, c\}\big)$ are the parameters, $\odot$ represents the Hadamard product (otherwise known as the element-wise product), and σ is the sigmoid function defined

as

$$\sigma(x) := \frac{1}{1 + e^{-x}} \quad .$$

The idea behind the LSTM model design is to allow multiple gates for the input to go through, and the model will learn to selectively choose which inputs will be remembered and which will be forgotten over a long sequence. This can be achieved through c_t which also acts like a hidden state vector that contains memory of all of the input sequences so far, but is updated only by multiplication and addition, thus preventing its gradient from shrinking over time. The GRU model has a similar design, but with less complexity:

$$\begin{aligned} z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\ r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\ \hat{h}_t &= \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \\ h_t &= z_t \odot \hat{h}_t + (1 - z_t) \odot h_{t-1} \quad . \end{aligned} \tag{2.4}$$

The GRU architecture allows the hidden state vectors h_t to have the functionality of LSTM's c_t, instead of having a separate context vector.

Although RNNs perform decently well with language, it is evident that there are some clear drawbacks even with the LSTM and the GRU architecture. By the nature of their design, RNNs are effective in dealing with sequential or time-series data. However, natural languages are fundamentally different from time-series data in that they have grammatical/syntactic structures embedded in them. When predicting weather, for example, recent data are far more important than past data. Conversely, the first few words of a sentence are no more or no less important than the last few words. Language is not sequential, but RNNs would consider it to be so, limiting its performance.

2.6 Attention

The attention mechanism was first introduced for machine translation [2]. It is a very simple, but powerful algorithm where an RNN decoder uses all of the encoder hidden states

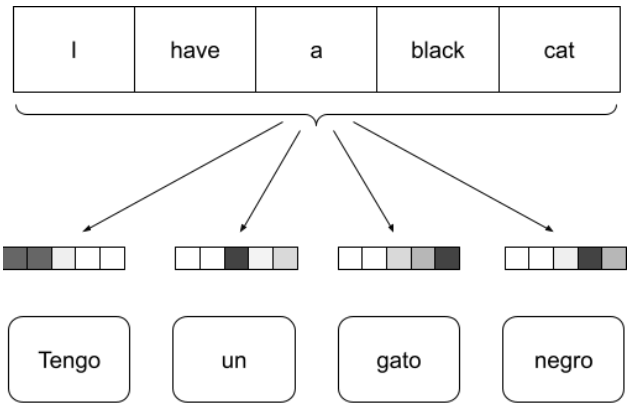


Figure 2.3: The attention mechanism learns to focus on different words of the sentence as the model translates the above sentence from English to Spanish. Darker shade represents higher attention score.

$H = (h_1, \ldots, h_T)$ instead of solely relying on the last hidden state h_T during the decoding phase. Let $(y_1, \ldots, y_t)$ be the embedded decoder inputs and $(s_1, \ldots, s_t)$ be the decoder hidden states produced so far by a decoder RNN, so $s_i = \boldsymbol{\rho}_{\mathbf{d}}(y_i, s_{i-1})$. The attention function first produces a context vector c_t by taking a weighted sum of the encoder hidden states,

$$\begin{aligned} c_t &= \sum_{i=1}^{n} \alpha_{t,i} h_i \\ &= \boldsymbol{\alpha}_t^{\mathsf{T}} H \end{aligned} \tag{2.5}$$

where $\mathbb{1}^T \boldsymbol{\alpha}_t = 1$. The weights $\boldsymbol{\alpha}_t$ are given by an alignment score function that calculates the relevance between the next word to be generated $\hat{y}_{t+1}$ and each word from the source sentence $(x_1, \ldots, x_T)$. In practice, the encoder hidden states, H, are used instead of the encoder inputs since the information of the source sentence is encoded in the hidden vectors. As y_{t+1} has not yet been generated, the last hidden state vector s_t is used as the input for this function. This gives us the following equation for the alignment weights:

$$\boldsymbol{\alpha}_t = \boldsymbol{\sigma}\big(\text{score}(s_t, H)\big)$$

where $\boldsymbol{\sigma}$ is the softmax function defined as

$$\boldsymbol{\sigma}(\mathbf{w})_i = \frac{e^{w_i}}{\sum_{j=1}^{K} e^{w_j}} \quad \text{for } \; i = 1, \ldots, K \;\; \text{and} \;\; \mathbf{w} = (w_1, \ldots, w_K) \in \mathbb{R}^K \quad .$$

In translation tasks, the attention mechanism treats the words to be generated in the target language and the words from the source sentence as a fully connected graph. As

illustrated from Figure 2.3, the model learns to know which word to attend to while generating in the target language. Different types of attention come from how the alignment score function is defined. The table below describes the most notable types of attention and their corresponding alignment score functions:

Name	Score function	Citation
Content-based attention	$\text{score}(s_t, h_i) = \frac{s_t^\mathsf{T} h_t}{\|\|s_t\|\| \, \|\|h_i\|\|}$	Graves 2014 [15]
Additive	$\text{score}(s_t, h_i) = v_a^\mathsf{T} \tanh(W_a s_{i-1} + U_a h_j)$	Bahdanau 2015 [2]
Dot-product	$\text{score}(s_t, h_i) = s_t^\mathsf{T} h_i$	Luong 2015 [24]
Scaled dot-product	$\text{score}(s_t, h_i) = \frac{s_t^\mathsf{T} h_i}{\sqrt{n}}$	Vaswani 2017 [46]

2.6.1 Image Captioning. The attention mechanism was initially introduced to be used in conjunction with an RNN encoder and decoder for translation tasks, but it soon proved to be useful in many other areas. One such usage demonstrated an effective way of creating a model that can generate precise and accurate descriptions of images using the attention mechanism [49]. To summarize, the model is trained using labeled data pairs of an image and its caption. The image is first fed into convolutional networks, which process the image and downsamples it. The resulting output is a 3-dimensional tensor with much smaller image dimensions, but with a large number of channels. For example, a square input image may be of shape $3 \times 128 \times 128$, where the channel dimension represents the RGB color channels. After being processed through convolutional layers, the resulting output might be of shape $256 \times 4 \times 4$.

By treating each "pixel" of the downsampled image as individual vectors, this results in 16 different 256-dimensional vectors. Hence, the output tensor can essentially be seen as a sequence of d-dimensional vectors, commonly referred to as the convolutional filters of the image. The filter vectors contain local, high level information about the image. These

Figure 2.4: Image captioning using attention. The model learns to attend to specific parts of the image that corresponds to the word that is being decoded. The brighter areas of the image correspond to higher attention score.

vectors are then used for the attention mechanism, where the decoder RNN will attend to the filter vectors as it generates each word. The result was shown to be remarkably successful, as seen from Figure 2.4. This research showed that it is possible to use attention on image data in the same manner as it is used with language data.

2.6.2 Self-attention. Another usage of the attention mechanism is self-attention [9]. Self-attention is a mechanism where the sequence which a word attends to comes from the sentence itself. As a result, each word of the sentence attends to every other word in the sentence to calculate how they are similar, related, and/or connected. This mechanism was used for machine reading that showed how attention can be used to find correlation between the current word and all words that come before it.

Self-attention is effective in machine reading when used with RNNs, but it proves to be

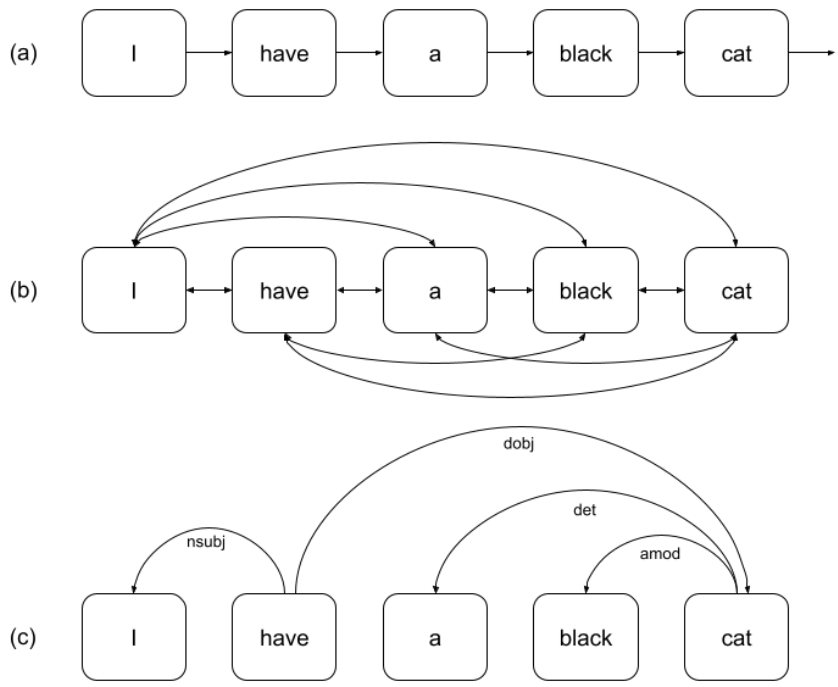


Figure 2.5: a) RNNs treat the input sentence as a sequence of words. b) Self-attention mechanism treats the sentence as a fully connected graph, with each word as a node. c) Dependency grammar of the sentence used in linguistics is similar to self-attention.

far more useful when RNNs are not involved at all. RNNs treat all inputs as a sequential data with a temporal element. Language is written and spoken sequentially, but it is not understood sequentially — we process the sentence holistically, taking into account syntactic and pragmatic rules, context, and so on. On the other hand, the self-attention mechanism treats words in a sentence as nodes of a fully connected graph. This is much closer to how language is modeled linguistically, such as with syntax trees and dependency grammar, which is a sign that attention mechanism is better suited for language modeling compared to the RNN. The self-attention mechanism is a core design of the transformer.

2.7 Transformer

The transformer [46] is a modern language processing architecture that does not involve RNNs. Instead, it fully utilizes the power of the attention and self-attention mechanisms. By solely relying on attention, it bypasses RNN's weakness in viewing language as sequential data. The full transformer model is a combination of the transformer encoder model and the transformer decoder model. The two models have near identical architectures, but with a key difference: the encoder model only uses self-attention along with a fully connected layer to encode the input sentence, whereas the decoder model has an additional attention layer to attend to the encoder outputs. The goal of this section is to describe the transformer architecture using simple mathematical expressions.

Let $\mathbf{X} = \begin{bmatrix}\mathbf{x}_1 & \cdots & \mathbf{x}_{\ell_\mathbf{x}}\end{bmatrix}^\mathsf{T} \in \mathbb{R}^{\ell_\mathbf{x} \times d}$ be the embedded vectors of the input sentence and $\mathbf{Y} = \begin{bmatrix}\mathbf{y}_1 & \cdots & \mathbf{y}_{\ell_\mathbf{y}}\end{bmatrix}^\mathsf{T} \in \mathbb{R}^{\ell_\mathbf{y} \times d}$ be the embedded vectors of the target sentence. The transformer model can be divided into two models: The transformer encoder, $\mathcal{T}_\mathbf{e} : \mathbb{R}^{\ell_\mathbf{x} \times d} \to \mathbb{R}^{\ell_\mathbf{x} \times d}$, and the transformer decoder, $\mathcal{T}_\mathbf{d} : \mathbb{R}^{\ell_\mathbf{x} \times d} \times \mathbb{R}^{\ell_\mathbf{y} \times d} \to \mathbb{R}^{\ell_\mathbf{y} \times d}$. In order to describe these two models in detail, it is necessary to first describe the architectures of the attention layer and the fully connected layers.

The attention mechanism used in the transformer is based on scaled dot-product attention. The inputs for the attention function are the Queries $Q \in \mathbb{R}^{\ell_1 \times d}$, the Keys $K \in \mathbb{R}^{\ell_2 \times d}$, and the Values $V \in \mathbb{R}^{\ell_2 \times d}$. These are all sentences represented as matrices, where each row represents a token vector of the sentence. In essence, each token from the Query sentence attends to each token from the Key sentence using a dot product. This is done simply using matrix multiplication QK^T. The output is scaled by a factor of $\sqrt{d}$ so that the variance of the dot products are normalized even when d is very large. Softmax function $\boldsymbol{\sigma}$ is then applied to the output matrix along the row dimension, so each row-sum becomes 1. The resulting matrix is then multiplied with the Value matrix. This can be summarized as:

$$\mathcal{A}(Q, K, V) := \boldsymbol{\sigma}\left(\frac{QK^\mathsf{T}}{\sqrt{d}}\right)V \quad . \tag{2.6}$$

Using this attention function as the base attention model, the transformer introduces a

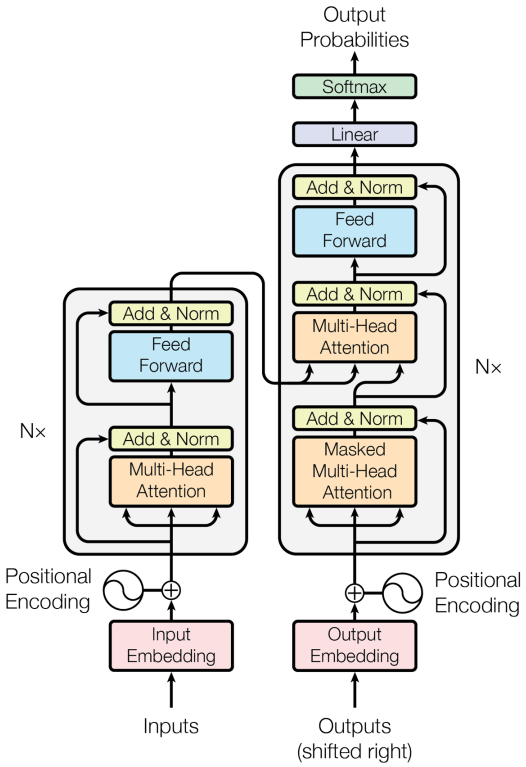


Figure 2.6: The transformer-model architecture

novel attention architecture called multihead attention. In essence, each of the matrices Q, K, and V are projected onto a lower dimensional space, using trainable parameters, from where the attention is performed. The attention output is called a *head*, and multiple *heads* are concatenated to restore the original dimension. Multihead attention with h heads is formalized as follows:

$$\mathcal{M}(Q, K, V \mid \boldsymbol{\theta}) := \begin{bmatrix} H_1 & \cdots & H_h \end{bmatrix}^{\mathsf{T}} W^O \tag{2.7}$$

$$H_i = \mathcal{A}\big(QW_i^Q, KW_i^K, VW_i^V\big) \tag{2.8}$$

where $\boldsymbol{\theta} = \big(W_i^Q, W_i^K, W_i^V \; ; \; i = 1, \ldots h\big)$ and the projection parameters are shaped $W_i^Q \in \mathbb{R}^{d \times d_k}$, $W_i^K \in \mathbb{R}^{d \times d_k}$, $W_i^V \in \mathbb{R}^{d \times d_v}$, and $W^O \in \mathbb{R}^{hd_v \times d}$ so that the dimensions of the output is consistent. In practice, $d_k = d_v = d/h$.

The fully connected layer in transformer architecture is just a two layer linear network with a Rectified Linear Unit (ReLU) activation function, defined as $\mathrm{ReLU}(x) = \max(0, x)$.

The full architecture of this network is:

$$\mathcal{F}(x \mid \boldsymbol{\theta}) = W_2 \,\mathrm{ReLU}(W_1 x + b_1) + b_2 \tag{2.9}$$

where $\boldsymbol{\theta} = (W_1, b_1, W_2, b_2)$.

With the attention layer and the fully connected layer defined, the transformer architecture can be described using a simple and concise set of equations. The transformer encoder, $\mathcal{T}_{\mathbf{e}} : \mathbb{R}^{\ell_{\mathbf{x}} \times d} \to \mathbb{R}^{\ell_{\mathbf{x}} \times d}$ with N layers is defined as

$$\begin{aligned} \mathcal{T}_{\mathbf{e}} &:= \mathcal{E}_N \circ \mathcal{E}_{N-1} \cdots \mathcal{E}_1 \\ \mathcal{E}_i &:= \mathcal{F}_{\mathbf{e},i} \circ \mathcal{M}_{\mathbf{e},i} \end{aligned} \tag{2.10}$$

where

$$\begin{aligned} \mathcal{M}_{\mathbf{e},i} &:= \mathcal{M}(X, X, X \mid \boldsymbol{\theta}_{\mathbf{e},i}) + X \\ \mathcal{F}_{\mathbf{e},i} &:= \mathcal{F}(X \mid \boldsymbol{\theta}_{\mathbf{e},i}) + X \end{aligned}$$

represent residual connections [16] between the input and the output. Similarly, the transformer decoder $\mathcal{T}_{\mathbf{d}} : \mathbb{R}^{\ell_{\mathbf{x}} \times d} \times \mathbb{R}^{\ell_{\mathbf{y}} \times d} \to \mathbb{R}^{\ell_{\mathbf{y}} \times d}$ with N layers is defined as:

$$\begin{aligned} \mathcal{T}_{\mathbf{d}} &:= \mathcal{D}_N \circ \mathcal{D}_{N-1} \cdots \mathcal{D}_1 \\ \mathcal{D}_i &:= \mathcal{F}_{\mathbf{d},i} \circ \mathcal{M}^{(2)}_{\mathbf{d},i} \circ \mathcal{M}^{(1)}_{\mathbf{d},i} \end{aligned} \tag{2.11}$$

with

$$\begin{aligned} \mathcal{M}^{(1)}_{\mathbf{d},i} &:= \mathcal{M}(Y, Y, Y \mid \boldsymbol{\theta}^{(1)}_{\mathbf{d},i}) + Y \\ \mathcal{M}^{(2)}_{\mathbf{d},i} &:= \mathcal{M}(Y, X, X \mid \boldsymbol{\theta}^{(2)}_{\mathbf{d},i}) + Y \\ \mathcal{F}_{\mathbf{d},i} &:= \mathcal{F}(Y \mid \boldsymbol{\theta}_{\mathbf{d},i}) + Y \quad . \end{aligned}$$

As can be seen, the transformer encoder and decoder layers are nearly identical. There is an additional attention sub-layer that the decoder uses to attend to the encoder outputs. The final part of the transformer is the generator, which takes in the d dimensional vectors of the decoder output and maps them into a probability distribution with $|\mathcal{V}|$ possible outputs, each of which represents a token in $\mathcal{V}$. The output distribution is what is used to predict the next token. Given a vector input $x \in \mathbb{R}^d$, the generator function $\mathcal{G} : \mathbb{R}^d \to \mathbb{R}^{|\mathcal{V}|}$ can be

summarized as:

$$\mathcal{G}(x \mid \boldsymbol{\theta}) = \boldsymbol{\sigma}(Wx + b)$$

where $\boldsymbol{\theta} = (W, b)$.

Given an embedded input X, an embedded target Y, the transformer encoder function $\mathcal{T}_{\mathbf{e}}$, the transformer decoder function $\mathcal{T}_{\mathbf{d}}$, and the generator function $\mathcal{G}$, the full transformer model with all of its parameters represented as $\boldsymbol{\theta}$ can be summarized as a function $\mathcal{T} : \mathbb{R}^{\ell_{\mathbf{x}} \times d} \times \mathbb{R}^{\ell_{\mathbf{y}} \times d} \to \mathbb{R}^{|\mathcal{V}| \times d}$ and is given by:

$$\mathcal{T}(X, Y \mid \boldsymbol{\theta}) = \mathcal{G} \circ \mathcal{T}_{\mathbf{d}} \left(\mathcal{T}_{\mathbf{e}}(X), Y \right) \tag{2.12}$$

2.8 Language Models

In this section, a discussion of current language models and their approaches will take place. This section focuses on the training objectives of the different models rather than the architectural designs, since all of the current state-of-the-art language models utilize the powerful transformer architecture. Transformer has shifted the approach in language modeling away from RNNs and have shown incredible performance and flexibility across all language related tasks.

2.8.1 Skip-Thought Vectors. The Skip-Gram model [28] [27], the model design behind Word2Vec, is an unsupervised word embedding model that attempts to extract the meaning of a word by predicting the words that come before and after the input word. Given an embedded word vector w_t, the Skip-Gram model is tasked to predict the surrounding n-gram $(w_{t-n}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+n})$. The Skip-Thought Vectors [22] use the same conceptual idea, but extend its application from words to sentences. The model takes a sentence as its input and attempts to predict the sentences that come before and after the input sentence. More formally, each datapoint can be represented as an ordered triplet (s_{i-1}, s_i, s_{i+1}). An encoder is first tasked to encode s_i to some latent vector z. Then, one decoder is tasked to predict s_{i-1} given z and another is tasked to predict s_{i+1} given z. By predicting the

surrounding context of the input sentence, the model is able to extract the meaning of s_i and embed it in the vector z.

2.8.2 Universal Sentence Encoder. The Universal Sentence Encoder [7] was the first sentence encoding language model developed using the transformer. The goal is to show how sentences can be encoded into a continuous representation which can then perform various NLP tasks, such as calculating semantic textual similarity, transfer learning, and Word Embedding Association Tests [6]. The transformer based model is constructed and trained through a procedure described as follows:

> The transformer based sentence encoding model constructs sentence embeddings using the encoding sub-graph of the transformer architecture... The context aware word representations are converted to a fixed length sentence encoding vector by computing the element-wise sum of the representations at each word position. The encoding model is designed to be as general as possible. This is accomplished by using multi-task learning whereby a single encoding model is used to feed multiple down-stream tasks. The supported tasks include: a Skip-Thought like task for the unsupervised learning from arbitrary running text; a conversational input-response task for the inclusion of parsed conversational data; and classification tasks for training on supervised data.

2.8.3 GPT. Generative Pre-Trained Transformers (GPT) [34] is trained semi-supervised using a decoder-only transformer. The model is pre-trained using unsupervised data with the objective of predicting the next token given an input corpus of tokens $\mathbf{T} = (\tau_1, ..., \tau_n)$ by maximizing the following likelihood:

$$L_1(\mathbf{T}) = \sum_i \log P(\tau_i \mid \tau_{i-k}, \ldots, \tau_{i-1}; \boldsymbol{\theta})$$

where k is the size of the context window, and the conditional probability P is modeled using a transformer with parameters $\boldsymbol{\theta}$.

The simple training objective allows for near unlimited amounts of data. The GPT model

is then fine-tuned using supervised data, labeled to perform four different linguistic tasks: classification, textual entailment, similarity, and question answering. The training procedure is described as follows:

> We assume a labeled dataset $\mathcal{C}$, where each instance consists of a sequence of input tokens, $x^1, \ldots, x^m$, along with a label y... This gives us the following objective to maximize:
>
> $$L_2(\mathcal{C}) = \sum_{(x,y)} \log P(y \mid x^1, \ldots, x^m) \quad .$$
>
> We additionally found that including language modeling as an auxiliary objective to the fine-tuning helped learning by (a) improving generalization of the supervised model, and (b) accelerating convergence... Specifically, we optimize the following objective (with weight λ):
>
> $$L_3(\mathcal{C}) = L_2(\mathcal{C}) + \lambda * L_1(\mathcal{C}) \quad .$$

The results showed that large language models trained on large amounts of unsupervised data have a great capacity for transfer learning, which "aims at improving the performance of target learners on target domains by transferring the knowledge contained in different but related source domains. [50]"

2.8.4 BERT. Bidirectional Encoder Representations from Transformers (BERT) [13] is similar to GPT, but has one significant difference in its architectural design; BERT is a bidirectional transformer, meaning it can predict the previous token as well as the next token given an input corpus. BERT also follows pre-training and fine-tuning method to train for downstream tasks but with a different training objective from GPT. The pre-training for BERT uses two unsupervised tasks: masked LM and next sentence prediction.

In masked LM, 15% of the input words are masked and BERT is tasked with predicting the masked words. This training method is reminiscent of the denoising autoencoders discussed in 2.3, but differs in that the model only predicts the masked portions rather than reconstructing the entire input. The second unsupervised task used to train BERT is next

sentence prediction. Given sentences A and B, 50% of the time B is the actual next sentence and 50% of the time B is a random sentence selected from the corpus. BERT is tasked with predicting whether sentence B is the true next sentence or not. The fine-tuning phase for BERT also involves four linguistic tasks: paraphrasing, entailment, question answering, and classification.

Chapter 3. Model Design

This research presents a way to represent language with images and to train an autoencoder model end-to-end using transformer encoder and decoder. There are a few challenges with encoding and decoding language with a latent vector. The first challenge is that sentences come with varying number of words or tokens. With many of the existing language models that utilizes the transformer, the approach often times is to sum or average the transformer outputs. For autoencoder training however, the aim is to not only encode, but also to decode, hence it is necessary for the latent vector to fully encapsulate the semantics and the syntactic structure of the input sentence for reconstruction.

Another major difference with the existing language models is the decoder half. Since a key design in the transformer decoder is attention on the encoder outputs, no known attempts have been made to use the transformer decoder on a single vector. As the encoder and decoder trains simultaneously on the same latent space, there is an advantage of being able to generate new sentences from a latent vector. This research presents a novel method in encoding and decoding language to and from latent vectors by using image representations.

3.1 Encoder Design

The main task of the encoder is to take the sequential vectors of the transformer output and contain all the necessary information about these vectors into a single vector representation. The challenge here is that the length of each sentence varies, and many sentences are structured differently linguistically. For example, some sentences start with a subject pronoun, followed by a verb and an object, whereas other sentence may start with an article, then a subject noun. Even more, interrogative sentences start with an auxiliary verb (such as be, can, do, etc.) and is then followed by the subject. With so many possible configurations for every sentence, reducing the transformer output vectors, where each vector represents corresponding input token, into a single vector without the loss of semantic and syntactic

information is challenging.

Given an embedded sentence input $\mathbf{X} \in \mathbb{R}^{\ell_{\mathbf{x}} \times d}$, let $\mathcal{T}_{\mathbf{e}}(\mathbf{X}) = \mathbf{A}$. Since the output dimension of $\mathcal{T}_{\mathbf{e}}$ is consistent with the input dimension, $\mathbf{A} \in \mathbb{R}^{\ell_{\mathbf{x}} \times d}$. The approach that is used in order to first eliminate the dimension $\ell_{\mathbf{x}}$ is to multiply the transpose of $\mathbf{A}$ by itself. For $\mathbf{B} = \mathbf{A}^{\mathsf{T}}\mathbf{A}$, $\mathbf{B} \in \mathbb{R}^{d \times d}$ is free of the sequence-length dimension $\ell_{\mathbf{x}}$. Define $\Psi : \mathbb{R}^{\ell_{\mathbf{x}} \times d} \to \mathbb{R}^{d \times d}$ as the transpose multiplication function:

$$\Psi(\mathbf{A}) := \mathbf{A}^{\mathsf{T}}\mathbf{A} \quad . \tag{3.1}$$

Since the dimension of the model is much larger than the sequence length, $\mathbf{A}$ almost always will be full-rank, and the $d \times d$ matrix $\mathbf{B}$ is a much larger matrix than $\mathbf{A}$. Therefore, Ψ is almost always an injective function, which means that all of the necessary information of $\mathbf{A}$ is preserved in $\mathbf{B}$.

The next step is to compress the matrix $\mathbf{B}$ into a single latent vector $\mathbf{z} \in \mathbb{R}^d$ through some function $\Phi : \mathbb{R}^{d \times d} \to \mathbb{R}^d$ with as little loss of information as possible. First, imagine the goal is not to transform $\mathbf{B}$ into a vector, but rather an image. An image can be thought of as a 3-dimensional tensor of shape $C \times k \times k$, where C is a channel dimension and k is the width and height of a square image. Allow d to be a divisor of C so that each i^{th} column of $\mathbf{B}$ multiplies with some $\frac{Ck^2}{d} \times d$ parameter matrix W_i, resulting in a vector that is reshaped to be an image of shape $\frac{C}{d} \times k \times k$. These are then stacked on top of each other along the channel dimension, producing a desired image of shape $C \times k \times k$.

In order for the encoding to be a vector, simply set $k = 1$ and $C = d$. Then, the output image is of shape $d \times 1 \times 1$, which is just a d-dimensional vector. Each parameter matrix also reduces to a d-dimensional vector w_i, and simply dot-products with the i^{th} column of $\mathbf{B}$. Let $W = \begin{bmatrix} w_1 & \cdots & w_n \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \cdots & \mathbf{b}_n \end{bmatrix}$. The process can be summarized as:

$$\Phi(\mathbf{B} \mid W) := \text{diagonal}(W^{\mathsf{T}}\mathbf{B}) = \begin{bmatrix} w_1^{\mathsf{T}}\mathbf{b}_1 \\ w_2^{\mathsf{T}}\mathbf{b}_2 \\ \vdots \\ w_n^{\mathsf{T}}\mathbf{b}_n \end{bmatrix} \quad . \tag{3.2}$$

Define $\mathbf{z} := \Phi(\mathbf{B} \mid W)$ to be the encoded latent vector for future reference.

3.2 Decoder Design

From the latent vector z, the image representation is generated using traditional methods. The vector is repeatedly upsampled and fed into a CNN until a desired shape of the image is reached. The fact that images can be attended on is one of the inspirations of the decoder design. The convolutional filters of the image representation is obtained following a similar procedure as shown in Section 2.6.1, which is then used for attention by the transformer decoder model. The transformer decoder then reconstructs the original input sentence.

The generated image representation space may act as a prior to the Euclidean latent space, since the transformer decoder model only "sees" the image representation and never actually interacts with the latent vector. This is unproven, but empirical results show that using a linear network instead to generate multiple vectors to be attended on doesn't perform nearly as well. This suggests that using image representation is actually advantageous for the decoder's performance.

3.3 Full Model Pipeline

The details of the entire model design is presented in this section. The model is coded in Python, and the neural network model is built using the PyTorch [30] library. A detailed diagram describing the model architecture is shown in Figure 3.1.

The first part of the model is the embedding model $E : |\mathcal{V}| \to d$ as described in equation 2.1. The embedding model is built using `nn.Embedding` module, in which M_E also acts as trainable weights. The model dimension d was set to be 256. The embedded input then passes through transformer encoder $\mathcal{T}_\mathbf{e}$. There are no major changes to the transformer architecture, so refer to equation 2.10 for how the transformer encoder is designed. The code for building the transformer was obtained from The Annotated transformer [39]. The number of encoder layers N was set to be 6, the number of attention heads for multihead attention h was set to be 8, and the dimension of the feed-forward network was set to be 1024.

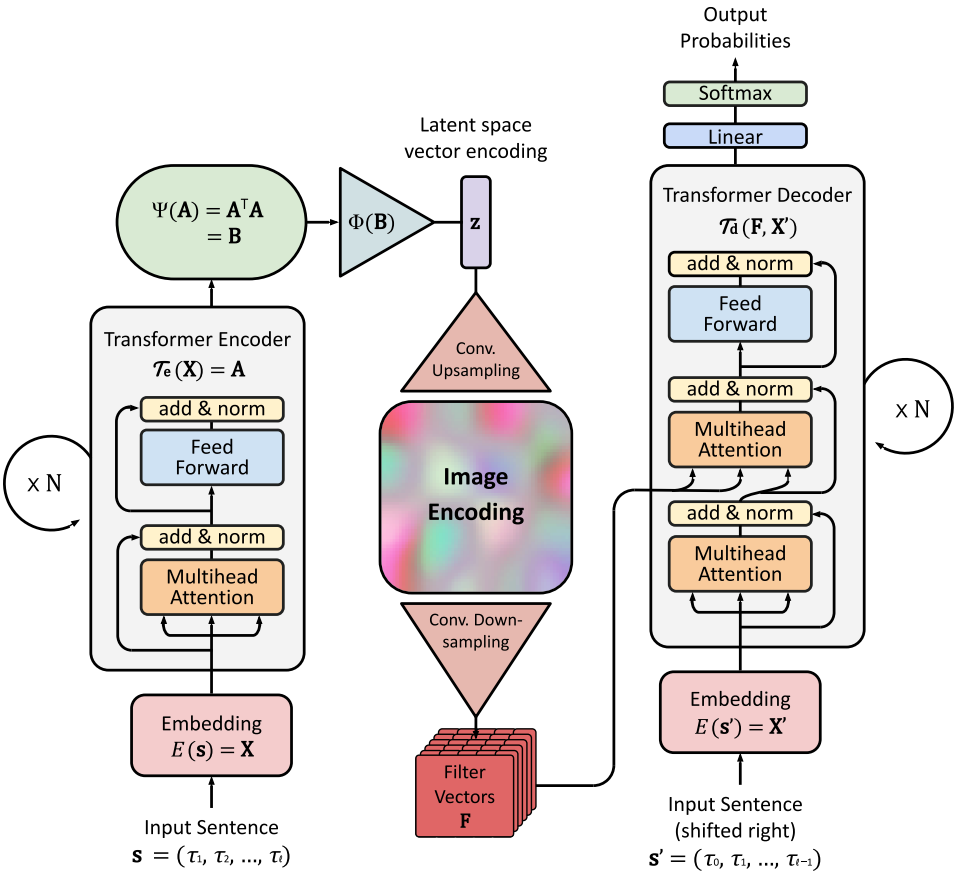


Figure 3.1: Full model diagram

The output of the transformer encoder progresses through the model exactly as outlined in equations 3.1 and 3.2 to produce the latent vector $\mathbf{z}$.

To generate the image encoding from $\mathbf{z}$, it passes through a convolutional upsampling function $\mathcal{C}_{\text{up}} : \mathbb{R}^d \to \mathbb{R}^{3\times d_\mathcal{I}\times d_\mathcal{I}}$, where 3 represents the RGB channel of the resulting image, and $d_\mathcal{I}$ is the width and height of the square image. First, $\mathbf{z}$ passes through a linear layer of output size $2d$ and is then reshaped to be a $\frac{d}{8} \times 4 \times 4$ tensor. It then passes through an initial convolutional layer with $2d$ output channels, square kernels of size 3, and stride set to be 1. All convolutional layers are built using `nn.Conv2d` module. By padding the edges by 1, this kernel size and stride preserves the shape of the image, resulting in a $2d \times 4 \times 4$ tensor output. Padding is done using `nn.ReflectionPad2d`, which pads the edges using values from the opposite edges. This helps the values at the boundaries to stay consistent with the other values during upsampling.

The output tensor then passes through a series of upsampling and convolutional layers, designed to increase the dimensions of the image while decreasing the number of channels. For each layer, the upsampling is done using `nn.Upsample` module with bilinear interpolation, and each convolution halves the input channels while preserving the size of the image with kernel size of 3 and stride of 1 with padding. After each layer, a ReLU function is applied as the activation function. The number of layers $N_{\mathcal{C}}$ is set to be 4, hence the resulting output tensor will be of shape $\frac{d}{8} \times 64 \times 64$. This tensor is then inputted to a final convolutional layer that reduces the channel dimension to 3, so that the output is a visual image with RGB color channels. A hyperbolic tangent function is applied at the end to bound the pixel values between -1 and 1. This whole process will be summarized as:

$$\mathcal{C}_{\text{up}}(\mathbf{z}) = \mathcal{I} \quad . \tag{3.3}$$

The downsampling method to obtain the filter vectors from the image works similarly to the upsampling method. The image tensor $\mathcal{I}$ passes through convolutional downsampling function $\mathcal{C}_{\text{down}} : \mathbb{R}^{3 \times d_{\mathcal{I}} \times d_{\mathcal{I}}} \to \mathbb{R}^{N_{\mathbf{f}} \times d}$, where $d_{\mathcal{I}}$ is the dimension of the image $\mathcal{I}$ and $N_{\mathbf{f}}$ is the number of filter vectors. The image first goes through an intial convolutional layer with kernel size 3 and stride 1 to set the channel dimension to be $d/2^{Nc}$. Then, it passes through a series of convolutional layers, where each layer contains a convolution that halves the number of channels and also halves the image dimension by using kernel size 2 and stride 2, then another convolution that preserves the channel and image dimensions with kernel size 3 and stride 1. ReLU function is applied after as well for activation. The number of layers for downsampling is identical to the number of upsampling layers, which is set to 4. This results in 16 filter vectors of dimension d that will be used as inputs for the transformer decoder. This process will be summarized as:

$$\mathcal{C}_{\text{down}}(\mathcal{I}) = \mathbf{F} = \begin{bmatrix} \mathbf{f}_1 & \mathbf{f}_2 & \cdots & \mathbf{f}_{N_{\mathbf{f}}} \end{bmatrix}^{\mathsf{T}} \quad . \tag{3.4}$$

For the decoding phase, the target sentence for the transformer decoder $\mathcal{T}_{\mathbf{d}}$ is identical to the input sentence since the model is trained as an autoencoder. The only difference is that the tokens are all shifted one to the right. For example, given the filter vectors $\mathbf{F}$ and

tokens $(\tau_0, \ldots, \tau_k)$ up to position k, the decoder's job is to predict the next token τ_{k+1}. τ_0 is always a start of sentence token that is used in predicting τ_1. Since the input and target are the same, they also share the same embedding model. The number of layers and the number of attention heads for the decoder model are identical to that of the encoder model.

Let $\mathbf{s} = (\tau_1, \tau_2, \ldots, \tau_\ell)$ be the input sentence and $E(\mathbf{s}) = \mathbf{X}$ be the embedded sentence. Let $\mathbf{s}' = (\tau_0, \tau_1, \ldots, \tau_{\ell-1})$ be the input sentence that is shifted one index to the right and $\mathbf{s}'_k = (\tau_0, \ldots, \tau_k)$ where $0 \leq k \leq \ell - 1$. This gives $E(\mathbf{s}'_k) = \mathbf{X}'_k$. Using these inputs, below is a summary of the entire model pipeline:

$$\begin{aligned} \mathcal{T}_e(\mathbf{X}) &= \mathbf{A} \\ \Psi(\mathbf{A}) &= \mathbf{B} \\ \Phi(\mathbf{B}) &= \mathbf{z} \\ \mathcal{C}_{\text{up}}(\mathbf{z}) &= \mathcal{I} \\ \mathcal{C}_{\text{down}}(\mathcal{I}) &= \mathbf{F} \\ (G \circ \mathcal{T}_d)(\mathbf{F}, \mathbf{X}'_k) &= p(\tau_{k+1} \mid \tau_1, \ldots, \tau_k) \quad . \end{aligned} \tag{3.5}$$

The resulting output of the model is the probability distribution over $\mathcal{V}$ for the next token τ_{k+1}.

Chapter 4. Gradient Analysis

In this chapter, the back-propagation of the model at the layers Ψ and Φ will be analyzed to show that the model is designed to learn effectively during training. Attention mechanism also involve similar matrix operations, so the analysis not only applies to this model, but all models that use dot product attention and the transformer.

4.1 Back-propagation

Before analyzing the layers of the model, the basics of back-propagation will be shown for a simple case. Suppose the model is an L-layer neural network, where each layer is a linear layer with an activation function. The forward-propagation of the model can be defined recursively with the equation:

$$f_l(W_l a_{l-1} + b_l) = a_l$$

where a_l is the activated output of the l^{th} layer, W_l and b_l are the trainable weight and bias, and f_l is some activation function, most commonly the ReLU function, or the Sigmoid/Softmax function at the output layer. (Note that f needs to be differentiable almost everywhere.) Also define z_l as the l^{th} weighted input, $z_l = W_l a_{l-1} + b_l$, hence $a_l = f_l(z_l)$.

Let $a_0 = x$, where x is the input of the model, and let y be the corresponding target with $\hat{y}$ being the prediction of y. Then, the L^{th}, or the final output of the model is $a_L = \hat{y}$. The cost function C is some differentiable function that measures how *close* the prediction $\hat{y}$ is to the target y. It maps y and $\hat{y}$ to a scalar-valued metric, commonly referred to as the *cost*. The training objective of a neural network model is to minimize this cost, which can be achieved through the back-propagation algorithm.

Back-propagation works in a similar way to forward-propagation. Just as how the input flows forward through each layer starting from the initial layer to the final layer, the gradient of the cost function flows backward through each layer from back to front using chain rule.

First, observe that at each layer l, the following derivatives are given:

$$\begin{aligned}
\frac{\partial a_l}{\partial z_l} &= f_l'(z_l) \\
\frac{\partial z_l}{\partial W_l} &= a_{l-1}^{\mathsf{T}} \\
\frac{\partial z_l}{\partial b_l} &= I \\
\frac{\partial z_l}{\partial a_{l-1}} &= W_l^{\mathsf{T}} \quad .
\end{aligned}$$

Then, to calculate the derivatives with respect to the cost function C, define the error at the l^{th} layer to be $\delta_l = \dfrac{\partial C}{\partial z_l}$. Using chain rule, the derivatives of C with respect to the variables at each layer can be recursively calculated as follows:

$$\delta_L = \frac{\partial C}{\partial z_L} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_L} = \nabla_{\hat{y}} C \; f_L'(z_L)$$

$$\delta_l = \frac{\partial C}{\partial z_l} = \frac{\partial C}{\partial z_{l+1}} \frac{\partial z_{l+1}}{\partial a_l} \frac{\partial a_l}{\partial z_l} = W_{l+1}^T \delta_{l+1} \; f_l'(z_l)$$

$$\frac{\partial C}{\partial b_l} = \frac{\partial C}{\partial z_l} \frac{\partial z_l}{\partial b_l} = I \delta_l = \delta_l$$

$$\frac{\partial C}{\partial W_l} = \frac{\partial C}{\partial z_l} \frac{\partial z_l}{\partial W_l} = \delta_l a_{l-1}^{\mathsf{T}} \quad .$$

As the gradients are computed at each layer, the weights and biases are updated by taking a small step in the direction of their gradient to the cost function. Let η be that small step, or *learning rate* of the model. The weight and bias at each layer l are updated as below:

$$W_l \leftarrow W_l - \eta \frac{\partial C}{\partial W_l}$$

$$b_l \leftarrow b_l - \eta \frac{\partial C}{\partial b_l} \quad .$$

This is called the *gradient descent* [37, 19]. As the model trains with many data over multiple epochs, the cost may eventually reach its minimum.

4.2 Gradient of Ψ

Before attempting to solve for the gradient of the matrix function $\Psi(X) = X^\mathsf{T}X$, the gradient of a general vector function will be reviewed. Let $F : \mathbb{R}^n \to \mathbb{R}^m$ be a vector function, where $F = \begin{bmatrix} f_1 & f_2 & \cdots & f_m \end{bmatrix}^\mathsf{T}$, with respect to an input vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^\mathsf{T}$. We can solve for the gradient of F with respect to $\mathbf{x}$ by solving for the partial derivatives of each function f_i with respect to each input x_j. This is written as

$$\nabla_\mathbf{x} F = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} . \tag{4.1}$$

This matrix with all of its first-order partial derivatives is also commonly referred to as the Jacobian matrix of the function F, written as $\mathbf{J}_F$.

Since the gradient of a vector-to-vector function F is an $m \times n$ matrix, we can logically conclude that the gradient of a matrix-to-matrix function Ψ must be a 4-dimensional tensor of shape $(n \times n) \times (m \times n)$ that, when right multiplied by an $m \times n$ matrix produces an $n \times n$ matrix, and when left multiplied by an $n \times n$ matrix produces an $m \times n$ matrix. Although this may be true, multiplication between a matrix and a higher-order tensor is not very well established, so it won't be a suitable method in computing the gradient of Ψ.

Instead, a way to work around this problem is by vectorizing the matrices. Vectorization is a linear transformation that converts a matrix into a vector, essentially by "flattening" the matrix. Given an $m \times n$ matrix A, vectorization transforms it into an mn-dimensional vector $\text{vec}(A)$, which is defined as follows:

$$\text{vec}(A) := \begin{bmatrix} a_{1,1} & \cdots & a_{m,1} & a_{1,2} & \cdots & a_{m,2} & \cdots & a_{1,n} & \cdots & a_{m,n} \end{bmatrix}^\mathsf{T}$$

By vectorizing both the input and the output matrix, the function Ψ can be converted to be a vector-valued function, from which its Jacobian can be solved. Let X be an $m \times n$ matrix, where $X = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots \mathbf{x}_n \end{bmatrix}$ and $\mathbf{x}_i = \begin{bmatrix} x_{1,i} & x_{2,i} & \cdots x_{m,i} \end{bmatrix}^\mathsf{T}$ for each $i = 1, \ldots, n$.

Then, Ψ can be rewritten as:

$$\Psi = \begin{bmatrix} \psi_{1,1} & \psi_{1,2} & \cdots & \psi_{1,n} \\ \psi_{2,1} & \psi_{2,2} & \cdots & \psi_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \psi_{n,1} & \psi_{n,2} & \cdots & \psi_{n,n} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^\mathsf{T}\mathbf{x}_1 & \mathbf{x}_1^\mathsf{T}\mathbf{x}_2 & \cdots & \mathbf{x}_1^\mathsf{T}\mathbf{x}_n \\ \mathbf{x}_2^\mathsf{T}\mathbf{x}_1 & \mathbf{x}_2^\mathsf{T}\mathbf{x}_2 & \cdots & \mathbf{x}_2^\mathsf{T}\mathbf{x}_n \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_n^\mathsf{T}\mathbf{x}_1 & \mathbf{x}_n^\mathsf{T}\mathbf{x}_2 & \cdots & \mathbf{x}_n^\mathsf{T}\mathbf{x}_n \end{bmatrix} \tag{4.2}$$

For each $\psi_{i,j}$, its partial derivatives with respect to each $x_{k,l}$ can be solved. Below shows each of these derivative calculations in all cases:

	$i \neq j$	$i = j$
$\psi_{i,j}$	$= \mathbf{x}_i^\mathsf{T}\mathbf{x}_j$	$= \mathbf{x}_i^\mathsf{T}\mathbf{x}_i$
	$= \sum_{k=1}^{m} x_{k,i}x_{k,j}$	$= \sum_{k=1}^{m} x_{k,i}^2$
$\frac{\partial \psi_{i,j}}{\partial x_{k,i}}$	$= x_{k,j}$	$= 2x_{k,i}$
$\frac{\partial \psi_{i,j}}{\partial x_{k,j}}$	$= x_{k,i}$	$= 2x_{k,i}$

Trivially, whenever $l \neq i$ or $l \neq j$, we have $\dfrac{\partial \psi_{i,j}}{\partial x_{k,l}} = 0$.

Using the above results, the gradient of Ψ is ready to be solved using vectorization. Let $\text{vec}(\Psi) : \mathbb{R}^{mn} \to \mathbb{R}^{nn}$ be the vectorized Ψ function:

$$\text{vec}(\Psi) = \begin{bmatrix} \psi_{1,1} & \cdots & \psi_{n,1} & \psi_{1,2} & \cdots & \psi_{n,2} & \cdots & \psi_{1,n} & \cdots & \psi_{n,n} \end{bmatrix}^\mathsf{T}$$

with respect to the input vectorized matrix

$$\text{vec}(X) = \begin{bmatrix} x_{11} & \cdots & x_{m,1} & x_{1,2} & \cdots & x_{m,2} & \cdots & x_{1,n} & \cdots & x_{m,n} \end{bmatrix}^\mathsf{T} \quad .$$

Then, the Jacobian matrix of this function, which will be simply referred to as $\mathbf{J}_\Psi$, is an $nn \times mn$ matrix. This matrix is essentially a "flattened" version of the 4-dimensional tensor $\nabla_X \Psi$. Whenever a matrix is multiplied to $\nabla_X \Psi$, we can first vectorize the input, multiply it with $\mathbf{J}_\Psi$, and unvectorize the output back into a matrix of desired shape. $\mathbf{J}_\Psi$ is given in block matrix form as follows:

$$
\mathbf{J}_{\Psi} = \left[\begin{array}{ccc|ccc|c|ccc}
\frac{\partial \psi_{1,1}}{\partial x_{1,1}} & \cdots & \frac{\partial \psi_{1,1}}{\partial x_{m,1}} & \frac{\partial \psi_{1,1}}{\partial x_{1,2}} & \cdots & \frac{\partial \psi_{1,1}}{\partial x_{m,2}} & & \frac{\partial \psi_{1,1}}{\partial x_{1,n}} & \cdots & \frac{\partial \psi_{1,1}}{\partial x_{m,n}} \\
\frac{\partial \psi_{2,1}}{\partial x_{1,1}} & \cdots & \frac{\partial \psi_{2,1}}{\partial x_{m,1}} & \frac{\partial \psi_{2,1}}{\partial x_{1,2}} & \cdots & \frac{\partial \psi_{2,1}}{\partial x_{m,2}} & & \frac{\partial \psi_{2,1}}{\partial x_{1,n}} & \cdots & \frac{\partial \psi_{2,1}}{\partial x_{m,n}} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots & \ddots & \vdots \\
\frac{\partial \psi_{n,1}}{\partial x_{1,1}} & \cdots & \frac{\partial \psi_{n,1}}{\partial x_{m,1}} & \frac{\partial \psi_{n,1}}{\partial x_{1,2}} & \cdots & \frac{\partial \psi_{n,1}}{\partial x_{m,2}} & & \frac{\partial \psi_{n,1}}{\partial x_{1,n}} & \cdots & \frac{\partial \psi_{n,1}}{\partial x_{m,n}} \\
\hline
\frac{\partial \psi_{1,2}}{\partial x_{1,1}} & \cdots & \frac{\partial \psi_{1,2}}{\partial x_{m,1}} & \frac{\partial \psi_{1,2}}{\partial x_{1,2}} & \cdots & \frac{\partial \psi_{1,2}}{\partial x_{m,2}} & & \frac{\partial \psi_{1,2}}{\partial x_{1,n}} & \cdots & \frac{\partial \psi_{1,2}}{\partial x_{m,n}} \\
\frac{\partial \psi_{2,2}}{\partial x_{1,1}} & \cdots & \frac{\partial \psi_{2,2}}{\partial x_{m,1}} & \frac{\partial \psi_{2,2}}{\partial x_{1,2}} & \cdots & \frac{\partial \psi_{2,2}}{\partial x_{m,2}} & & \frac{\partial \psi_{2,2}}{\partial x_{1,n}} & \cdots & \frac{\partial \psi_{2,2}}{\partial x_{m,n}} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots & \ddots & \vdots \\
\frac{\partial \psi_{n,2}}{\partial x_{1,1}} & \cdots & \frac{\partial \psi_{n,2}}{\partial x_{m,1}} & \frac{\partial \psi_{n,2}}{\partial x_{1,2}} & \cdots & \frac{\partial \psi_{n,2}}{\partial x_{m,2}} & & \frac{\partial \psi_{n,2}}{\partial x_{1,n}} & \cdots & \frac{\partial \psi_{n,2}}{\partial x_{m,n}} \\
\hline
 & \vdots & & & \vdots & & \ddots & & \vdots & \\
\hline
\frac{\partial \psi_{1,n}}{\partial x_{1,1}} & \cdots & \frac{\partial \psi_{1,n}}{\partial x_{m,1}} & \frac{\partial \psi_{1,n}}{\partial x_{1,2}} & \cdots & \frac{\partial \psi_{1,n}}{\partial x_{m,2}} & & \frac{\partial \psi_{1,n}}{\partial x_{1,n}} & \cdots & \frac{\partial \psi_{1,n}}{\partial x_{m,n}} \\
\frac{\partial \psi_{2,n}}{\partial x_{1,1}} & \cdots & \frac{\partial \psi_{2,n}}{\partial x_{m,1}} & \frac{\partial \psi_{2,n}}{\partial x_{1,2}} & \cdots & \frac{\partial \psi_{2,n}}{\partial x_{m,2}} & & \frac{\partial \psi_{2,n}}{\partial x_{1,n}} & \cdots & \frac{\partial \psi_{2,n}}{\partial x_{m,n}} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots & \ddots & \vdots \\
\frac{\partial \psi_{n,n}}{\partial x_{1,1}} & \cdots & \frac{\partial \psi_{n,n}}{\partial x_{m,1}} & \frac{\partial \psi_{n,n}}{\partial x_{1,2}} & \cdots & \frac{\partial \psi_{n,n}}{\partial x_{m,2}} & & \frac{\partial \psi_{n,n}}{\partial x_{1,n}} & \cdots & \frac{\partial \psi_{n,n}}{\partial x_{m,n}}
\end{array}\right]
$$

$$
= \left[\begin{array}{ccc|ccc|c|ccc}
2x_{1,1} & \cdots & 2x_{m,1} & 0 & \cdots & 0 & & 0 & \cdots & 0 \\
x_{1,2} & \cdots & x_{m,2} & x_{1,1} & \cdots & x_{m,1} & & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots & \ddots & \vdots \\
x_{1,n} & \cdots & x_{m,n} & 0 & \cdots & 0 & & x_{1,1} & \cdots & x_{m,1} \\
\hline
x_{1,2} & \cdots & x_{m,2} & x_{1,1} & \cdots & x_{m,1} & & 0 & \cdots & 0 \\
0 & \cdots & 0 & 2x_{1,2} & \cdots & 2x_{m,2} & & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & x_{1,n} & \cdots & x_{m,n} & & x_{1,2} & \cdots & x_{m,2} \\
\hline
 & \vdots & & & \vdots & & \ddots & & \vdots & \\
\hline
x_{1,n} & \cdots & x_{m,n} & 0 & \cdots & 0 & & x_{1,1} & \cdots & x_{m,1} \\
0 & \cdots & 0 & x_{1,n} & \cdots & x_{m,n} & & x_{1,2} & \cdots & x_{m,2} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & 0 & \cdots & 0 & & 2x_{1,n} & \cdots & 2x_{m,n}
\end{array}\right]
$$

$$= \left[\begin{array}{c:c:c:c} \mathbf{x}_1^\mathsf{T} & \mathbf{0}^\mathsf{T} & & \mathbf{0}^\mathsf{T} \\ \mathbf{0}^\mathsf{T} & \mathbf{x}_1^\mathsf{T} & & \mathbf{0}^\mathsf{T} \\ \vdots & \vdots & \cdots & \vdots \\ \mathbf{0}^\mathsf{T} & \mathbf{0}^\mathsf{T} & & \mathbf{x}_1^\mathsf{T} \\ \hdashline \mathbf{x}_2^\mathsf{T} & \mathbf{0}^\mathsf{T} & & \mathbf{0}^\mathsf{T} \\ \mathbf{0}^\mathsf{T} & \mathbf{x}_2^\mathsf{T} & & \mathbf{0}^\mathsf{T} \\ \vdots & \vdots & \cdots & \vdots \\ \mathbf{0}^\mathsf{T} & \mathbf{0}^\mathsf{T} & & \mathbf{x}_2^\mathsf{T} \\ \hdashline \vdots & \vdots & \ddots & \vdots \\ \hdashline \mathbf{x}_n^\mathsf{T} & \mathbf{0}^\mathsf{T} & & \mathbf{0}^\mathsf{T} \\ \mathbf{0}^\mathsf{T} & \mathbf{x}_n^\mathsf{T} & & \mathbf{0}^\mathsf{T} \\ \vdots & \vdots & \cdots & \vdots \\ \mathbf{0}^\mathsf{T} & \mathbf{0}^\mathsf{T} & & \mathbf{x}_n^\mathsf{T} \end{array}\right] + \left[\begin{array}{c:c:c:c} X^\mathsf{T} & \mathbf{0}_{n,m} & \cdots & \mathbf{0}_{n,m} \\ \hdashline \mathbf{0}_{n,m} & X^\mathsf{T} & \cdots & \mathbf{0}_{n,m} \\ \hdashline \vdots & \vdots & \ddots & \vdots \\ \hdashline \mathbf{0}_{n,m} & \mathbf{0}_{n,m} & \cdots & X^\mathsf{T} \end{array}\right] \quad . \tag{4.3}$$

Another way to solve for $\mathbf{J}_\Psi$ is by using vectorization's properties with the Kronecker product and the commutation matrix. If A is an $m \times n$ matrix and B is a $p \times q$ matrix, then the Kronecker product $A \otimes B$ is defined as the $pm \times qn$ block matrix:

$$A \otimes B = \begin{bmatrix} a_{1,1}B & a_{1,2}B & \cdots & a_{1,n}B \\ a_{2,1}B & a_{2,2}B & \cdots & a_{1,n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}B & a_{n,2}B & \cdots & a_{n,n}B \end{bmatrix} \quad .$$

Now, let A, B, and C be $k \times l$, $l \times m$, and $m \times n$ matrices respectively. The following formulation is given using the Kronecker product [25]:

$$\mathrm{vec}(ABC) = (C^\mathsf{T} \otimes A)\mathrm{vec}(B) \quad .$$

From this, the following formulations can also be derived:

$$\begin{aligned} \mathrm{vec}(ABC) &= \mathrm{vec}\big((AB)CI_n\big) = (I_n \otimes AB)\mathrm{vec}(C) \\ &= \mathrm{vec}\big(I_k A(BC)\big) = (C^\mathsf{T}B^\mathsf{T} \otimes I_k)\mathrm{vec}(A) \quad . \end{aligned}$$

Furthermore, the formulations in the case of two matrices is derived as follows:

$$\text{vec}(AB) = \text{vec}(ABI_m) = (I_m \otimes A)\text{vec}(B) \tag{4.4}$$

$$= \text{vec}(I_k AB) = (B^\mathsf{T} \otimes I_k)\text{vec}(A) \quad . \tag{4.5}$$

The commutation matrix $\mathbf{K}^{(m,n)}$ is the $nm \times mn$ matrix that transforms $\text{vec}(A)$ into $\text{vec}(A^\mathsf{T})$ for any $m \times n$ matrix A:

$$\mathbf{K}^{(m,n)}\text{vec}(A) = \text{vec}(A^\mathsf{T}) \quad , \tag{4.6}$$

which can be defined as the following block matrix

$$\mathbf{K}^{(m,n)} = \begin{bmatrix} \mathbf{K}_{1,1} & \mathbf{K}_{1,2} & \cdots & \mathbf{K}_{1,n} \\ \mathbf{K}_{2,1} & \mathbf{K}_{2,2} & \cdots & \mathbf{K}_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{K}_{m,1} & \mathbf{K}_{m,2} & \cdots & \mathbf{K}_{m,n} \end{bmatrix}$$

where each $\mathbf{K}_{i,j}$ is an $n \times m$ sub-matrix whose ji^{th} (**not** ij^{th}) entry is 1 and is 0 everywhere else. The commutation matrix has a special property with the Kronecker product where, given an $m \times n$ matrix A and $p \times q$ matrix B,

$$\mathbf{K}^{(p,m)}(A \otimes B)\mathbf{K}^{(n,q)} = B \otimes A \quad . \tag{4.7}$$

The last notable properties of the commutation matrix are $\mathbf{K}^{(n,m)}\mathbf{K}^{(m,n)} = I_{mn}$ and $\mathbf{K}^{(n,m)\mathsf{T}} = \mathbf{K}^{(m,n)}$, both of which can easily be verified.

Using all of the tools above, we are now ready to symbolically solve for the gradient of the function Ψ. Given an $m \times n$ matrix X and the function Ψ, we have:

$$\begin{aligned}
\Psi &= X^\mathsf{T}X \\
d\Psi &= d(X^\mathsf{T}X) = (dX^\mathsf{T})X + X^\mathsf{T}(dX) \\
\text{vec}(d\Psi) &= \text{vec}\big((dX^\mathsf{T})X\big) + \text{vec}\big(X^\mathsf{T}(dX)\big) \\
&= (X^\mathsf{T} \otimes I_n)\text{vec}(dX^\mathsf{T}) + (I_n \otimes X^\mathsf{T})\text{vec}(dX) && \text{by 4.4 and 4.5} \\
&= (X^\mathsf{T} \otimes I_n)\mathbf{K}^{(m,n)}\text{vec}(dX) + (I_n \otimes X^\mathsf{T})\text{vec}(dX) && \text{by 4.6} \\
&= \Big((X^\mathsf{T} \otimes I_n)\mathbf{K}^{(m,n)} + (I_n \otimes X^\mathsf{T})\Big)\,\text{vec}(dX)
\end{aligned}$$

$$= \mathbf{J}_\Psi \text{vec}(dX) \quad .$$

It can be verified that $\mathbf{J}_\Psi = (X^\mathsf{T} \otimes I_n)\mathbf{K}^{(m,n)} + (I_n \otimes X^\mathsf{T})$ by comparing the results with 4.3.

From here, we can calculate how $\nabla_X \Psi$ behaves when it is left multiplied by a matrix. It is not as important to calculate the right multiplication case since, in back-propagation, the gradients are gathered from back to front by the chain rule. However, it can still be useful to understand the behavior of right-multiplication for other applications. It can also demonstrate properties of multiplication between the higher order tensor $\nabla_X \Psi$ and a matrix, so both will be shown. Let A be an $m \times n$ matrix. Right-multiplying $\mathbf{J}_\Psi$ by $\text{vec}(A)$ gives

$$\begin{aligned}
\mathbf{J}_\Psi \text{ vec}(A) =& \Big((X^\mathsf{T} \otimes I_n)\mathbf{K}^{(m,n)} + (I_n \otimes X^\mathsf{T}) \Big) \text{vec}(A) \\
&= (X^\mathsf{T} \otimes I_n)\mathbf{K}^{(m,n)} \text{vec}(A) + (I_n \otimes X^\mathsf{T})\text{vec}(A) \\
&= (X^\mathsf{T} \otimes I_n)\text{vec}(A^\mathsf{T}) + (I_n \otimes X^\mathsf{T})\text{vec}(A) \\
&= \text{vec}(A^\mathsf{T} X) + \text{vec}(X^\mathsf{T} A) \\
&= \text{vec}(A^\mathsf{T} X + X^\mathsf{T} A) \quad .
\end{aligned}$$

Unvectorizing the result gives the desired output

$$\big(\nabla_X \Psi\big)\, A = A^\mathsf{T} X + X^\mathsf{T} A \quad . \tag{4.8}$$

For left multiplication, let B be an $n \times n$ matrix. Left-multiplying $\text{vec}(B)^\mathsf{T}$ with $\mathbf{J}_\Psi$ gives:

$$\begin{aligned}
\text{vec}(B)^\mathsf{T} \mathbf{J}_\Psi &= \text{vec}(B)^\mathsf{T} \Big((X^\mathsf{T} \otimes I_n)\mathbf{K}^{(m,n)} + (I_n \otimes X^\mathsf{T}) \Big) \\
&= \text{vec}(B)^\mathsf{T} (X^\mathsf{T} \otimes I_n)\mathbf{K}^{(m,n)} + \text{vec}(B)^\mathsf{T} (I_n \otimes X^\mathsf{T}) \quad .
\end{aligned}$$

From here, each part will be solved separately, starting with the second part:

$$\begin{aligned}
\text{vec}(B)^\mathsf{T} (I_n \otimes X^\mathsf{T}) =& \big((I_n \otimes X^\mathsf{T})^\mathsf{T} \text{vec}(B) \big)^\mathsf{T} \\
=& \big((I_n \otimes X)\text{vec}(B) \big)^\mathsf{T} \\
=& \text{vec}(XB)^\mathsf{T} \quad \text{, and} \\
\text{vec}(B)^\mathsf{T} (X^\mathsf{T} \otimes I_n)\mathbf{K}^{(m,n)} =& \big(\mathbf{K}^{(n,m)} (X^\mathsf{T} \otimes I_n)^\mathsf{T} \text{vec}(B) \big)^\mathsf{T} \\
=& \big(\mathbf{K}^{(n,m)} (X \otimes I_n)^\mathsf{T} \mathbf{K}^{(n,n)} \mathbf{K}^{(n,n)} \text{vec}(B) \big)^\mathsf{T}
\end{aligned}$$

$$
\begin{aligned}
&= \big((I_n \otimes X)\mathbf{K}^{(n,n)}\mathrm{vec}(B)\big)^\mathsf{T} && \text{by 4.7} \\
&= \big((I_n \otimes X)\mathrm{vec}(B^\mathsf{T})\big)^\mathsf{T} \\
&= \mathrm{vec}(XB^\mathsf{T})^\mathsf{T} \quad .
\end{aligned}
$$

Therefore, this results in:

$$
\begin{aligned}
\mathrm{vec}(B)^\mathsf{T}\mathbf{J}_\Psi &= \mathrm{vec}(XB^\mathsf{T})^\mathsf{T} + \mathrm{vec}(XB)^\mathsf{T} \\
&= \mathrm{vec}(XB^\mathsf{T} + XB)^\mathsf{T} \quad .
\end{aligned}
$$

Finally, unvectorizing the result gives:

$$
\begin{aligned}
B\left(\nabla_X \Psi\right) &= (XB^\mathsf{T} + XB)^\mathsf{T} \\
&= BX^\mathsf{T} + B^\mathsf{T}X^\mathsf{T} \\
&= (B + B^\mathsf{T})X^\mathsf{T} \quad . \qquad (4.9)
\end{aligned}
$$

Observe that in both cases, symbolically solving for the multiplication with the gradient is much less costly than multiplication with the vectorized Jacobian. Speicifically, left-multiplication with 4.3 has a temporal complexity of $O(mn^3)$, where as following the results given in 4.9 is $O(mn^2)$, reducing the temporal complexity by a factor of n.

4.3 Gradient of Φ

Solving for the gradient of the function $\Phi(X \mid W) = \mathrm{diagonal}(W^\mathsf{T}X)$ for an $n \times n$ matrix input X and an $n \times n$ parameter matrix W is much simpler. Since $\Phi : \mathbb{R}^{n\times n} \to \mathbb{R}^n$ we can imagine that much like before, $\nabla_X \Phi$ (and similarly $\nabla_W \Phi$) is a 3-dimensional tensor of shape $n \times (n \times n)$, where, when left-multiplied by an n-dimensional vector, gives an $n \times n$ matrix output, and when right-multiplied by an $n \times n$ matrix, gives an n-dimensional vector output.

First, let $X = \begin{bmatrix}\mathbf{x}_1 & \mathbf{x}_2 & \cdots \mathbf{x}_n\end{bmatrix}$ and $W = \begin{bmatrix}\mathbf{w}_1 & \mathbf{w}_2 & \cdots \mathbf{w}_n\end{bmatrix}$. This gives

$$
\Phi = \begin{bmatrix}\phi_1 \\ \phi_2 \\ \vdots \\ \phi_n\end{bmatrix} = \begin{bmatrix}\mathbf{w}_1^\mathsf{T}\mathbf{x}_1 \\ \mathbf{w}_2^\mathsf{T}\mathbf{x}_2 \\ \vdots \\ \mathbf{w}_n^\mathsf{T}\mathbf{x}_n\end{bmatrix} \quad .
$$

Then, the scalar-by-vector derivatives of each ϕ_i with respect to each $\mathbf{x}_j$ is given by

$$\frac{\partial \phi_i}{\partial \mathbf{x}_i} = \frac{\partial \mathbf{w}_i^\mathsf{T} \mathbf{x}_i}{\partial \mathbf{x}_i} = \mathbf{w}_i^\mathsf{T}$$

$$\frac{\partial \phi_i}{\partial \mathbf{x}_j} = \frac{\partial \mathbf{w}_i^\mathsf{T} \mathbf{x}_i}{\partial \mathbf{x}_j} = \mathbf{0}^\mathsf{T} \quad \text{whenever} \quad i \neq j.$$

Putting these results together gives us the $n \times nn$ Jacobian matrix $\mathbf{J}_\Psi$ as follows:

$$\mathbf{J}_\Phi = \begin{bmatrix} \frac{\partial \phi_1}{\partial \mathbf{x}_1} & \frac{\partial \phi_1}{\partial \mathbf{x}_2} & \cdots & \frac{\partial \phi_1}{\partial \mathbf{x}_n} \\ \frac{\partial \phi_2}{\partial \mathbf{x}_1} & \frac{\partial \phi_2}{\partial \mathbf{x}_2} & \cdots & \frac{\partial \phi_2}{\partial \mathbf{x}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \phi_n}{\partial \mathbf{x}_1} & \frac{\partial \phi_n}{\partial \mathbf{x}_2} & \cdots & \frac{\partial \phi_n}{\partial \mathbf{x}_n} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^\mathsf{T} & \mathbf{0}^\mathsf{T} & \cdots & \mathbf{0}^\mathsf{T} \\ \mathbf{0}^\mathsf{T} & \mathbf{w}_2^\mathsf{T} & \cdots & \mathbf{0}^\mathsf{T} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}^\mathsf{T} & \mathbf{0}^\mathsf{T} & \cdots & \mathbf{w}_n^\mathsf{T} \end{bmatrix}.$$

Given $\mathbf{v} \in \mathbb{R}^n$, where $\mathbf{v} = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}^\mathsf{T}$, left-multiplying it with $\nabla_X \Phi$ results in:

$$\begin{aligned} \mathbf{v}^\mathsf{T} \left(\nabla_X \Phi \right) &= \text{unvec}(\mathbf{v}^\mathsf{T} \mathbf{J}_\Phi) \\ &= \text{unvec}(\begin{bmatrix} v_1 \mathbf{w}_1^\mathsf{T} & v_2 \mathbf{w}_2^\mathsf{T} & \cdots & v_n \mathbf{w}_n^\mathsf{T} \end{bmatrix})^\mathsf{T} \\ &= \begin{bmatrix} v_1 \mathbf{w}_1^\mathsf{T} \\ v_2 \mathbf{w}_2^\mathsf{T} \\ \cdots \\ v_n \mathbf{w}_n^\mathsf{T} \end{bmatrix} = \mathbf{v} * W^\mathsf{T} \quad . \end{aligned} \tag{4.10}$$

The $*$ operation is defined as a broadcast multiplication, where each row of the matrix is multiplied by the corresponding value of the vector exactly as shown above. For right-multiplication, let A be an $n \times n$ matrix, where $A = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \end{bmatrix}$. Right-multiplying A with $\nabla_X \Phi$ gives:

$$\left(\nabla_X \Phi \right) A = \mathbf{J}_\Phi \text{vec}(A) = \mathbf{J}_\Phi \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_n \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^\mathsf{T} \mathbf{a}_1 \\ \mathbf{w}_2^\mathsf{T} \mathbf{a}_2 \\ \vdots \\ \mathbf{w}_n^\mathsf{T} \mathbf{a}_n \end{bmatrix} = \text{diagonal}(W^\mathsf{T} A) \tag{4.11}$$

Without loss of generality, the derivative of Φ with respect to its parameter matrix W can be calculated in the exact same way, hence $\mathbf{v}^\mathsf{T} \left(\nabla_W \Phi \right) = \mathbf{v} * X^\mathsf{T}$ and $\left(\nabla_W \Phi \right) A = \text{diagonal}(X^\mathsf{T} A)$.

4.4 Back-propagation of the model at Φ and Ψ

Using the gradients found in the previous sections, the back-propagation of the model during training phase at the layers $\Psi(\mathbf{A}) = \mathbf{A}^\mathsf{T}\mathbf{A} = \mathbf{B}$ and $\Phi(\mathbf{B} \mid W_\varphi) = \text{diagonal}(W_\varphi^\mathsf{T}\mathbf{B}) = \mathbf{z}$ will be shown. Refer back to section 3.3 on the full forward-propagation of the model.

Given a cost function C, let $\frac{\partial C}{\partial \mathbf{z}}$ be given as an n-dimensional row-vector $\delta_\mathbf{z}^\mathsf{T}$. Then, the gradients $\frac{\partial C}{\partial \mathbf{W}_\varphi}$, $\frac{\partial C}{\partial \mathbf{B}}$, and $\frac{\partial C}{\partial \mathbf{A}}$ is given using chain rule as follows:

$$\frac{\partial C}{\partial \mathbf{W}_\varphi} = \frac{\partial C}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{W}_\varphi} = \delta_\mathbf{z}^\mathsf{T}\frac{\partial \Phi}{\partial \mathbf{W}_\varphi} = \delta_\mathbf{z}^\mathsf{T}\left(\nabla_{\mathbf{W}_\varphi}\Phi\right)$$

$$\frac{\partial C}{\partial \mathbf{B}} = \frac{\partial C}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{B}} = \delta_\mathbf{z}^\mathsf{T}\frac{\partial \Phi}{\partial \mathbf{B}} = \delta_\mathbf{z}^\mathsf{T}\left(\nabla_\mathbf{B}\Phi\right)$$

$$\frac{\partial C}{\partial \mathbf{A}} = \frac{\partial C}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{A}} = \frac{\partial C}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{B}}\frac{\partial B}{\partial \mathbf{A}} = \delta_\mathbf{z}^\mathsf{T}\frac{\partial \Phi}{\partial \mathbf{B}}\frac{\partial \Psi}{\partial \mathbf{A}} = \delta_\mathbf{z}^\mathsf{T}\left(\nabla_\mathbf{B}\Phi\right)\left(\nabla_\mathbf{A}\Psi\right) \quad .$$

Using the results from 4.9 and 4.10, these then give the following results:

$$\frac{\partial C}{\partial \mathbf{W}_\varphi} = \delta_\mathbf{z}^\mathsf{T}\left(\nabla_{\mathbf{W}_\varphi}\Phi\right) = \delta_\mathbf{z} * \mathbf{B}^\mathsf{T}$$

$$\frac{\partial C}{\partial \mathbf{B}} = \delta_\mathbf{z}^\mathsf{T}\left(\nabla_\mathbf{B}\Phi\right) = \delta_\mathbf{z} * \mathbf{W}_\varphi^\mathsf{T}$$

$$\begin{aligned}\frac{\partial C}{\partial \mathbf{A}} &= \delta_\mathbf{z}^\mathsf{T}\left(\nabla_\mathbf{B}\Phi\right)\left(\nabla_\mathbf{A}\Psi\right) \\ &= \left(\delta_\mathbf{z} * \mathbf{W}_\varphi^\mathsf{T}\right)\left(\nabla_\mathbf{A}\Psi\right) \\ &= \left(\left(\delta_\mathbf{z} * \mathbf{W}_\varphi^\mathsf{T}\right)^\mathsf{T} + \left(\delta_\mathbf{z} * \mathbf{W}_\varphi^\mathsf{T}\right)\right)\mathbf{A}^\mathsf{T} \quad .\end{aligned}$$

With the right algorithm, each of the gradient calculations should be fast and numerically stable, with a smooth gradient surface for training the model.

Chapter 5. Method

In this chapter, the specifics of how the model was trained will be presented. First, there will be a discussion about how the data was curated, prepared, and processed. Then, the details of the training procedure will be shared, including the cost function and the optimizer. Finally, results of the training will be shown.

5.1 Dataset

The data used to train the model is from a book dataset, where texts from many different books were collected into a single corpus. The data is prepared by assigning each word or symbol with a token. Unique tokens are gathered from the corpus to create a vocabulary set. Then, each element of the vocabulary is assigned an integer value to create a Python dictionary object. Initially the data consisted of 40 million lines of English sentences with over a million unique vocabulary tokens. In order to reduce the number of tokens, several preprocessing strategies were used. Below is a list of preprocessing that was done to reduce the size of the vocabulary:

- replace all numbers with a unique number token
- convert all % and $ symbols to words 'percent' and 'dollars'
- add spaces around all non-letter symbols
- remove all remaining non-punctuation symbols
- remove all low frequency tokens (occurs n times or less throughout the dataset)
- normalize all texts by changing all letters to lowercase
- remove all sentences that are too long (over `max_len` number of tokens)

At first, n was set to 10 and `max_len` was set to 50. After processing the data, the size of vocabulary was reduced from over a million to just over 73 thousand. However, training a large model with this size was still unsuccessful, so further processing was done to reduce both the size of the vocabulary and the data. In order to further reduce the size and normalize the data, more dramatic approaches were used:

- `max_len` was set from 50 to 20.
- Using the `PyEnchant` library, an English dictionary was used to remove any lines that contained words not in the dictionary. This removed all lines with uncommon proper nouns, and odd spelling of words.
- Only a tenth of the data was kept.
- Vocabulary was further reduced by setting n to 20.

In the end, the vocabulary size reduced down to 16,138 and the number of lines to just under 1.3 million.

The processed dataset was then split into training, validation, and testing sets. First, 10 thousand lines were randomly selected to be the testing set. This dataset is reserved for the end of all model trainings in order to evaluate the performance of the final model. The remaining data was split into training and validation set, with 90% of the data used for training and 10% of the data for validation. The training dataset is used for the actual training of the model with an optimizer, and the model is evaluated after each epoch with the validation dataset to track the progress of the training. Because neural network models can often overfit to the training data, validation data is used to evaluate the model's ability to generalize to new data as it trains.

Below are some example sentences from the dataset after preprocessing:

```
i finally got my shot at some screen time .
he looked tired and worn , which troubled me more than anger would have .
i stumbled down to the ground , still making a spectacle of myself .
```

```
in turn they whispered their name , acknowledging their presence .
i spun around and backed up , unnerved by his angry look .
she climbed back up the pool steps .
it had been months since the last visit .
bored and victorious , what more could a man ask for ?
she s resting , so i do nt want you to disturb her .
the demon inside you .
so he provides a remedy for your rightful passions .
but , hey , it s a million times better than the institute !
listening carefully could produce two clues .
call me when you can , and be safe .
i ca nt connect at all in this wretched valley .
```

5.2 Opimizer and Cost Function

Since the majority of the model is the transformer, it was ideal to follow much of the training procedure from the original paper [46]. Optimization was done using the Adam [20] optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.98$, and $\epsilon = 10^{-9}$. The learning rate η varied over the course of the training according to the formula

$$\eta = d^{-0.5}\min(\mathtt{i}^{-0.5}, \mathtt{i}\,\omega^{-1.5})$$

where d is the dimension of the model, $\mathtt{i}$ is the current number of iterations of training, and ω is a hyperparameter for the number of warm-up steps. The learning rate η increases linearly until $\mathtt{i} = \omega$, from which point η decreases gradually proportional to $\mathtt{i}^{-0.5}$. ω was set to 2000.

For the cost function, Kullback-Leibler divergence, D_{KL}, is used, which is defined as:

$$D_{\mathrm{KL}}(P||Q) := \sum_{x \in X} P(x)\log(\frac{P(x)}{Q(x)}) \tag{5.1}$$

where P represents the target output probability distribution and Q is the predicted output probability distribution produced by the model. In the context of language model training,

the probability space is a discrete space over all vocabulary words, where the desired output distribution, P, has a concentrated mass at the index that maps to the correct next token. In practice, this usually means that 100% of the mass is at the correct token and is zero everywhere else. For this training, label smoothing strategy was used, where the confidence of the correct token is subtracted by a smoothing value. The remaining mass is then spread out evenly across all other tokens. This causes the model to learn to be more unsure about the next token prediction, but helps with regularization during evaluation phase. The smoothing value was set to 0.1.

5.3 Training

Training was done using a single NVIDIA GeForce RTX 2080 Ti GPU for 20 epochs over the training set. Each training session was done 10 epochs at a time, and after each epoch, validation loss was computed using the validation set, and a checkpoint of the model was saved. Two full training sessions were done before the validation loss started to increase, at which point a saved model with minimum validation loss was chosen to be the final model for testing. The graph from Figure 5.1 shows both the train and validation loss over the entire training, where the validation loss achieved a minimum score of $D_{\mathrm{KL}} = 0.5587$. In total, the training took about 26 hours.

5.4 Evaluation

Using the final model, testing was done using 10,000 sentences from the test set that had been reserved during model training and selection phase. The model achieved a loss value of $D_{\mathrm{KL}} = 0.5478$ averaged over the test set. To evaluate the model's performance on decoding without the use of the target inputs, two different decoding strategies are used: greedy decoding and beam search decoding.

Greedy decoding is a naïve method where the token with the highest likelihood is chosen

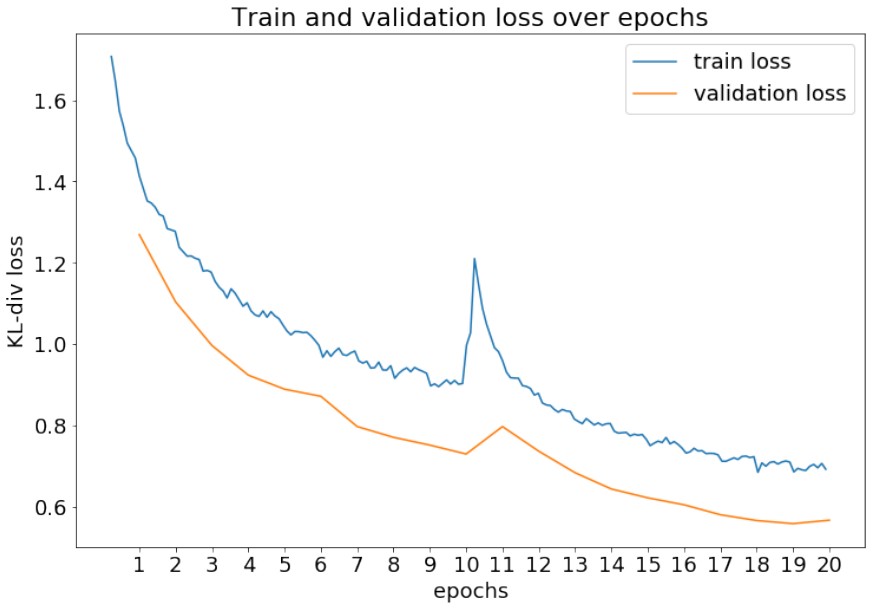


Figure 5.1: Graph of train and validation loss. The validation loss is lower because the model is set to evaluation mode, turning off all regularizations such as normalization layers and dropout layers. The spike at epoch 10 signifies a new training session with warm-up steps.

at each generation. More precisely, given the latent vector $\mathbf{z}$, the inputs for the transformer decoder are the filter vectors $\mathbf{F}$ and $E(\tau_0, ..., \tau_k)$ for the k^{th} decoding step. Then, the generator produces the output distribution $p(\tau_{k+1} \mid \tau_0, \ldots, \tau_k)$. The greedy decoding method chooses

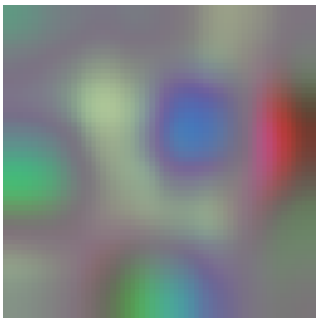
$$\tau_{k+1} = \text{argmax}\Big(p(\tau_{k+1} \mid \tau_0, \ldots, \tau_k)\Big)$$

for the next token τ_{k+1}, which is then appended to the end of the current sequence as the next decoder input. This is repeated recursively until $\tau_{k+1} = \texttt{EOS}$, which signifies that the end of the sentence has been reached.

Beam search is a breadth-first search algorithm that builds a search tree, where, at each level, β best states are stored and the rest are pruned. The stored states are then expanded further in the next level and pruned in the same way. In context of language generation, the best states refer to the most probable sequences for each generation. Observe that beam search with $\beta = 1$ is equivalent to greedy decoding. Given a trained model and a latent vector $\mathbf{z}$, the following code snippet shows the details of the beam search algorithm. The code is simplified for readability.

```python
def Beam_Search(z, model, beta=5, max_len=20)
    # Obtain the convolutional filters from z
    filters = model.extract_filters(z)
    # Initialize the first token using the SOS token
    ys = SOS
    # Initialize a list to store the sequences
    top_seqs = [(ys, 0, False)] # (sequence, log-prob, is EOS)
    for i in range(max_len - 1):
        # Initialize a new list to store next generations
        new_seqs = []
        for seq, score, is_eos in top_seqs:
            # Keep and skip if sequence reached eos,
            if is_eos:
                new_seqs.append((seq, score, is_eos))
                continue
            decoder_out = model.decode(seq, filters)
            prob_dist = model.generator(decoder_out)
            log_probs, words = torch.topk(prob_dist, beta)
            # Update and store next generation
            for log_prob, word in zip(log_probs, words):
                new_seq = torch.cat(seq, word)
                new_score = score + log_prob
                new_is_eos = True if word == EOS else False
                new_seqs.append((new_seq, new_score, new_is_eos))
        # Sort the sequences by log probability score
        top_seqs = sorted(new_seqs, key=lambda x: x[1], reverse=True)
        # Only keep beta sequences
        top_seqs = top_seqs[:beta]
        # If all current sequences reached EOS, break and return
        if sum(eos for _, _, eos in top_seqs) == beta:
            break
    return top_seqs
```

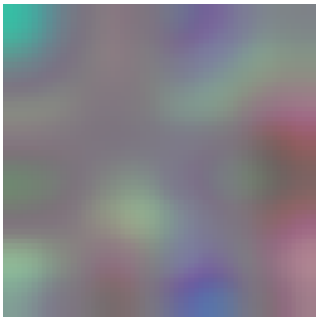
Below are some of the sentences from the test set, the image representations of the sentences, and the decoded outputs of the model using greedy decoding and beam search decoding. The beam width β for all beam search decoding is set to 5, although not all 5 candidates are shown for all instances. The beam search outputs are ordered from most likely to least.



Input sentence:
`well , maybe this once .`

Greedy decode output:
`well , maybe this once .`

Beam search outputs:
`well , maybe this once .`
`well , maybe this time .`
`well , maybe this afternoon .`



Input sentence:
`oh , there are some things i can work with them on .`

Greedy decode output:
`oh , there are some things i can work with them on .`

Top beam search output:
`oh , there are some things i can work with them on .`

Input sentence:
`he based that on the number of times a cell could divide .`

Greedy decode output:
`he on that page the years of answering a force would sell youth .`

Top beam search output:
`he regained that number on the force of a student who lay .`

The following are a few more interesting examples from the test set. The first two shown are instances where the beam search clearly outperforms greedy decoding in terms of exact word-for-word matching with the input sentence.

Input sentence:
`if i am to train you , you re going to have to trust me .`

Greedy decode output:
`if you are to train , i m going to have to trust me i .`

Top beam search output:
`if i am to train you , you re going to have to trust me .`

Input sentence:
`he cheated on me with other women while we were married .`

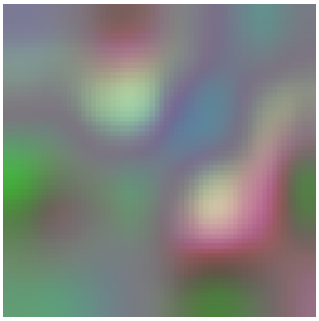
Greedy decode output:
`he dated on me after other kids we went with harm .`

Top beam search output:
`he cheated on me with other women while we were married .`

The next two examples show all five decoded outputs of the beam search decoding. These examples demonstrate that words that are semantically similar have higher likelihoods in the output distribution $p(\tau_{k+1} \mid \tau_0, \ldots, \tau_k)$.

Input sentence:
`sure , i replied softly .`

Beam search outputs:
`sure , i replied softly .`
`sure , i softly replied .`
`sure , i murmured softly .`
`sure , i replied quietly .`
`sure , i replied gently .`



Input sentence:
`they disappeared up the drive .`

Beam search outputs:
`they disappeared up the drive .`
`they disappeared up the road .`
`they disappeared up the hallway .`
`they disappeared up the hospital .`
`they disappeared up the walk .`

The following example shows a case where the greedy decoding and the top beam search decoding are both not very semantically close to the input sentence, but the less likely candidates from the beam search decoding are.

Input sentence:
`i ll go get the other drinks , i said smiling as i turned back toward the kitchen .`

Greedy decode output:
`i ll go get the other plate , i said as i walked back toward the tea meeting .`

Beam search outputs:
`i ll go get the other evening , i said as i leaned back toward the waitress .`
`i ll go get the other drinks , i said as i walked back toward the kitchen smiling .`
`i ll go get the other drinks , i said as i walked back toward the kitchen .`

The last example shows a case where the decoded sentences are semantically similar in interesting ways with the input sentence.

Input sentence:
`a kiss that still made the roots of her hair tingle .`

Greedy decode output:
`a kiss that still made the fabric of her scalp shiver .`

Beam search outputs:
`a kiss that still made the softness of her hair quiver .`
`a kiss that still made the softness of her flesh curl .`

Chapter 6. Experiments and Analysis

Using the trained model, different experiments were conducted to find interesting properties about the latent space and the encoded vectors. Understanding the model's encoding and decoding capabilities can help us understand how the model can be used for further applications and its limitations. Throughout this chapter, let $\mathbf{E} : \mathcal{L} \to \mathbb{R}^d$ be defined as the encoding half of the entire model, $\mathbf{E} := \Phi \circ \Psi \circ \mathcal{T}_{\mathbf{e}} \circ E$, and let $\mathbf{D} : \mathbb{R}^d \to \mathcal{L}$ be defined as the top beam search decoding of the latent vector with $\beta = 5$. With a well trained model, $\mathbf{E}$ and $\mathbf{D}$ are theoretically the inverse functions of each other. Therefore, we first make sure $\mathbf{D}\big(\mathbf{E}(\mathbf{s})\big) \approx \mathbf{s}$ whenever an example sentence $\mathbf{s} \in \mathcal{L}$ is chosen.

6.1 Gaussian Mixture Model Fitting

One of the underlying assumptions that led to the model design choices is that Gaussian distribution should not be used as the prior distribution of large language data. To test this assumption, a Gaussian Mixture Model was fitted on the set of latent vectors obtained from the test set. A Gaussian mixture model fitted on a distribution of data will attempt to estimate the data distribution with multiple Gaussian components, each i^{th} component weighted with some scalar α_i with its own mean $\boldsymbol{\mu}_i$ and covariance matrix $\boldsymbol{\Sigma}_i$. Specifically, the prior distribution is given by

$$p(\boldsymbol{\theta}) = \sum_{i=1}^{K} \alpha_i \mathcal{N}\big(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\big)$$

The values of the parameter for each component is updated conditioned on the data $\mathbf{x}$ by using the Expectation-Maximization algorithm and will result in a posterior distribution

$$p(\boldsymbol{\theta} \mid \mathbf{x}) = \sum_{i=1}^{K} \tilde{\alpha}_i \mathcal{N}\big(\tilde{\boldsymbol{\mu}}_i, \tilde{\boldsymbol{\Sigma}}_i\big)$$

In Python, this can be done easily with the help of the Scikit-Learn library [31].

In this experiment, a Gaussian mixture model with 20 components was used to fit the latent vectors. Afterwards, vectors were sampled from each Gaussian and decoded. For each

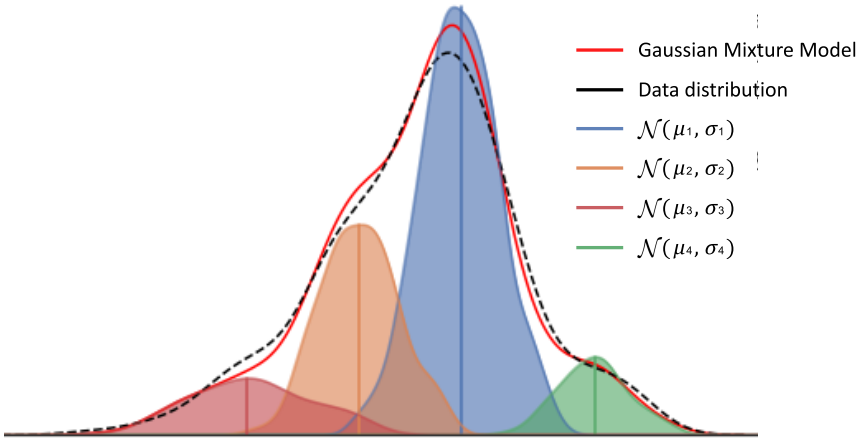


Figure 6.1: Gaussian mixture model can approximate a given distribution [43]

component i, the sampling process can be described as

$$\mathbf{z}_i \sim \mathcal{N}\big(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\big)$$

$$\mathbf{D}(\mathbf{z}_i) \rightarrow \text{decoded sentence}$$

The following shows some of the sampled vectors from three different Gaussian components decoded into sentences. The scores are the log-likelihood of the sentences defined as

$$\texttt{score}\big(\mathbf{D}(\mathbf{z})\big) := \log\big(p(\tau_1, \ldots, \tau_\ell)\big)$$

```
Component 1
Sentence 1.1 score:  -14.57
Decoded:  i do , but i just arrived a small oven from another jet .

Sentence 1.2 score:  -18.10
Decoded:  i pushed against the air , not sure he could notice sorrow in me of a intercom .

Sentence 1.3 score:  -16.01
Decoded:  i wrapped my head to kiss her , when things she picked away from his father .

Sentence 1.4 score:  -24.49
Decoded:  i laughed in front of her lungs so softly i pushed myself forward , and immediately
toward his spell .

Sentence 1.5 score:  -12.55
```

```
Decoded:  i grabbing their food for the point so that it creaked back .
```

Component 5

```
Sentence 5.1 score:  -16.63
Decoded:  what have yourself in there , do nt she come !

Sentence 5.2 score:  -11.71
Decoded:  was huh there focus to world ?

Sentence 5.3 score:  -12.52
Decoded:  how could this baby how your parents we re with him ?

Sentence 5.4 score:  -18.36
Decoded:  are you wondering why to do it down the family time in home ?

Sentence 5.5 score:  -12.81
Decoded:  you know who was everything , was her ?
```

Component 18

```
Sentence 18.1 score:  -14.47
Decoded:  the rest might change in any amount of worlds .

Sentence 18.2 score:  -11.44
Decoded:  the houses shall look complete , both of prison .

Sentence 18.3 score:  -6.44
Decoded:  men to the field too .

Sentence 18.4 score:  -6.97
Decoded:  the prince will leave very somewhat surprised .

Sentence 18.5 score:  -8.49
Decoded:  the last skill disappear .
```

From the decoded sentence, it can be observed that when a Gaussian mixture model is used, the type of sentences that cluster around each of the components are largely influenced by the first few words of the sentence. However, most of the decoded sentences are nonsensical. This is reflected by the sentence scores, which can be used as a metric to determine how

likely a given vector can be decoded as a sensible sentence. This metric is used for further experiments to measure the viability of the decoded sentences.

6.2 Linear Interpolation

Let $\mathbf{s}_1, \mathbf{s}_2 \in \mathcal{L}$ and let $\mathbf{E}(\mathbf{s}_1) = \mathbf{z}_1$, $\mathbf{E}(\mathbf{s}_2) = \mathbf{z}_2$. One simple experiment that can be done using these two vectors is a linear interpolation

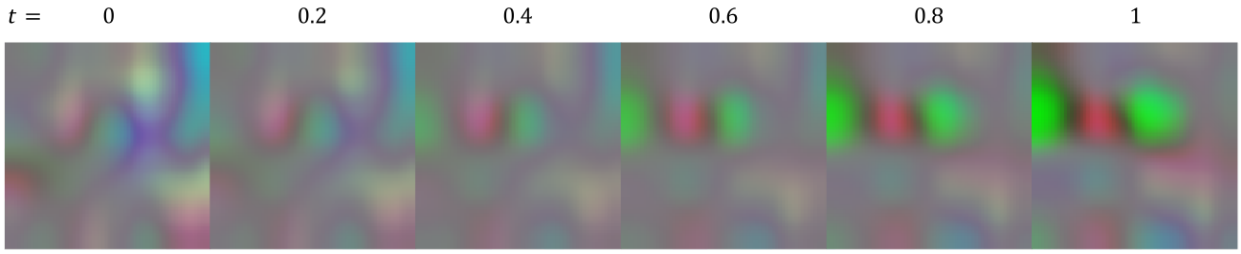
$$\mathbf{z}_t = (1-t)\,\mathbf{z}_1 + t\,\mathbf{z}_2 \quad \text{for} \quad 0 \leq t \leq 1 \quad .$$

By decoding $\mathbf{z}_t$ for different values of t, it can be observed whether $\mathbf{z_t}$ shows a smooth transition between the sentences $\mathbf{s}_1$ and $\mathbf{s}_2$. The linear interpolation experiment was first conducted with the sentences

$$\mathbf{s}^1 = \texttt{he stayed completely still for the next moment .}$$

$$\mathbf{s}^2 = \texttt{then she continued as calmly as she could .}$$

The following shows the images and the decoded sentences of $\mathbf{z}_t$ for varying values of t:



t	$\mathbf{D}(\mathbf{z}_t)$	score
0	**he stayed completely still for the next moment .**	-1.09
0.2	he stayed completely still for the next then .	-4.12
0.4	she stayed now as his head barely i wondered .	-9.13
0.6	then she stayed silent as he ever watched .	-6.92
0.8	then she continued quietly as she could at my wounds .	-5.11
1	**then she continued as calmly as she could .**	-0.99

The result shows a smooth transition of the images, and the token-wise transition of

$\mathbf{D}(\mathbf{z}_t)$ seems acceptable but don't make a lot of sense grammatically or semantically, which is evident from the increased log-probability scores of the transition sentences. This is likely because the two sentences are not similar enough, so the experiment was repeated with more similar pairs of sentences. Below shows a linear interpolation between the sentences

$$\mathbf{s}^3 = \texttt{my mom was upset when she saw me .}$$

$$\mathbf{s}^4 = \texttt{my dad is happy to see you .}$$

t	$\mathbf{D}(\mathbf{z}_t)$	score
$0 - 0.2$	**my mom was upset when she saw me .**	-0.76
0.5	my mom was happy to see me now .	-5.94
0.6	my dad is upset to see you happy .	-5.66
0.7	my dad is happy to see you upset .	-2.37
$0.8 - 1$	**my dad is happy to see you .**	-0.99

These results show a smoother transition between the two sentences and the transitions are grammatically correct as well. This suggest a sparsity of vectors in the latent space that are likely to be sentences.

Finally, a linear combination between three different sentences was done using words of emotion.

$$\begin{aligned}
\mathbf{s}_{\text{happy}} &= \texttt{i am happy .} & \mathbf{E}\big(\mathbf{s}_{\text{happy}}\big) &= \mathbf{z}_{\text{happy}} \\
\mathbf{s}_{\text{disappointed}} &= \texttt{i am disappointed .} & \mathbf{E}\big(\mathbf{s}_{\text{disappointed}}\big) &= \mathbf{z}_{\text{disappointed}} \\
\mathbf{s}_{\text{excited}} &= \texttt{i am excited .} & \mathbf{E}\big(\mathbf{s}_{\text{excited}}\big) &= \mathbf{z}_{\text{excited}}
\end{aligned}$$

Below is a few combinations that produced interesting results:

$$\mathbf{D}\big(0.5\,\mathbf{z}_{\text{disappointed}} + 0.5\,\mathbf{z}_{\text{excited}}\big) = \texttt{i am angry .}$$

$$\mathbf{D}\big(-0.7\,\mathbf{z}_{\text{happy}} + 0.8\,\mathbf{z}_{\text{disappointed}} + 0.9\,\mathbf{z}_{\text{excited}}\big) = \texttt{i am annoyed .}$$

$$\mathbf{D}\big(-\mathbf{z}_{\text{happy}} + \mathbf{z}_{\text{disappointed}} + \mathbf{z}_{\text{excited}}\big) = \texttt{i am humor curious .}$$

These results show that emotions may lie in a particular subset of the latent space, which may be traversed to produce different emotion for the output sentence.

6.3 Vector Operations

The Word2Vec [28] model showed that it is possible to perform vector operations of encoded words to manipulate the semantics of words. One famous example is `king - man + woman` = `queen` which is the expected and desired output. In this section, similar experiments are performed with encoded sentences.

6.3.1 Subject Pronoun Manipulation. For the first experiment, the goal is to manipulate the subject pronouns. The following sentences are identical except for their subject pronouns:

$$\begin{aligned}
\mathbf{s}^1_{\text{he}} &= \texttt{he stayed completely still for the next moment .} & \mathbf{E}\big(\mathbf{s}^1_{\text{he}}\big) &= \mathbf{z}^1_{\text{he}} \\
\mathbf{s}^1_{\text{she}} &= \texttt{she stayed completely still for the next moment .} & \mathbf{E}\big(\mathbf{s}^1_{\text{she}}\big) &= \mathbf{z}^1_{\text{she}} \\
\mathbf{s}^1_{\text{they}} &= \texttt{they stayed completely still for the next moment .} & \mathbf{E}\big(\mathbf{s}^1_{\text{they}}\big) &= \mathbf{z}^1_{\text{they}} \\
\mathbf{s}^1_{\text{I}} &= \texttt{i stayed completely still for the next moment .} & \mathbf{E}\big(\mathbf{s}^1_{\text{I}}\big) &= \mathbf{z}^1_{\text{I}} \\
\mathbf{s}^1_{\text{you}} &= \texttt{you stayed completely still for the next moment .} & \mathbf{E}\big(\mathbf{s}^1_{\text{you}}\big) &= \mathbf{z}^1_{\text{you}} \\
\mathbf{s}^1_{\text{we}} &= \texttt{we stayed completely still for the next moment .} & \mathbf{E}\big(\mathbf{s}^1_{\text{we}}\big) &= \mathbf{z}^1_{\text{we}}
\end{aligned}$$

Since each of these sentences are identical except for the subject pronouns, the difference of two sentences would represent the difference in the pronouns used. For example, since 'I' is a first-person singular pronoun and 'we' is a first-person plural pronoun, the difference $\mathbf{z}^1_{\text{we}} - \mathbf{z}^1_{\text{I}}$ will represent the plurality element. Then, the expected outcome when adding this difference to $\mathbf{z}^1_{\text{he}}$, a third-person singular pronoun, would be a vector close to $\mathbf{z}^1_{\text{they}}$, a third-person plural pronoun. The following shows the result of the experiment.

$$\mathbf{D}\big(\mathbf{z}^1_{\text{we}} - \mathbf{z}^1_{\text{I}} + \mathbf{z}^1_{\text{he}}\big) = \texttt{they stayed completely still for the next moment .}$$

$$\mathbf{D}\big(\mathbf{z}^1_{\text{they}} - \mathbf{z}^1_{\text{he}} + \mathbf{z}^1_{\text{I}}\big) = \texttt{completely we stayed still for the next moment .}$$

$$\mathbf{D}\big(\mathbf{z}^1_{\text{they}} - \mathbf{z}^1_{\text{he}} + \mathbf{z}^1_{\text{you}}\big) = \texttt{you stayed completely still for their next time .}$$

Although there were some changes to the sentence itself for some, the subject pronouns of the output were changed as intended. The reverse of this experiment was conducted and

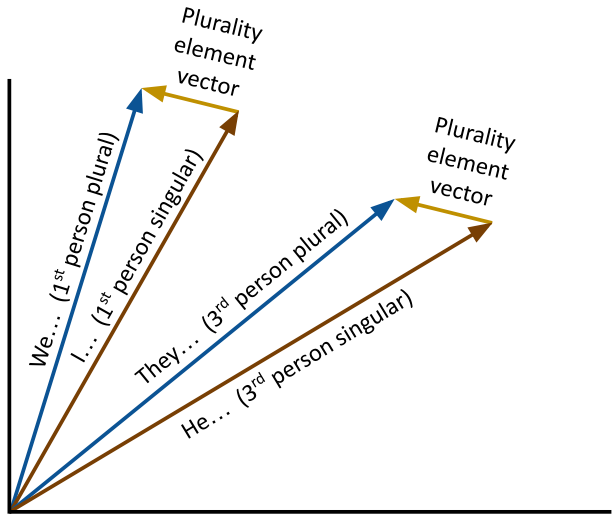


Figure 6.2: Illustration of vector operations between two pairs of vectors and their difference.

showed a similar result.

$$\mathbf{D}\left(\mathbf{z}^1_{\text{I}} - \mathbf{z}^1_{\text{we}} + \mathbf{z}^1_{\text{they}}\right) = \texttt{completely he stayed still for her next moment .}$$

$$\mathbf{D}\left(\mathbf{z}^1_{\text{he}} - \mathbf{z}^1_{\text{they}} + \mathbf{z}^1_{\text{we}}\right) = \texttt{i stayed completely still for the next moment .}$$

Similar experiment was conducted but instead adding the difference of sentences in $\mathbf{s}^1$ to a new sentence $\mathbf{s}^2$

$$\mathbf{s}^2_{\text{he}} = \texttt{then he continued as calmly as he could .} \quad \mathbf{E}\left(\mathbf{s}^2_{\text{he}}\right) = \mathbf{z}^2_{\text{he}}$$

$$\mathbf{D}\left(\mathbf{z}^1_{\text{they}} - \mathbf{z}^1_{\text{he}} + \mathbf{z}^2_{\text{he}}\right) = \texttt{then they continued as calmly as he could .}$$

$$\mathbf{D}\left(\mathbf{z}^1_{\text{we}} - \mathbf{z}^1_{\text{i}} + \mathbf{z}^2_{\text{he}}\right) = \texttt{then they continued as he calmly as we could .}$$

$$\mathbf{D}\left(\mathbf{z}^1_{\text{she}} - \mathbf{z}^1_{\text{he}} + \mathbf{z}^2_{\text{he}}\right) = \texttt{then she continued as calmly as he could .}$$

The subject pronouns changed as desired, but subject-object agreement wasn't achieved, which suggest that the vectors for the subject and the object may be parts of a separate subset of the latent space.

6.3.2 Tense Manipulation. The following experiment shows manipulation of tense for the predicate. The sentences used and the result of the vector operations are shown below.

$$\begin{aligned}
\mathbf{s}^3_{\text{present}} &= \texttt{i am there .} & \mathbf{E}\big(\mathbf{s}^3_{\text{present}}\big) &= \mathbf{z}^3_{\text{present}} \\
\mathbf{s}^3_{\text{future}} &= \texttt{i will be there .} & \mathbf{E}\big(\mathbf{s}^3_{\text{future}}\big) &= \mathbf{z}^3_{\text{future}} \\
\mathbf{s}^3_{\text{past}} &= \texttt{i was there .} & \mathbf{E}\big(\mathbf{s}^3_{\text{past}}\big) &= \mathbf{z}^3_{\text{past}} \\
\mathbf{s}^4_{\text{present}} &= \texttt{i am going with you .} & \mathbf{E}\big(\mathbf{s}^4_{\text{present}}\big) &= \mathbf{z}^4_{\text{present}}
\end{aligned}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{future}} + \mathbf{z}^3_{\text{past}}\big) = \texttt{i m there .}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{future}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^4_{\text{present}}\big) = \texttt{i will be going with you .}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{past}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^4_{\text{present}}\big) = \texttt{i was going with you .}$$

Using the sentences from s^1, further experiment is done to manipulate both the subject pronoun and the tense at once.

$$\mathbf{D}\big(\mathbf{z}^1_{\text{they}} - \mathbf{z}^1_{\text{i}} + \mathbf{z}^4_{\text{present}}\big) = \texttt{they are going with you .}$$

$$\mathbf{D}\big(\mathbf{z}^1_{\text{they}} - \mathbf{z}^1_{\text{i}} + \mathbf{z}^3_{\text{future}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^4_{\text{present}}\big) = \texttt{they will be going with you .}$$

$$\mathbf{D}\big(\mathbf{z}^1_{\text{they}} - \mathbf{z}^1_{\text{i}} + \mathbf{z}^3_{\text{past}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^4_{\text{present}}\big) = \texttt{they was going with you .}$$

The result shows that both the subject and the predicate changed as intended. Although the first result achieved subject-verb agreement, the third did not. This was fixed by adjusting the weight of the differences, as shown below:

$$\mathbf{D}\big(\mathbf{z}^1_{\text{they}} - \mathbf{z}^1_{\text{i}} + 0.5(\mathbf{z}^3_{\text{past}} - \mathbf{z}^3_{\text{present}}) + \mathbf{z}^4_{\text{present}}\big) = \texttt{they were going with you .}$$

Further experiment was done using the verbs 'run' and 'ran'. The results weren't as consistent but produced some interesting outputs.

$$\begin{aligned}
\mathbf{s}^5_{\text{present}} &= \texttt{i run in the morning .} & \mathbf{E}\big(\mathbf{s}^5_{\text{present}}\big) &= \mathbf{z}^5_{\text{present}} \\
\mathbf{s}^5_{\text{past}} &= \texttt{i ran in the morning .} & \mathbf{E}\big(\mathbf{s}^5_{\text{past}}\big) &= \mathbf{z}^5_{\text{past}}
\end{aligned}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{future}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^5_{\text{present}}\big) = \texttt{i ll run in the morning being .}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{past}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^5_{\text{present}}\big) = \texttt{i run in the morning .}$$

$$\mathbf{D}\big(1.2(\mathbf{z}^3_{\text{past}} - \mathbf{z}^3_{\text{present}}) + \mathbf{z}^5_{\text{present}}\big) = \texttt{i slid in the morning run .}$$

$$\mathbf{D}\big(\mathbf{z}^5_{\text{past}} - \mathbf{z}^5_{\text{present}} + \mathbf{z}^3_{\text{present}}\big) = \texttt{i were there .}$$

6.3.3 Negation. This experiment shows negations of sentences using the word 'not'.

$$\mathbf{s}^3_{\text{not}} = \texttt{i am not there .} \quad \mathbf{E}\big(\mathbf{s}^3_{\text{not}}\big) = \mathbf{z}^3_{\text{not}}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{not}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^5_{\text{future}}\big) = \texttt{i will not be there .}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{not}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^5_{\text{past}}\big) = \texttt{i was not there .}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{not}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^5_{\text{present}}\big) = \texttt{i run not in the morning .}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{not}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^4_{\text{present}}\big) = \texttt{i am not going with you .}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{not}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^1_{\text{he}}\big) = \texttt{he stayed completely not still for the next moment .}$$

$$\mathbf{D}\big(\mathbf{z}^3_{\text{not}} - \mathbf{z}^3_{\text{present}} + \mathbf{z}^2_{\text{he}}\big) = \texttt{then he continued not as calmly as he could .}$$

When there isn't an auxiliary (be, will, can, do, etc.) verb, negation is done by inserting the 'do' verb. Although the model wasn't able to achieve that, it still placed 'not' in the grammatically correct places.

6.3.4 Statements vs. Questions. Next, experiments were conducted to see whether statements could be turned into questions and vice-versa.

$$\begin{aligned}
\mathbf{s}^6_{\text{statement}} &= \texttt{you can go home .} & \mathbf{E}\big(\mathbf{s}^6_{\text{statement}}\big) &= \mathbf{z}^6_{\text{statement}} \\
\mathbf{s}^6_{\text{question}} &= \texttt{can you go home ?} & \mathbf{E}\big(\mathbf{s}^6_{\text{question}}\big) &= \mathbf{z}^6_{\text{question}} \\
\mathbf{s}^3_{\text{question}} &= \texttt{am i there ?} & \mathbf{E}\big(\mathbf{s}^3_{\text{question}}\big) &= \mathbf{z}^3_{\text{question}} \\
\mathbf{s}^4_{\text{question}} &= \texttt{am i going with you ?} & \mathbf{E}\big(\mathbf{s}^4_{\text{question}}\big) &= \mathbf{z}^4_{\text{question}}
\end{aligned}$$

$$\mathbf{D}\big(\mathbf{z}^6_{\text{question}} - \mathbf{z}^6_{\text{statement}} + \mathbf{z}^3_{\text{present}}\big) = \texttt{am i there ?}$$

$$\mathbf{D}\big(\mathbf{z}^6_{\text{question}} - \mathbf{z}^6_{\text{statement}} + \mathbf{z}^3_{\text{future}}\big) = \texttt{will i be there ?}$$

$$\mathbf{D}\big(\mathbf{z}^6_{\text{question}} - \mathbf{z}^6_{\text{statement}} + \mathbf{z}^3_{\text{past}}\big) = \texttt{i was there ?}$$

$$\mathbf{D}\big(1.5(\mathbf{z}^6_{\text{question}} - \mathbf{z}^6_{\text{statement}}) + \mathbf{z}^3_{\text{past}}\big) = \texttt{was i there ?}$$

$$\mathbf{D}\big(\mathbf{z}^6_{\text{question}} - \mathbf{z}^6_{\text{statement}} + \mathbf{z}^4_{\text{present}}\big) = \texttt{am i going with you ?}$$

$$\mathbf{D}\big(\mathbf{z}^6_{\text{statement}} - \mathbf{z}^6_{\text{question}} + \mathbf{z}^3_{\text{question}}\big) = \texttt{i am there .}$$

$$\mathbf{D}\big(\mathbf{z}^6_{\text{question}} - \mathbf{z}^6_{\text{statement}} + \mathbf{z}^4_{\text{question}}\big) = \texttt{i am going with you .}$$

6.3.5 Command vs. Request. Finally, experiments were conducted to see whether commands could be turned into requests and vice-versa.

$$\begin{aligned}
\mathbf{s}^7_{\text{command}} &= \texttt{clean your room .} & \mathbf{E}\big(\mathbf{s}^7_{\text{command}}\big) &= \mathbf{z}^7_{\text{command}} \\
\mathbf{s}^7_{\text{request}} &= \texttt{can you clean your room ?} & \mathbf{E}\big(\mathbf{s}^7_{\text{request}}\big) &= \mathbf{z}^7_{\text{request}} \\
\mathbf{s}^8_{\text{command}} &= \texttt{come to my house .} & \mathbf{E}\big(\mathbf{s}^8_{\text{command}}\big) &= \mathbf{z}^8_{\text{command}} \\
\mathbf{s}^8_{\text{request}} &= \texttt{can you come to my house ?} & \mathbf{E}\big(\mathbf{s}^8_{\text{request}}\big) &= \mathbf{z}^8_{\text{request}}
\end{aligned}$$

$$\begin{aligned}
\mathbf{D}\big(\mathbf{z}^7_{\text{request}} - \mathbf{z}^7_{\text{command}} + \mathbf{z}^8_{\text{command}}\big) &= \texttt{can you come to my house ?} \\
\mathbf{D}\big(\mathbf{z}^7_{\text{command}} - \mathbf{z}^7_{\text{request}} + \mathbf{z}^8_{\text{request}}\big) &= \texttt{come to my house .} \\
\mathbf{D}\big(\mathbf{z}^8_{\text{request}} - \mathbf{z}^8_{\text{command}} + \mathbf{z}^7_{\text{command}}\big) &= \texttt{can you clean your room ?} \\
\mathbf{D}\big(\mathbf{z}^8_{\text{command}} - \mathbf{z}^8_{\text{request}} + \mathbf{z}^7_{\text{request}}\big) &= \texttt{clean your room .}
\end{aligned}$$

The results for this and the previous experiment suggest that the structure of the sentence may be a subset of the latent space as well. Isolating these elements could lead to a new way of building output sentences within the latent space.

Chapter 7. Future Work

The results and experiments of the trained model shows promise. One of the biggest challenges for this work was training a larger model with a bigger dataset, so that is the most important next step. Because the corpus only came from a book dataset, language of the data was limited to a single domain. It would be better to use a corpus gathered from multiple domains for a more general-purpose model. With a larger model, it could be possible to extract specific information from the encoded word vectors and also generate new sentences in the latent space by using different vectors as building blocks. This could lead to a new way of modeling a conversational AI.

It would also be beneficial to explore alternate model architectures or training schemes and compare their empirical results. The model could be designed as a denoising autoencoder by masking some of the input words, or as a variational autoencoder by sampling the latent vectors from a Gaussian. One thing that lacked in this work is empirical results using standard methods. Comparing this model to other existing language models in standard LM tasks could show unique properties of the model.

Another important work to be done is implementing the symbolic derivatives of the matrix-by-matrix functions to autograd libraries. Current autograd systems are not completely optimized for solving the gradients of matrix-by-matrix functions during the backward pass. By using the results from 4.9, the complexity of these derivative calculations and gradient multiplications can be reduced by a factor of 2. The rise in usage of transformers and attention mechanism leads to more and more matrix-by-matrix operations, hence a more optimal algorithm will greatly reduce time and cost of training large language models.

Bibliography

[1] Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *ArXiv*, abs/1607.06450, 2016.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[3] Yoshua Bengio, Li Yao, Guillaume Alain, and Pascal Vincent. Generalized denoising auto-encoders as generative models. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[4] Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*, pages 10–21, Berlin, Germany, August 2016. Association for Computational Linguistics.

[5] Denny Britz, Melody Guan, and Minh-Thang Luong. Efficient attention using a fixed-size memory representation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 392–400, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[6] Aylin Caliskan, Joanna J. Bryson, and Arvind Narayanan. Semantics derived automatically from language corpora contain human-like biases. *Science*, 356(6334):183–186, 2017.

[7] Daniel Cer, Yinfei Yang, Sheng yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder, 2018.

[8] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4960–4964, 2016.

[9] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 551–561, Austin, Texas, November 2016. Association for Computational Linguistics.

[10] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[11] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 670–680, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[12] George V. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[15] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[17] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

[18] Sepp Hochreiter and Jürgen Schmidhuber. LSTM can solve hard long time lag problems. In Michael Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems 9, NIPS, Denver, CO, USA, December 2-5, 1996*, pages 473–479. MIT Press, 1996.

[19] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Collected Papers*, page 60–64, 1985.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[21] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[22] Ryan Kiros, Yukun Zhu, Russ R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[23] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

[24] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[25] Hugo Daniel Macedo and José Nuno Oliveira. Typing linear algebra: A biproduct-oriented approach. *Science of Computer Programming*, 78(11):2160–2191, 2013. Special section on Mathematics of Program Construction (MPC 2010) and Special section on methodological development of interactive systems from Interaccion 2011.

[26] Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey. Adversarial autoencoders, 2015.

[27] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[28] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[29] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau,

M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[32] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195, 1999.

[33] Ofir Press, Amir Bar, Ben Bogin, Jonathan Berant, and Lior Wolf. Language generation with recurrent generative adversarial networks without pre-training, 2017.

[34] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[35] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[36] Scott Reed, Zeynep Akata, Xinchen Yan, Lajanugen Logeswaran, Bernt Schiele, and Honglak Lee. Generative adversarial text to image synthesis. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1060–1069, New York, New York, USA, 20–22 Jun 2016. PMLR.

[37] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[38] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors, 1986.

[39] Sasha Rush, Austin Huang, Suraj Subramanian, Jonathan Sum, Khalid Almubarak, and Stella Biderman. The annotated transformer, 2022.

[40] John Schulman. Introducing chatgpt, 2022.

[41] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. A hybrid convolutional variational autoencoder for text generation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 627–637, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[42] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and et al. Mastering the game of go with deep neural networks and tree search, Jan 2016.

[43] O.rka (stats.stackexchange.com/users/92493/o rka). How to evaluate the loss on a gaussian mixture model? Cross Validated. URL:stats.stackexchange.com/q/517652.

[44] Sandeep Subramanian, Sai Rajeswar, Francis Dutil, Chris Pal, and Aaron Courville. Adversarial generation of natural language. In *Proceedings of the 2nd Workshop on Representation Learning for NLP*, pages 241–251, Vancouver, Canada, August 2017. Association for Computational Linguistics.

[45] Aaron van den Oord, Oriol Vinyals, and koray kavukcuoglu. Neural discrete representation learning. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[47] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, page 1096–1103, New York, NY, USA, 2008. Association for Computing Machinery.

[48] Xinyi Wang, Hieu Pham, Pengcheng Yin, and Graham Neubig. A tree-based decoder for neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4772–4777, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[49] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2048–2057, Lille, France, 07–09 Jul 2015. PMLR.

[50] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning, 2020.