DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY

Differentiating Human and Machine Intelligence with Contextualized Embeddings

Citation

Tang, Brandon Ed. 2023. Differentiating Human and Machine Intelligence with Contextualized Embeddings. Bachelor's thesis, Harvard University Engineering and Applied Sciences.

Link

https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37378269

Terms of use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material (LAA), as set forth at

https://harvardwiki.atlassian.net/wiki/external/NGY5NDE4ZjgzNTc5NDQzMGIzZWZhMGFlOWI2M2EwYTg

Accessibility

https://accessibility.huit.harvard.edu/digital-accessibility-policy

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story

Differentiating Human and Machine Intelligence with Contextualized Embeddings

A THESIS PRESENTED
BY
Brandon E. Tang
TO
The Applied Mathematics Department of Harvard College

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Bachelor's of Arts
IN THE SUBJECT OF
Applied Mathematics

Harvard University
Cambridge, Massachusetts

May 2023

Thesis advisor: Professor Gabriel Kreiman Brandon E. Tang

Differentiating Human and Machine Intelligence with Contextualized Embeddings

Abstract

This thesis explores the application of deep learning models as contextual embedding functions for data enrichment in the Turing test, allowing an AI-based judge that automates the Turing test to more effectively differentiate between human and artificial intelligence inputs in tasks such as image captioning. Image captioning data was manually collected from both humans and AI models such as BLIP and GIT. Static embedding functions are first applied to the data before being propagated through pre-trained deep learning models of various architectures to obtain contextualized embeddings for classification. Specifically, transformers and convolutional neural networks are used to generate contextual embeddings for image and text data respectively. PCA dimensionality reduction is applied on the contextual embedding space to alleviate memory and resource constraints for training the AI-based judge for human-machine classification. AI judges based on support vector machines (SVM), Gaussian Naive Bayes and deep neural networks are trained and evaluated on the contextual embeddings, and the resulting performance metrics for classification accuracy are discussed. Further insights about possible innate differences between humans and AI in the domains of vision and language are analyzed.

Contents

0 Introduction 1

1 Background 4

1.1 Related Work 5

1.2 Turing Test 6

1.3 Classical Machine Learning 7

1.4 Deep Learning 17

1.5 Embeddings 23

2 Model 39

2.1 Task Domain 40

2.2 Contextual Embedding Combinations 40

2.3 Disjoint Encoder Transformer and Residual Convolutional Neural Network Contextual Embeddings 41

2.4 Combined Vision Transformer Contextual Embeddings 42

2.5 Classifier Models 44

3 Experiments and Results 45

3.1 Disjoint SentencePiece and Residual Convolutional Neural Network Embeddings 46

3.2 Disjoint Encoder Transformer and Residual Convolutional Neural Network Contextual Embeddings 47

3.3 Combined Vision Transformer Contextual Embeddings 49

4 Conclusion 51

References 55

Listing of figures

1.1 Gradient Descent 9

1.2 Linear Support Vector Machine Hyperplane Classifier 12

1.3 Non-linear Support Vector Machine with Kernel Transformation 15

1.4 Comparison of Biological and Artificial Neurons 19

1.5 Deep Neural Network Architecture. 19

1.6 SentencePiece Subword Tokenization. 24

1.7 Convolutions and Output Feature Map Generation 26

1.8 Convolutional Neural Network Architecture Visualization 28

1.9 Image Patch Tokenization 30

1.10 Self Attention for Text. 31

1.11 Transformer Encoder Block. 34

1.12 Encoder Transformer Architecture. 35

1.13 PCA Transformation with 2 Principle Components 36

2.1 Original ResNet-18 Architecture 41

2.2 Multimodal Input Concatenation. 43

Acknowledgments

I would like to give my deepest thanks and appreciation to Professor Gabriel Kreiman. From the day I listened to his first lecture in Neuroscience 240 to the entire journey of writing this thesis, I found his deep insights about how to better understand the intersection of human and AI behavior extraordinarily valuable, guiding me towards asking the right questions and finding the appropriate solutions. I would also like to thank Mengmi Zhang and Prachi Agrawal for their immensely helpful research advice and support throughout this endeavor. Finally, I express my deepest gratitude and appreciation to my mother, for her unwavering, unconditional support.

I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted.

Alan M. Turing

0 Introduction

The emulation of human cognition and rationality through artificial intelligence is an idea that has its roots in classical philosophy and mathematics, when the earliest thinkers began developing systems of mathematical logic and formulating human reasoning as symbolic systems. The development of these frameworks led to the famous *Church-Turing thesis* in the 1930's, which theorizes that any conceivable process of logical reason-

ing can be simulated by manipulating basic symbols such as "0" and "1". Shortly thereafter, what is generally considered the first form of artificial intelligence was introduced by McCulloch and Pitts in 1943, who created a computation model of the neuron. However, it was not until the Dartmouth Summer Research Project on Artificial Intelligence in 1956 that the term "artificial intelligence" was officially coined by John McCarthy. Since then, an explosion of innovations have developed in the field of artificial intelligence, from Yann LeCun's convolutional neural network that emulates the animal visual cortex, to Google's transformer architecture that processes sequences of data through a self-attention mechanism that mimics how humans perceive, associate and memorize information. State-of-the-art machine learning models such as DALL·E 2 and ChatGPT are able to generate hyper-realistic images and complex, logical sentences, blurring the line between human and machine capabilities. Amid these breakthroughs, a crucial question has become more important than ever before: how does one distinguish between human and machine generated data? How will we verify the authenticity of information and news with the rise of advanced generative capabilities from artificial intelligence? Knowing the differentiating factors between human and machine intelligence will not only help prevent the spread of misinformation from artificially generated data, but simultaneously make AI models more "human-like" in situations that require it, engineering models with the aforementioned differences in mind. Yet, differentiating between human and machine generated information and behavior remains a relatively unexplored research topic by the machine learning community. This thesis seeks to address the aforementioned issue by building automated judges that classify human and machine inputs, trained on contextual embeddings that represent those inputs. A **contextual embedding function** considers the relationships between various sections of its input and the implications they have on its overall mean-

ing when generating an embedding. In the case of text input, this would entail comparing each word's role with respect to other words in the given sentence. For image input, this would mean relating different sections of the given image with each other. A **static embedding function** for text data is a pre-trained, fixed representation of words in a vector space, meaning that it does not consider the specific input sentence as a whole when generating an embedding for each word in the sentence. As such, static embedding functions may fail to capture the nuanced context for words in a sentence. The central premise of this thesis is that contextual embeddings generated by deep learning architectures are distilled with richer language and image understanding than that of static embeddings, which only represent the meaning of the underlying data as fixed, isolated vectors. Hence, for the task of differentiating between human and machine inputs, contextual embeddings may increase the efficacy of classifier training compared to static embeddings.

As an AI language model, I do not pose any inherent danger to people or society. One of the main concerns with AI language models is the potential for misinformation and manipulation. Because language models like ChatGPT can generate human-like responses, they may be used to spread false information or propaganda. This could have serious implications for democracy, public discourse, and social cohesion.

ChatGPT

1 Background

An overview of the technical concepts used in this thesis is provided in this chapter. We begin by delineating the experimental setup of the Turing test, followed by in-depth explanations of various machine learning paradigms, embedding spaces and deep learning architectures.

1.1 Related Work

The Turing test was proposed by famous mathematician and computer scientist Alan Turing, functioning as an "imitation game" where a machine or artificial intelligence tries to seem human when conversing with a human judge, while the judge ultimately has to decide if they are interacting with a human or machine [13]. Since the 70 years after Turing's famous paper, there has been ample debate about whether the Turing test serves as a valid and relevant metric of intelligence. Creating an automated Turing test where the human judge is replaced by an AI judge lies at the essence of this thesis, where the decision making of the AI judge may reveal crucial insights on what cues and model designs make a machine seem less human. Furthermore, the original Turing test only pertains to text conversations (language) between a machine and human, but the imitation game concept can be applied to any domain of expression. For instance, machines can also imitate humans in tasks such as color estimation, image captioning, object detection, attention prediction, word associations, and so on. Zhang, Mengmi, et al. [17] collected a large amount of human and AI data for the aforementioned tasks and conducted the Turing test using both a human and AI judge for each task domain. The AI judge used in this paper was a support vector machine (SMV) model, meaning that no deep learning models were utilized in the training process and a separate AI judge was trained for each task domain. Leveraging the power of deep learning for differentiating human and AI behavior in diverse task domains is a clear step for further exploration, as there may be similar patterns among different task domains that can help discern between human and machine. The challenge lies in building a single multi-modal AI judge that can universally receive inputs from different task domains such as vision and language so that only one trained model is needed to conduct

the Turing test in all task domains. Until recently, transformer architectures were primarily trained for tasks in natural language processing, with Ashish Vaswani et al, 2017[14] and Devlin et al., 2019[4] introducing bidirectional encoder transformers (BERT) that led to significant progress in classification problems solely limited to text. Reed, Scott, et al.[12] propose "Gato", a generalist agent with a transformer architecture that can be trained in domains such as reinforcement learning and computer vision, far beyond the normal realm of natural language processing for the transformer architecture. The key technique that allows for the multi-modality of Gato is that all input forms are serialized into a flattened sequence of tokens such that the transformer architecture can readily receive them and be trained much like in the traditional context of natural language processing. Wang, Peng, et al.[16] introduces a similar transformer-based generalist agent called "One For All (OFA)" that is a sequence to sequence model with similar multi-model capabilities as Gato. OFA also makes use of an input serialization that flattens and tokenizes different data types into a "Unified Vocabulary". Kiela, Douwe, et al.[7] propose a multi-modal bitransformer model for classifying images and text, where the input sequence is a concatenation of text and image embedding sequences for tasks that involve both modalities of data. By applying the self-attention mechanism over both modalities of data simultaneously, their model is able to effectively process the mixed data in a fine-grained manner.

1.2 Turing Test

The Turing Test come from the "Imitation Game" concept from Alan Turing's "Computing Machinery and Intelligence" paper. The imitation game involves three participants. Participant **A** is a human, participant **B** is a machine, and participant **C** is a judge or "interrogator" as referred to by the paper. **A** and **B** converse with each other in a separate room

from **C**, who reviews their conversation by reading it in text. **C** knows the two other participants as labels **X** and **Y**, and at the end of their conversation, **C** has to say "X is A and Y is B" or "X is B and Y is A", giving their judgement on which participant is the human and which one is the machine. If the machine **B** can trick the judge **C** into thinking that it is the human **A**, then machine **B** passes the Turing test because it has convinced the judge **C** that is human. While the Turing test originally only refers to the domain of language, it can be applied to any domain of expression that humans and machines are both capable of. For instance, can one discern the difference between how a human would caption an image compared to a machine? In this task, **C** would receive an image and caption from both **A** and **B**, and would have to decide which one was captioned by a human. A common criticism for the Turing test is that **C** knows there is exactly one human and one machine in the imitation game, rather than being uncertain about whether both participants could be humans or machines, which may drastically alter how **C** makes their judgement. The original definition of the imitation game makes it such that **C**'s role is selecting which one of **X** and **Y** is **more human**, rather than only focusing on whether an certain pattern of behavior seems human in isolation.

1.3 Classical Machine Learning

The field of machine learning fundamentally embodies the idea of a machine autonomously learning from a set of data to improve performance on a given set of tasks, without any direct human intervention. In this thesis, we will be primarily concerned with binary classification in the context of supervised learning, a major paradigm of machine learning. **Supervised learning** pertains to problems where each data point (a set of features) in the dataset has an associated label. Supervised learning algorithms aim to learn a function that

accurately maps each set of input features x_i to its corresponding label y_i, where this learned function should be generalizable in that it can predict labels for input features that it has not observed during training. Taken together, (x_i, y_i) form a single data point $d_i \in D$, where D is the entire dataset. In binary classification, the labels for x_i are $y_i \in \{0, 1\}$ where the label is either 0 or 1. Supervised learning algorithms learn by optimizing with respect to a **loss function** J, which represents the difference between the algorithm's predictions and the true labels. A model that minimizes this loss function thus yields a model that consistently makes accurate predictions.

1.3.1 Linear Estimator

To understand the intuition behind machine learning, let us first describe how a simple linear estimator model works. We are given a labelled dataset D composed of X and Y each of size n, where each data point $x_i \in X$ has a corresponding label $y_i \in Y$. A estimator has a randomly initialized weight w and bias b; it receives input data X and maps these inputs to a prediction $\hat{Y}(w, b) = w \cdot X + b$. Notice that X is fixed and the only variables that will change are w and b. The predictions $\hat{Y}$ are compared to the true labels Y and both w and b are adjusted accordingly to minimize the difference between $\hat{Y}$ and Y, meaning that the estimator's predictions are as close to the real "solution" as possible. The difference between $\hat{Y}$ and Y is represented by an objective function; in this case, the objective function takes the form of a cost function. The mean square error cost function is

$$J(w) = \frac{1}{n} \cdot \sum_{i=1}^{n} (y_i - \hat{y}_i(w))^2 \tag{1.1}$$

Adjusting the weight w_{old} to obtain a new weight w_{new} to minimize the difference between

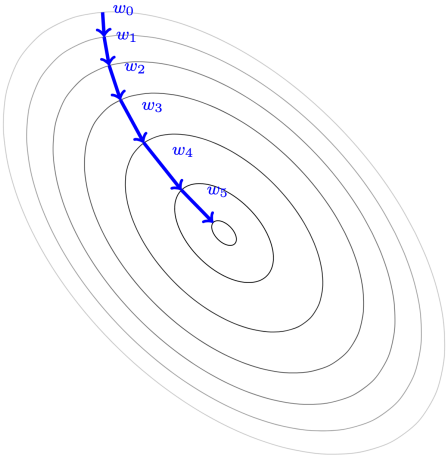


Figure 1.1: Illustration of how the gradient descent algorithm updates the parameter values w to converge on the minimum point of the cost function.

$\hat{Y}$ and Y is done by taking $\frac{dJ}{dw_{old}}$, the derivative of the cost function with respect to w_{old}, multiplying it by a learning rate α, and subtracting it from w_{old}:

$$w_{new} = w_{old} - \alpha \cdot \frac{dJ}{dw_{old}} \tag{1.2}$$

The intuition behind why this weight update rule minimizes the cost function is the following. If one assumes that the cost function J, which is a function of the weight w, is convex, then there must be a value for w that minimizes J. If we examine the mean squared error cost function, it takes the form of a parabola that opens upwards, meaning it is convex and has a minimum at $J(w_{min})$. Suppose we have a value w_{old} that is greater than w_{min}. Then $\frac{dJ}{dw_{old}}$ must be positive (this is the slope of the cost function at w_{old}, so in order to move w_{old} towards w_{min}, we subtract the slope which moves w_{old} towards the left. In the other case, suppose we have a value w_{old} that is smaller than w_{min}. Then $\frac{dJ}{dw_{old}}$ must be negative (this is the slope of the cost function at w_{old} so in order to move w_{old} towards w_{min}, we subtract the

slope which moves w_{old} towards the right. This is known as the gradient descent update method, since it uses the gradient of the cost function to adjust the weight w to "descend" along the cost function and reach its lowest point, effectively minimizing the cost. A learning rate is included in this update rule to control the size of the weight adjustment at each iteration of the update, and is often tuned to become smaller over iterations of updates. This allows the gradient descent process to actually converge on the minimum point of the cost function rather than endlessly jumping around because the update size is too large and overshoots the minimal point. This method is used the exact same way for the bias b.

1.3.2 Support Vector Machines (SVM)

The original Support Vector Machine (SVM) model[2] was developed at AT&T Bell Laboratories by Vladimir Vapnik and Alexey Chervonenkis in 1964, consisting of a linear hyperplane classifier. A more powerful SVM classifier was introduced in 1992 by Bernhard Boser, Isabelle Guyon and Vladimir Vapnik, leveraging the kernel trick to implement non-linear hyperplane classifiers.

Linear SVM

Suppose we have a dataset $D = \{(x_i, y_i)\}\, i = 1, \ldots, n$. Each data point $d_i \in D$ is composed of a set of input features and a corresponding target label (x_i, y_i), where x_i is a k-dimensional real vector and $y_i \in \{-1, 1\}$. The linear SVM algorithm finds a classifier in the form of a hyperplane that can best divide the two different classes of points. A hyperplane in k-dimensional space is defined as:

$$w^T x - b = 0 \tag{1.3}$$

where w is a k-dimensional vector normal to the hyperplane. The hyperplane should satisfy the **maximum-margin** property, which maximizes the distance between the hyperplane and nearest points from both groups. To do so, the linear SVM algorithm finds *two parallel hyperplanes* that can also separate the two classes of data points, with the distance between these two hyperplanes being maximized. The two parallel hyperplanes are:

$$w^T x - b = 1$$

and

$$w^T x - b = -1$$

Suppose point x_1 is on hyperplane $w^T x - b = 1$ and x_2 is on hyperplane $w^T x - b = -1$. The distance between these two parallel hyperplanes is equivalent to the distance between x_1 and x_2, given by $||x_2 - x_1|| = \frac{2}{||w||}$. We can show this by considering the line L (in k-dimensional space) that passes through x_1 in the direction of the normal vector w. We have $L = x_1 + wc$ where $c \in \mathbb{R}$. The intersection of L with the second hyperplane at x_2 can be found by plugging L into the second hyperplane equation:

$$w^T(x_1 + wc) - b = -1 \iff c = \frac{-1 + b - w^T x_1}{w^T w} = \frac{-2}{||w||^2}$$

meaning that distance between the two parallel hyperplanes is given by:

$$L = x_1 + wc = x_2 \iff ||x_2 - x_1|| = ||w|| \frac{|-2|}{||w||^2} = \frac{2}{||w||}$$

Minimizing $\|w\|$ maximizes $||x_2 - x_1||$, which results in the maximum-margin since $\|w\|$ is always positive. Fig 1.2 visualizes the maximum-margin property for a support vector

machine classifier on a set of linearly separable data.

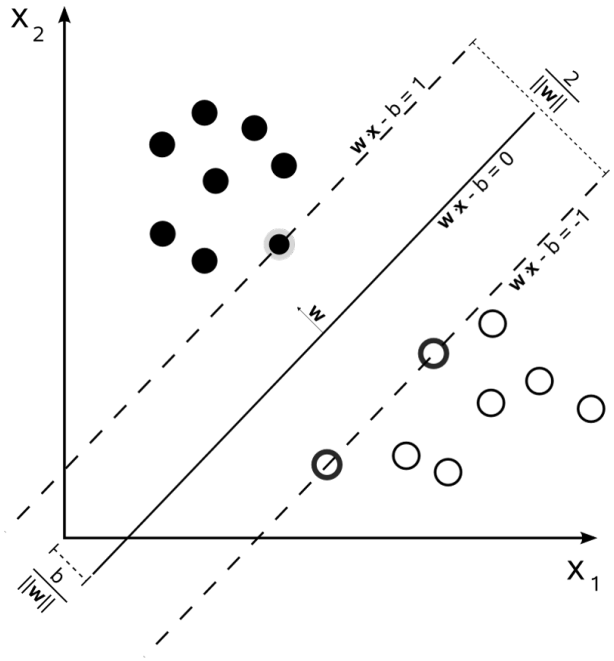


Figure 1.2: Two parallel hyperplanes satisfying the maximum-margin property for a linear support vector machine.

The constraints to be satisfied for every data point (x_i, y_i) are given by:

$$\mathbf{w}^T\mathbf{x}_i + b \geq +1, \quad y_i = +1 \tag{1.4}$$

$$\mathbf{w}^T\mathbf{x}_i + b \leq -1, \quad y_i = -1 \tag{1.5}$$

which is equivalent to

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 \geq 0, \quad \forall i \tag{1.6}$$

However, datasets are usually not linearly separable and hence will not satisfy the constraint above for all i. Thus, the **hard-margin** optimization approach (requiring linear separa-

bility) needs to be extended to not linearly separable datasets through the inclusion of the **hinge loss** function for a **soft-margin** optimization (not requiring linear separability):

$$\max\left(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b)\right)$$

The hinge loss function is zero if the original constraint is satisfied —that is, if $\mathbf{x}_i$ is on the correct side of the margin with respect to its label. For each data point on the wrong side of the margin (misclassified by the linear SVM hyperplane margin), the value of the hinge loss function is directly proportional to the distance of the datapoint from the margin. The reformulated optimization problem becomes

$$J(\mathbf{w}, b) = \lambda\|\mathbf{w}\|^2 + \left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b)\right)\right]$$

Since J is a convex function of $\mathbf{w}$ and b, sub-gradient descent can be used as solution method where instead of taking a step in the direction of the gradient of J, a step is taken in the direction of a vector selected from the sub-gradient of J. A **sub-gradient** is a generalization of the concept of a gradient to functions that may not be differentiable everywhere. Given a function $f(x)$ that is not differentiable at a point x, the sub-gradient at x is a vector that provides a lower bound on the slope of the function at that point. If $f(x) = |x|$ where $f(x)$ is not differentiable at $x = 0$, the set of sub-gradients of $f(x)$ at $x = 0$ is the interval $[-1, 1]$. The formal definition of the sub-gradient of a convex function $f : I \to \mathbb{R}$ at a point x_0 in the open interval I is a real number c where:

$$f(x) - f(x_0) \geq c(x - x_0), \forall x \in I$$

The set of sub-gradients at x_0 for a convex function f is a nonempty, closed interval $[a, b]$, where a and b are the one-sided limits:

$$a = \lim_{x \to x_0^-} \frac{f(x) - f(x_0)}{x - x_0}$$

$$b = \lim_{x \to x_0^+} \frac{f(x) - f(x_0)}{x - x_0}$$

The sub-gradient used on each iteration of sub-gradient descent is selected either randomly or by evaluating several candidate sub-gradients by observing how the model performs upon the application of each candidate, and selecting the one that yields the lowest loss.

Non-linear SVM: The Kernel Trick

When the dataset is not linearly separable, the **kernel trick** can be utilized to transform the dataset into a higher-dimensional space where it is more likely to be linearly separable. In the transformed feature space, the support vector machine can then find the hyperplane that maximizes the margin between the two classes of data points as usual. In particular, the kernel function used in this transformation is crucial in non-linear support vector machines, since it determines the similarity between pairs of data points in the feature space. For all $\mathbf{x}$ and $\mathbf{x}'$ in the input space $\mathcal{X}$, certain functions $k(\mathbf{x}, \mathbf{x}')$ can be expressed as an inner product in another space $\mathcal{V}$. The function $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is often referred to as a kernel or a kernel function. The word "kernel" is used in mathematics to denote a weighting function for a weighted sum or integral. Certain problems in machine learning have more structure than an arbitrary weighting function k. The computation is made much simpler if the kernel can be written in the form of a "feature map" $\phi\colon \mathcal{X} \to \mathcal{V}$ which satisfies

$k(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle_{\mathcal{V}}$. The primary restriction is that $\langle \cdot, \cdot \rangle_{\mathcal{V}}$ must be a proper inner product.

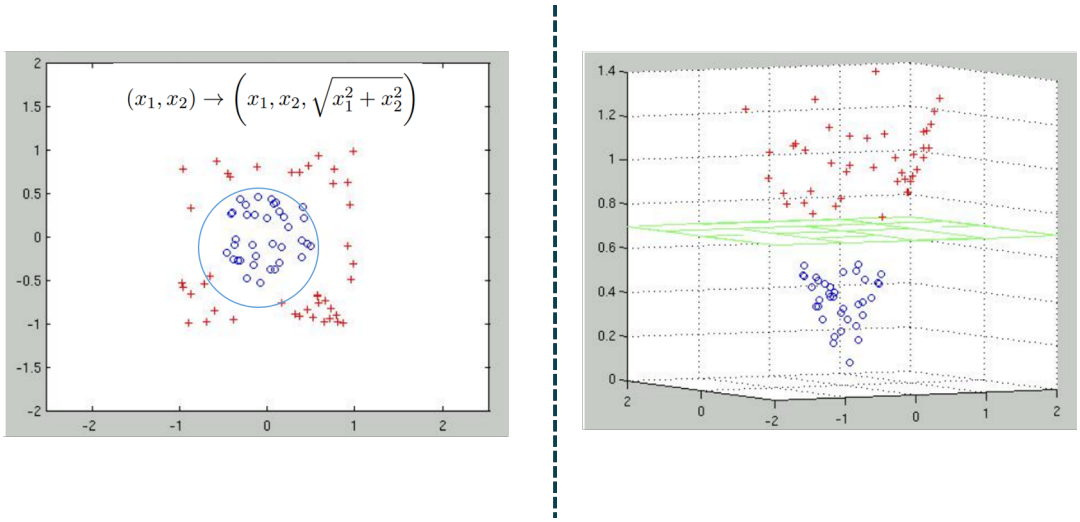


Figure 1.3: Transforming a 2-dimensional not linearly separable dataset into a 3-dimensional dimensional space where the transformed dataset is linearly separable by a maximum margin hyperplane found by a support vector machine.

Figure 1.3 shows how a not linearly separable dataset can become transformed into a higher dimensional space that is linearly separable by a hyperplane. Specifically, it has $\mathbf{x} = (a, b)$ and defines $\phi((a, b)) = (a, b, \sqrt{a^2 + b^2})$ which gives:

$$k(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle_{\mathcal{V}} = (a_1 \cdot a_2 + b_1 \cdot b_2 + \sqrt{a_1^2 + b_1^2} \cdot \sqrt{a_2^2 + b_2^2}) = \mathbf{x} \cdot \mathbf{x}' + \|\mathbf{x}\| \cdot \|\mathbf{x}'\|$$

utilizing the kernel trick to implement a non-linear support vector machine. Common kernels used for the kernel trick include polynomial kernels $k(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^d$ and Gaussian radial basis function kernels $k(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\gamma \left\|\mathbf{x}_i - \mathbf{x}_j\right\|^2\right)$ for $\gamma > 0$.

1.3.3 Naive Bayes Classifier

Suppose that the probability density function for a feature x in the dataset is given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \tag{1.7}$$

where the mean $\mu = 0$ and the variance $\sigma^2 = 1$ for the standard normal distribution $N(0,1)$. The conditional probability for x given a class label c_k for $k \in (0,1)$ is

$$f(x \mid c_k) = \frac{1}{\sigma_{c_k}\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu_{c_k}}{\sigma_{c_k}})^2} \tag{1.8}$$

where the μ_{c_k} is the mean value of the feature under the class c_k and σ_{c_k} is the standard deviation for the feature under class c_k. We want to find $P(c_k \mid x_1, x_2, x_3, ..., x_n)$, the probability that a given input feature was generated by a human or machine, for $k \in (0,1)$ where $k = 0$ is a machine and $k = 1$ is a human.

Let $\mathbf{x} = (x_1, x_2, x_3, ..., x_n)$ be the vector of features used for this classification problem. By Baye's rule, we have that:

$$P(c_k \mid \mathbf{x}) = \frac{P(c_k) \cdot P(\mathbf{x} \mid c_k)}{P(\mathbf{x})} \tag{1.9}$$

Since the denominator is not dependent on c_k and the values for all the features in $\mathbf{x}$ are known, we can treat the denominator as a constant and only focus on the numerator $P(c_k) \cdot P(\mathbf{x} \mid c_k)$. By "naively" assuming that each feature in $\mathbf{x}$ is independent from other features, we have that:

$$P(c_k) \cdot P(\mathbf{x} \mid c_k) = P(x_1, x_2, x_3, ..., x_n, c_k) \tag{1.10}$$

and can rewrite the joint probability $P(x_1, x_2, x_3, ..., x_n, c_k)$ as the following:

$$P(x_1, x_2, x_3, ..., x_n, c_k) = P(x_1 \mid x_2, x_3, ..., x_n, c_k) \cdot P(x_2, x_3, ..., x_n, c_k) \tag{1.11}$$

and we can recursively repeat this process for the joint probability $P(x_2, x_3, ..., x_n, c_k) = P(x_2 \mid x_3, ..., x_n, c_k) \cdot P(x_3, ..., x_n, c_k)$, and so on. Since we "naively" assume that each feature is independent from each other, we have that:

$$P(c_k \mid \mathbf{x}) \propto P(x_1, x_2, x_3, ..., x_n, c_k) = P(c_k) \cdot \prod_{i=1}^{n} P(x_i \mid c_k) \tag{1.12}$$

where each $P(x_i \mid c_k)$ can be found by the Gaussian probability density function conditioned on class c_k given by Equation (2). The naive Bayes classifier's prediction $\hat{y}$ of the class of input features $\mathbf{x}$ is given by the following assignment rule:

$$\hat{y} = \text{argmax}_{k \in (0,1)} \; P(c_k) \cdot \prod_{i=1}^{n} P(x_i \mid c_k) \tag{1.13}$$

which assigns to x_i the class c_k that has the highest probability.

1.4 Deep Learning

Deep learning is a sub-field of machine learning, pertaining to the usage of artificial neural networks —which take inspiration from the biological neural networks that compose animal and human brains —for learning representations that discover patterns in raw data needed for prediction tasks.

1.4.1 Deep Neural Networks

A linear model can only model linear relationships, which limits its predictive power and representational ability. There are three changes we can make to the current linear esti-

mator such that it becomes a "deep learning" neural network model. The first is adding a non-linear activation function φ in our computation of $\hat{y}$, like the following:

$$\hat{y}(w, b) = \varphi(w \cdot x + b) \tag{1.14}$$

where it could be the Rectified Linear Unit (ReLU) function $\varphi(v) = \max(0, v)$. This allows our estimator to model non-linear relationships, greatly expanding its expressive capability. The second is that we can have multiple weights rather than just one, such that $\vec{w} = w_1, w_2, ..., w_n$ given an input datapoint $\vec{x} = x_1, x_2, ..., x_n$ of size n. Hence, we would take a dot product and have:

$$\hat{y}(w, b) = \varphi(\vec{w}^\top \vec{x} + b) \tag{1.15}$$

The third adjustment we can make is treating Eq. 1.4 as a *node*, where the input "flows" through this node and gets transformed into an output. This conception of a node or *neuron* takes inspiration from the action potential of a biological neuron, depicted in Figure 1.4. We can then have many of these nodes together in a *layer*, with input flowing through each node of the layer independently. If we have k nodes in a single layer and the input flows through these nodes simultaneously, then the output from that layer will be a new, transformed vector input z of size k as input for the nodes of the next layer. We can have multiple layers (different layers can have different number of nodes) that come after one another, where the output of one layer is the input for the next layer. By feeding this transformed vector input z into the nodes of the next layer, we can repeatedly stack layers together such that the estimator now becomes "deep" because it has multiple so called "hidden" layers between the original input and final output prediction. Hence, each node of

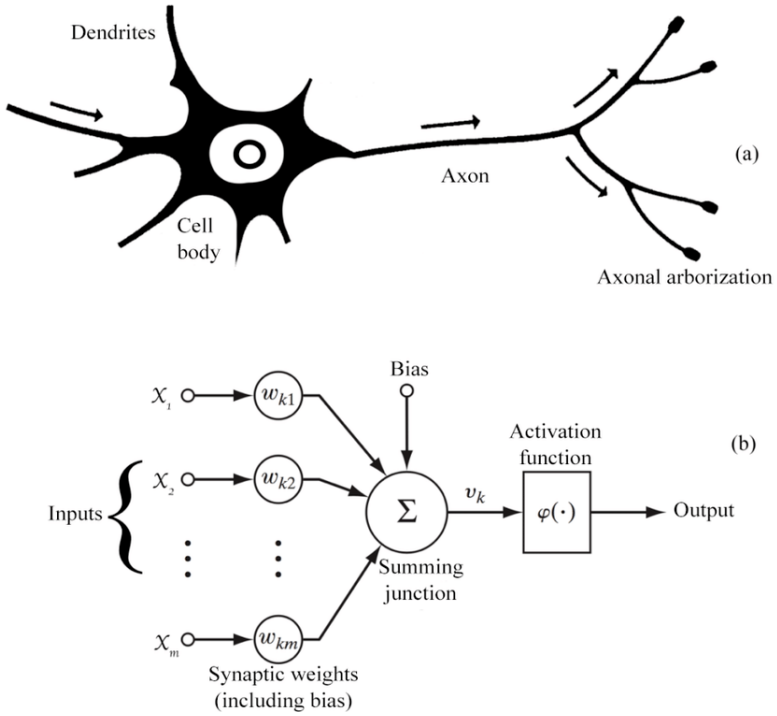


Figure 1.4: Comparison of a biological neuron with an artificial neuron. Image credit to (Arbib, 2003a; Haykin, 2009b).

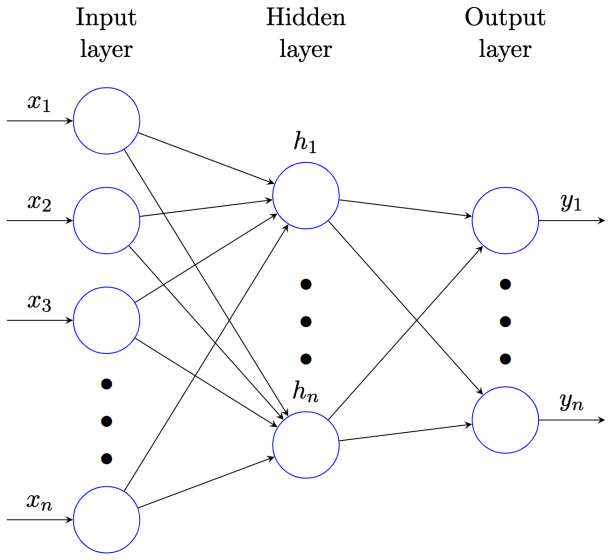


Figure 1.5: Diagram of a standard, fully connected deep neural network architecture from input to output.

each hidden layer h_i outputs $z_{i+1} = \varphi_i(w_i \cdot z_i + b_i)$, which is collectively denoted as $Z_{i+1} = A_i(W_i \cdot Z_i + B_i)$ and used as input for the next hidden layer h_{i+1} and so on, until a single output prediction $\hat{y}$ is reached. This process of mapping inputs to outputs in a deep neural network is known as **forward propagation**. Formally, forward propagation through a deep neural network with n hidden layers is described the following set of equations, starting with a base case that begins with the input data X:

$$Z_2 = A_1(W_1 \cdot X + B_1) \tag{1.16}$$

The recurrence relation that describes the input Z_i to the i^{th} layer is (A represents the activation function previously denoted by φ):

$$Z_{i+1} = A_i(W_i \cdot Z_i + B_i) \tag{1.17}$$

The final output prediction from the neural network upon the completion of forward propagation is therefore:

$$\hat{y} = A_n(Z_n) \tag{1.18}$$

with the loss function:

$$J(w) = \frac{1}{D} \cdot \sum_{j=1}^{D} (y_j - \hat{y}_j))^2 \tag{1.19}$$

The weights of a deep learning neural network are still updated by the gradient descent update rule, but since there are multiple hidden layers and the impact of the weights in each layer on the final output prediction is dependent on the weights of the layers that come after it, we need to apply the chain rule in order to properly compute the appropriate gradi-

ents. This process of updating weights in all previous layers is known as **back-propagation**. For a deep neural network with n hidden layers, the gradient of the weight matrix in the k^{th} hidden layer W_k is computed as the following:

$$\frac{dJ}{dW_k} = \frac{dJ}{d\hat{y}} \cdot \frac{d\hat{y}}{dZ_n} \cdot \frac{dZ_n}{dA_{n-1}} \cdot \frac{dA_{n-1}}{dZ_{n-1}} \cdot \frac{dZ_{n-1}}{dA_{n-2}} \cdot \ldots \cdot \frac{dA_{k+1}}{dZ_{k+1}} \cdot \frac{dZ_{k+1}}{dA_k} \cdot \frac{dA_k}{dZ_k} \cdot \frac{dZ_k}{dW_k} \quad (1.20)$$

A modification on the traditional deep neural network is the **residual network**, where outputs from a given layer can "skip" over several proceeding layers and contribute to the input several layers ahead. For instance, given $W^{k-1,k}$ for weights that connect layer $k-1$ to k, and $W^{k-p,k}$ for weights that connect layer $k-p$ to k, forward propagation for a residual network is defined as:

$$a^k := \mathbf{g}(W^{k-1,k} \cdot a^{k-1} + b^k + W^{k-p,k} \cdot a^{k-p})$$

where $W^{k-1,k} \cdot a^{k-1} + b^k$ is the output from the previous adjacent layer $k-1$ and $W^{k-p,k} \cdot a^{k-p}$ is the skipped output from p layers back. Back-propagation between layer k and layer $k-p$ for residual networks is computed as the following:

$$W_{new}^{k-p,k} = W_{old}^{k-p,k} - \alpha \frac{\partial J}{\partial W_{old}^{k-p,k}}$$

simply adapting the original back-propagation update rule to the weight connection between layer $k-p$ and k.

1.4.2 Regularization

Regularization is a method to prevent a neural network from over-fitting by penalizing large weight values. When the weights of a neural network are unstable and have large mag-

nitudes, the model may be overconfident in its solution and may not be generalizable. This penalization is incorporated into the objective cost function as an additional term. Below is the equation for L1 regularization with mean squared error loss, which includes the first order term for the weights in the cost function:

$$J_1(w) = MSE(Y - \widehat{Y}) + \lambda \sum_{1}^{n} |w_i| \tag{1.21}$$

Below is the equation for L2 regularization with mean squared error loss, which includes the second order term for the weights in the cost function:

$$J_2(w) = MSE(Y - \widehat{Y}) + \lambda \sum_{1}^{n} w_i^2 \tag{1.22}$$

The difference between these two forms of regularization is that when one takes the derivative of the cost function with respect to the weights, $\lambda \sum_1^n |w_i|$ becomes a constant and subtracting $\frac{J_1(w)}{dw}$ from the neural network weights during the gradient descent update rule means subtracting a constant. On the contrary, $\lambda \sum_1^n w_i^2$ becomes proportionate to w and subtracting $\frac{J_2(w)}{dw}$ from the neural network weights during the gradient descent update rule means subtracting a value proportional to the weight itself. Hence, L2 regularization penalizes large weights more heavily than L1 regularization, but when the weights are smaller, L2 does not zero the weights and keeps them small, while L1 may penalize weights to zero, irrespective of how small the weights are already. Hence, L1 regularization shrinks many weights to zero, while L2 regularization shrinks weights relatively evenly and does not zero weights.

1.4.3 Hyperparameters

While weights and biases are parameters that are updated by the neural network model during training, hyperparameters are values that govern the architecture and training of the model without actually being changed by the model itself. This includes the learning rate, input batch size, number of training iterations, number of hidden layers and nodes, the regularization constant and more.

1.4.4 Evaluation

Datasets are split into two components, the training set and the test set, where the neural network trains on the training set and is evaluated on the test set, where it has never observed the test data before. During evaluation, the neural network only runs inference (forward propagation) on the test set and makes predictions while the weights are held fixed.

1.5 Embeddings

1.5.1 Static Text Embeddings

Transforming text (sequence of words) into a sequence of static vector embeddings is a two stage process that involves tokenization and embedding. Tokenization refers to the process of breaking down unstructured data into structured, discrete components. For text data, sentences are tokenized into discrete word or subword tokens, as shown in Figure 1.6. These text tokens are then typically embedded with an embedding matrix that maps each token to a vector in an embedding space, where vectors that are close together in the embedding space tend to represent words that are similar to each other. The embedding matrix is obtained by training a separate supervised learning neural network on the words

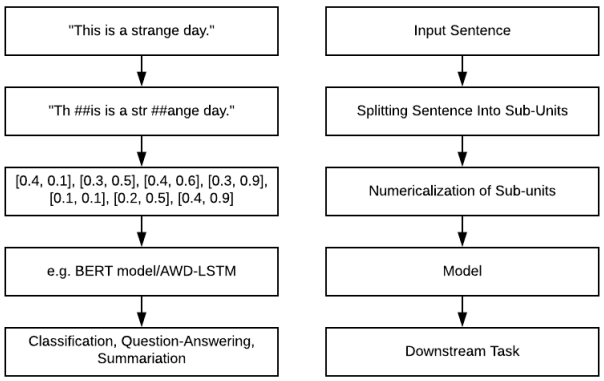
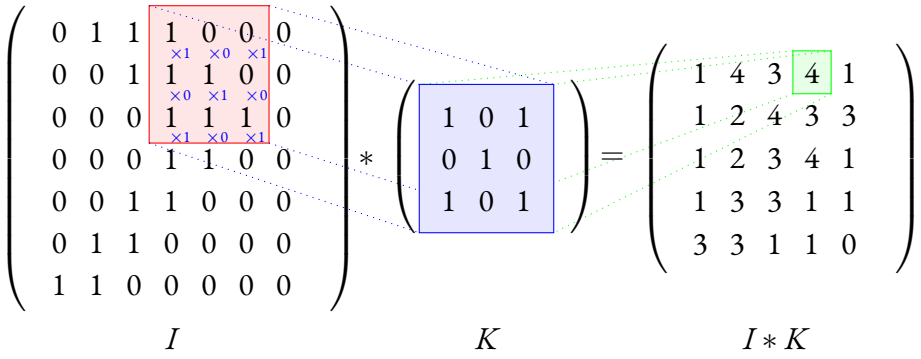


Figure 1.6: Diagram of how the SentencePiece subword tokenization function processes a text sentence. Image credit to Jacky Wong.

and obtaining a mapping between words and the vector embedding space.

1.5.2 Contextual Embeddings: Convolutional Neural Networks

A convolutional neural network (CNN) is a type of artificial neural network that utilizes the convolution operation instead of matrix multiplication, and are designed to process pixel data. A **convolution** operation is the dot product between a **kernel** matrix and a section of an input matrix, where the kernel matrices in a given layer contain the weights of that layer. For 2-dimensional input and kernel matrices, the convolution operation would be the following:



I K $I * K$

Convolutional neural networks work with 3-dimensional input pixel data and kernel matrices (width, height, depth), where the kernel matrix always has the same depth size as the input matrix that the convolution is being performed on. Abstractly, the kernel matrix "slides" through the input matrix and performs a convolution at each stride, generating a feature map which becomes part of the input matrix to the next layer of the convolutional neural network. This process of convolving kernel and input matrices for forward propagation is described in more detail below.

Given input pixel data $I^{n \times n \times d}$ and a set of kernels $F_i^{k \times k \times d}$, $\forall i$, the convolution operation involves sliding each kernel over the image, computing the dot product (typically the Frobenius inner product) between the kernel and the portion of the $I^{n \times n \times d}$ it overlaps with, and storing the result in a new output feature map to be passed as part of the input to the next hidden layer. The three primary hyperparameters that govern the dimensions of the output feature map are **number of kernels**, **stride** and **padding**. The number of kernels is equal to the number of output feature maps, where convolving each kernel with the input matrix generates a single output feature map; all the output feature maps are directly stacked upon each other to form the input for the next layer. The stride parameter controls the spacing between consecutive positions where the kernel is applied to the input matrix.

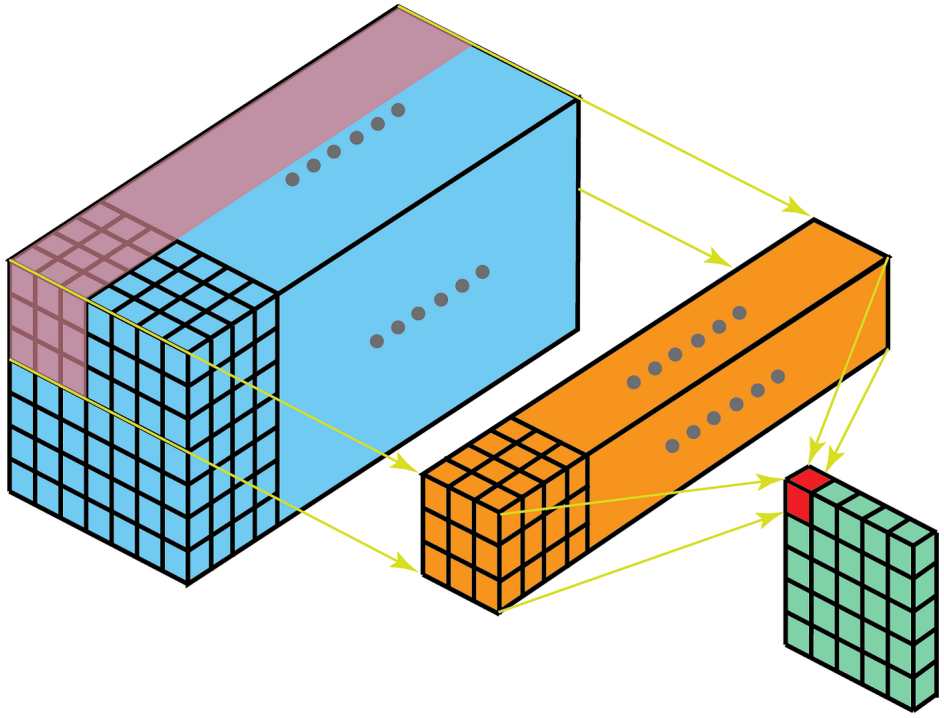


Figure 1.7: Visualization of how the convolution operation with input and kernel matrix generates an output feature map. The kernel matrix (orange) moves along in the input matrix (blue) with stride $s = 1$, performing a convolution at each stride and generating the output feature map (green). Image credit to Prakhar Ganesh.

Specifically, a stride of s means that the kernel is applied to every s^{th} position along the width and height dimensions of the input. The padding parameter governs how many extra pixels or zeros are added around the edges along the width and height dimensions of an input matrix before performing a convolution operation. The primary purpose of padding is to maintain the spatial dimensions of the output feature maps, which will otherwise be reduced due to the size of the kernels used in the convolution operation. When a kernel is applied to an input matrix, the size of the output feature map is smaller than the input (with respect to the width and height dimensions), as the edges along the height and width dimensions of the input matrix are only partially covered by the kernel. This can lead to a loss of information that carries on through the convolutional layers in the rest of the network. To avoid this issue, padding can be used to expand the size of the input image, such that the output feature map has the same height and width dimensions as the input. The l^{th} output feature map generated from convolving kernel $F_l^{k\times k\times d}$ and $I^{n\times n\times d}$ is computed as the following:

$$O[m,n,l] = \sum_{w}\sum_{u}\sum_{v} I^{n\times n\times d}[m+u, n+v, w] \cdot F_1^{k\times k\times d}[u,v,w]$$

Convolutional neural networks, as a type of artificial neural network, have an input layer, multiple hidden layers and an output layer. When used as a function for generating contextual embedding on images, a regular fully connected deep neural network is appended to the traditional convolutional neural network output layer, as shown in Figure 1.8. The 3-dimensional output feature map from the last convolutional hidden layer is a flattened into a 1-dimensional sequence, which is then fed into the fully connected deep neural network such that the final contextual embedding is a h-dimensional vector where h

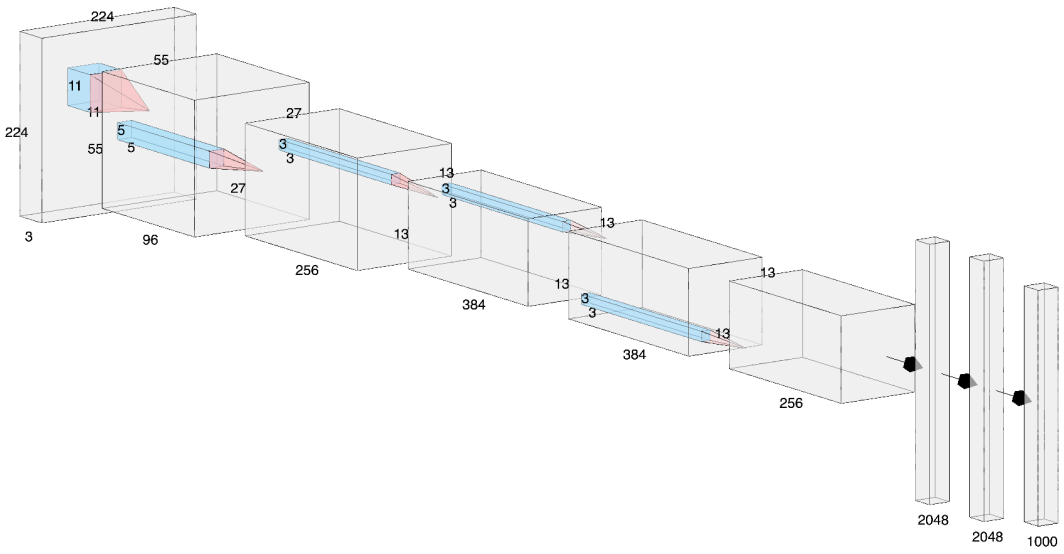


Figure 1.8: Visualization of the convolutional neural network architecture merged with a fully connected deep neural network for generating contextual embeddings.

is the number of nodes in the last layer. Convolutional neural networks produce contextual embeddings rather than static embeddings because they account for the relationship between certain parts of an image or feature map with other parts through the sliding kernel mechanism. The specific contextual embedding that will is used is **ResNet**[6], a modification on the convolutional neural network where outputs from a layer can "skip" over several proceeding layers and contribute to the input several layers ahead.

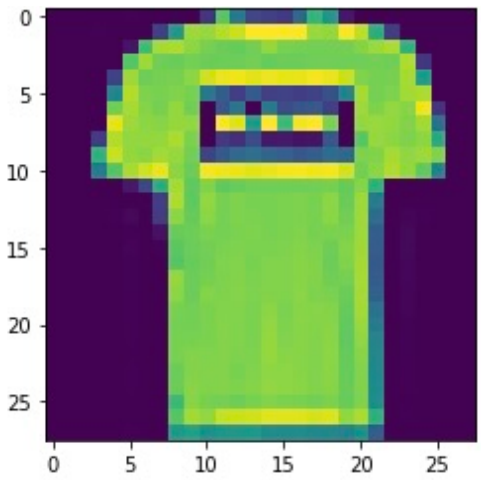
1.5.3 Contextual Embeddings: Transformers

A transformer is a deep learning model that utilizes the mechanism of self-attention to assign importance to different sections of the flat input sequence while relating those sections together. Transformers were originally used for natural language processing tasks since sentences are naturally encoded as a flat sequence of embedded words; furthermore, words are

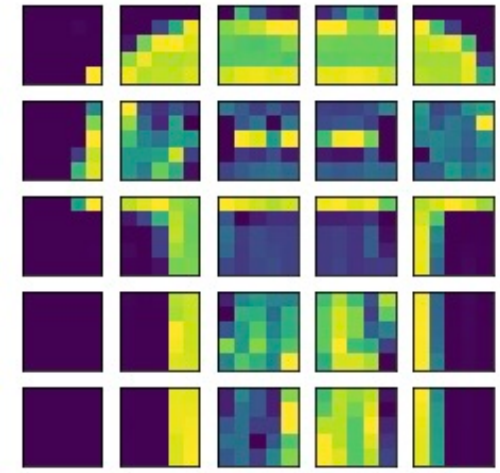
related to other words in the sentence despite being far apart from each other. Since transformers can process any input that can serialized into a flat sequence of vector embeddings, they have recently been applied to computer vision tasks where images are serialized into a sequence of smaller embedded patches. Transformers process sequential input data all at once in parallel, as opposed to other models such as recurrent neural networks which process sequential inputs sequentially.

Input Sequence Transformation

The input type to a transformer model is a sequence of vector embeddings of the same dimension D. Both text and images can be transformed into the input form mentioned above. This transformation is composed of a two stage process that involves tokenization and vector embedding. Text is transformed into a sequence of tokens, then to a sequence of vector embeddings through a static embedding function such as SentencePiece. Image tokenization involves dividing the image into a grid of smaller square, image patches, forming a sequence of patch tokens that compose the original image as illustrated in Figure 1.9. Once the original data has transformed into a sequence of tokens, an embedding function is applied on the tokens to yield a sequence of vector embeddings. For images, a convolutional neural network is usually applied to each patch token —typically a single ResNet block[6] — generating a vector embedding as output. These resultant sequence of vector embeddings can then be fed into the transformer for training, where operations such as dot products and matrix multiplications can be performed.



(a) Original unprocessed image.



(b) Image divided into grid of smaller patches.

(c) Sequence of patch tokens.

Figure 1.9: Visual illustration of image patch tokenization. Image credit to Mehreen Saeed from AI Exchange.

Positional Encoding

Positional encoding is used to indicate and preserve the order of the words in the input sequence. The position of a token in a sequence is transformed into a learnable positional embedding which is added to the original tokenized embedding.

Self Attention

The self-attention mechanism allows the transformer model to access all previous states and sections of the input sequence and learn to weigh each section depending on their relevance, describing contextual relationships between sections of the input sequence that may be far apart and not adjacent to each other.

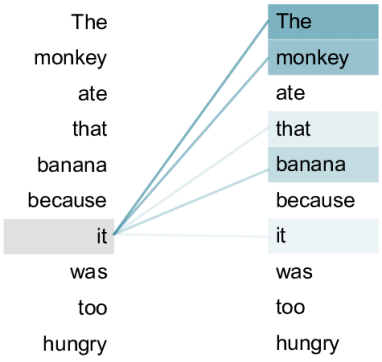


Figure 1.10: Visualization of how self attention is computed for a given word with respect to all other words in the input sequence. Image credit to Huiqiang Xie, et al..

Fundamentally, the self attention layer of a transformer model has three main weight components, the Query, Key and Value weight matrices. The input vector embeddings are multiplied with these three types of weight matrices, producing a ”query” vector, ”key” vector and ”value” vector for each input vector embedding. We will assume that the input sequence of embeddings represents a sentence with tokenized words in the following ex-

planation for clarity. Next, a score for each word with respect to every other word in the input sequence needs to be computed, where this score represents where in the input sequence that the transformer model should focus on. Suppose we are computing the score of the first word embedding with every other word embedding in the sequence, and the first word embedding has corresponding query, key and value vectors q_1, k_1 and v_1 respectively. The score between the first and second word embedding in the sequence is the dot product $q_1 \cdot k_2$. The score between the third and fifth word embedding in the sequence is the dot product $q_3 \cdot k_5$. Once all the scores are computed, they are divided by the square root of the dimension of the key vectors for stability in training, and passed through a softmax function which normalizes the scores and ensures that they add up to 1. The softmax function is given by the following:

$$\sigma(s_i) = \frac{e^{s_i}}{\sum_{j=1}^{K} e^{s_j}} \quad for\, i = 1, 2, \ldots, K \tag{1.23}$$

These softmax scores for a word embedding at given position in the sequence show the extent that each word embedding in the sequence is expressed or related to that position. The given word embedding and itself will always have a non-negligible score but often there are word embeddings far away in the input sequence that also have a high score, indicating that the "far away" word embedding is relevant to the word embedding at the current position and is worth giving "attention" to. The self attention mechanism is illustrated in Figure 1.10. Finally, the self attention layer multiplies the value vector of each word embedding with their softmax scores, and sums the resulting vectors together into a single output that is the weighted sum of value vectors with respect to their softmax scores. Multiplying the value vectors by their softmax scores has the effect of keeping the presence of relevant

words and diminishing the impact of irrelevant word embeddings in the final self-attention vector for the word embedding at the given position. In other words, the softmax score accounts for the attention given to other word embeddings in the sequence with respect to the current word embedding, while the value vector represents the intrinsic importance of those word embeddings. This self-attention computation is carried out for each word in the sequence, and all the self attention outputs are then fed into a fully connected neural network. Together, the self attention layer and the fully connected neural network form an *Encoder* block; transformers models stack many encoder blocks together where the output of the previous block is the input to the next block.

Multi-Headed Attention

The self-attention layer can have more than a single ”head” of attention. That is, it can have multiple attention heads, each with its own separate set of Query, Key and Value weight matrices. This way, each attention head serves as a separate representation space for the input data and can focus on more positions than a single attention head. Furthermore, all attention heads can be run in parallel as the computations on the input embeddings are independent.

Encoder Blocks

An encoder block in the transformer architecture is composed of a self-attention layer with multi-headed attention whose outputs are merged and inputted into a fully connected neural network. Within the encoder, there are residual connections between the input and self attention layer as well as the self attention layer and the fully connected network. Residual connections incorporate the input of a layer into the construction of the output of that

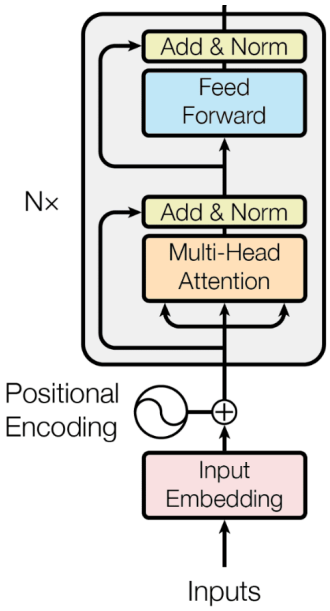


Figure 1.11: Diagram of encoder block in the transformer architecture. Image credit to Ashish Vaswani, et al., 2017.

same layer, separate from the main propagation path that maps input to output. This essentially mitigates the vanishing gradient problem.

Transformer Architecture

The final encoder transformer architecture that will be used as a contextual embedding function simply stacks multiple encoder blocks together, with each encoder functioning as a hidden layer in a deep neural network.

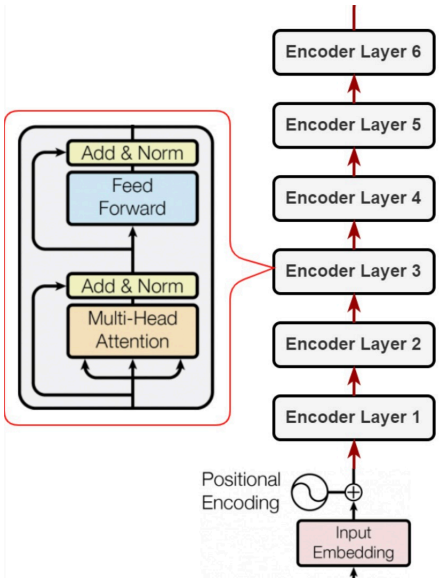
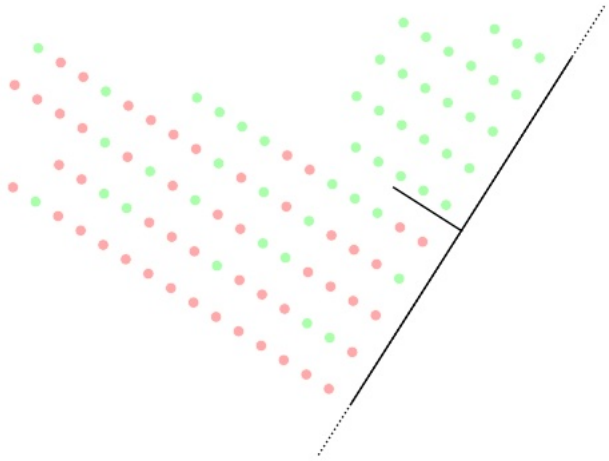


Figure 1.12: Diagram of the transformer architecture being composed of multiple, stacked encoder blocks. Image credit to Deep Learning Bible.

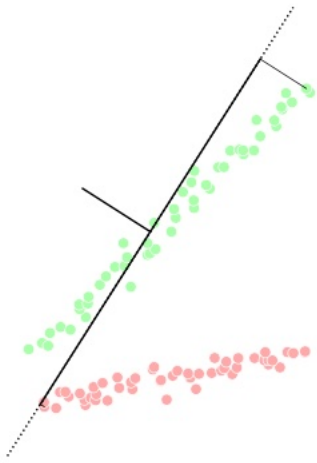
1.5.4 Principle Component Analysis (PCA)

Principal Component Analysis (PCA) is a statistical method used to reduce the dimensionality of a dataset, typically when the dimensionality is too large to train models efficiently. It does this by identifying the most important "principal" features of the dataset and creating new variables, called principal components, that represent a combination of the original features. The basic idea behind PCA is to find the directions in the data that have the most variance. These directions, or principal components, are a linear combination of the original features. The first principal component is the direction that has the highest variance, the second principal component is the direction with the second highest variance, and so on. Each principal component is orthogonal to the others, meaning that they are uncorrelated.

There are several steps that need to be taken in order to perform PCA. Firstly, the dataset



(a) Original Dataset



(b) PCA Projected Dataset

Figure 1.13: PCA transformation with 2 principle components on a 2D dataset. Dimensionality reduction occurs when the number of principle components used is less than the dimensionality of the original dataset.

needs to be standardized such that each feature has a mean $\mu = 0$ and a variance $\sigma^2 = 1$. Then, the **covariance matrix** of the standardized data needs to be computed. A covariance matrix is a square matrix that contains the variances and covariances of a set of features, describing the relationships between multiple features in a dataset. Specifically, the diagonal elements of a covariance matrix represent the variances of each individual feature, and the off-diagonal elements represent the covariances between each pair of features. **Covariance** is a measure of how two features change together, while **variance** is a measure of how much a single feature varies. The covariance matrix $\mathbf{K_{XX}}$ is the matrix whose (i, j) entry is the covariance between two features X_i and X_j:

$$cov(X, Y) = E[(X - E[X])(Y - E[Y])]$$

where X_i and X_j are random variables with mean $\mu = 0$ and variance $\sigma^2 = 1$ and $E[X_i]$ and $E[X_j]$ are the expected values of X_i and X_j, respectively. Intuitively, when the covariance between two variables is positive, it means that they tend to increase or decrease together. If the covariance is negative, it means that they tend to move in opposite directions. If the covariance is zero, it means that the variables are uncorrelated. The next step is to compute the *eigenvectors* and *eigenvalues* of the covariance matrix. An eigenvector is a non-zero vector that, when multiplied by a given square matrix A, results in a scalar multiple of itself. More formally, let A be an $n \times n$ square matrix and v be a non-zero vector in n-dimensional space. If there exists a scalar λ such that the following equation holds,

$$A \cdot v = \lambda * v$$

then v is called an eigenvector of A corresponding to the eigenvalue λ. Eigenvalues, on the

other hand, are the scalar values λ that satisfy the above equation. In other words, an eigenvalue of a matrix A is a scalar λ that, when multiplied by an eigenvector v, results in the product Av being equal to λv. Intuitively, eigenvectors represent directions along which a linear transformation (represented by the matrix A) stretches or compresses the vector space, while eigenvalues indicate the magnitude of that stretching or compression. To compute the eigenvectors and eigenvalues of a square matrix A, the characteristic polynomial of A needs to be found, which is the polynomial obtained by taking the determinant of the matrix A minus the scalar λ multiplied by the identity matrix $p(\lambda) = det(A - \lambda I)$. The eigenvalues of A can be found by solving for the characteristic equation $p(\lambda) = 0$. The eigenvectors can be found by solving the system of equations $(A - \lambda I)x = 0$. The non-trivial solutions to this equation (i.e., the solutions that are not the zero vector) will give us the eigenvectors. Once all the eigenvectors are computed, the next step is to sort them in descending order by their corresponding eigenvalues and select the first p eigenvectors, where p is the desired number of principle components. Finally, each input feature X_i is projected onto the p eigenvectors to obtain the resultant principal components, comprising a p-dimensional feature which has reduced dimensionality compared to X_i.

As an AI language model, I do not pose any inherent danger to people or society. One of the main concerns with AI language models is the potential for misinformation and manipulation. Because language models like ChatGPT can generate human-like responses, they may be used to spread false information or propaganda. This could have serious implications for democracy, public discourse, and social cohesion.

ChatGPT

2

Model

The specific setup of the pre-trained machine learning models and image captioning are detailed in this chapter. In particular, data collection details for the task of image captioning, as well as the hyperparameters of both the contextual embeddings and machine learning classification models are addressed.

2.1 Task Domain

Human and AI data for the image captioning task was collected, where the subject produces a single sentence text description for a given image. The images were obtained from the **Microsoft COCO: Common Objects in Context** dataset [10] and the **nocaps: novel object captioning at scale** dataset [1]. Each image in the dataset has 5 associated machine-generated captions from GIT [15], OFA [16], BLIP [9], ClipCap [11], and Microsoft's Azure Cognitive Services respectively, as well as 2 self-collected, human-generated captions. In total, the dataset contains 1000 images, each with 7 associated captions. A single image-caption pair is considered a single datapoint, so the entire dataset contains 7000 datapoints.

2.2 Contextual Embedding Combinations

Transformers and convolutional neural networks are both considered contextual embedding functions because they both account for the contextual relationships between different sections of the input. On the other hand, SentencePiece only considers sections of the input in a standalone manner, mapping subword tokens to embeddings without any consideration of their relationships with other subwords. The two main types of contextual embedding functions will be used to generate the final dataset for classification. The first is a disjoint encoder transformer and convolutional neural network function, where for each image-caption pair, the static sequence of caption text embeddings are given as input to an encoder transformer model and the image is fed as input to the residual convolutional neural network. The resultant embeddings from the encoder transformer and residual convolutional neural network are then concatenated together (with a separator token in between the concatenation) to form a single, contextual embedding. The second is a combined vi-

sion transformer function, where for each image-caption pair, the image is transformed into a sequence of patch embeddings and concatenated with the static sequence of caption text embeddings, and the concatenated sequence is fed as input to the vision transformer. A vision transformer is identical to the encoder transformer in terms of architecture; the only difference is that it is trained on image patch rather than text data.

2.3 Disjoint Encoder Transformer and Residual Convolutional Neural Network Contextual Embeddings

The architecture for the Bidirectional Encoder Representations from Transformers (BERT) transformer applied on the text captions is the **bert-base-uncased**[4] model from HuggingFace, shown in Figure. The bert-base-uncased model contains 110 million trainable parameters, with 12 encoder hidden layer blocks of hidden size 768, and 12 attention heads in each layer. The architecture for the residual convolutional neural network applied on the images is **ResNet-18**, shown in Figure 2.1, where the output from earlier layers is added to the input of layers further along the network.

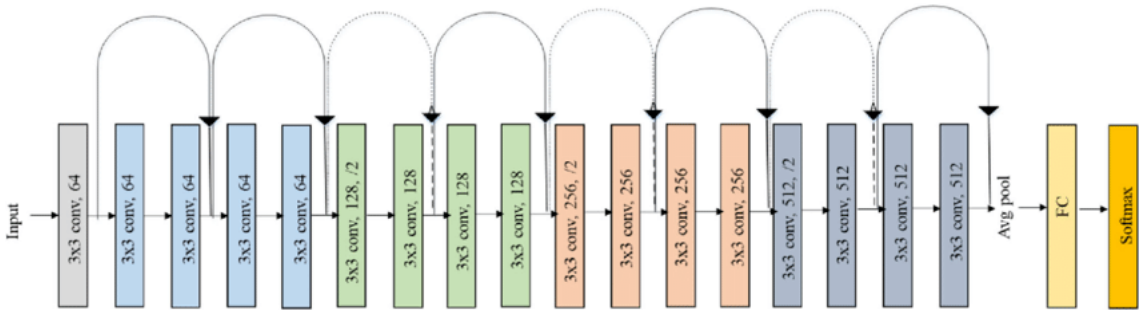


Figure 2.1: Original ResNet-18 Architecture.

2.3.1 Input Transformations

Text embeddings through SentencePiece (Kudo & Richardson, 2018) with 32000 subwords into the range of integers [0, 32000). The tokens are then embedded by a lookup table into a learned vector embedding space. A separator token is added between every subword token of the sequence. Learnable positional embeddings indicating position within the text sentence are added to all tokens. The final result is a sequence of embeddings where each embedding represents a subword token or separator token. The resultant sequence of text embeddings representing each caption is fed into a Bidirectional Encoder Representations from Transformers (BERT) model and a single output vector embedding is produced. Separately, each image is directly fed into a Residual Network (pre-trained weights from *resnet18* are used) and a single output vector embedding is produced. The outputs of the BERT and ResNet models are then concatenated together to form a single, contextual embedding with dimensions (1, 154880) that represents both the image and its caption.

2.4 Combined Vision Transformer Contextual Embeddings

The architecture of the Vision Transformer is identical to the Bidirectional Encoder Representations from Transformers (BERT) transformer, except it is trained on both text and image patch embeddings, as shown in Figure 2.2.

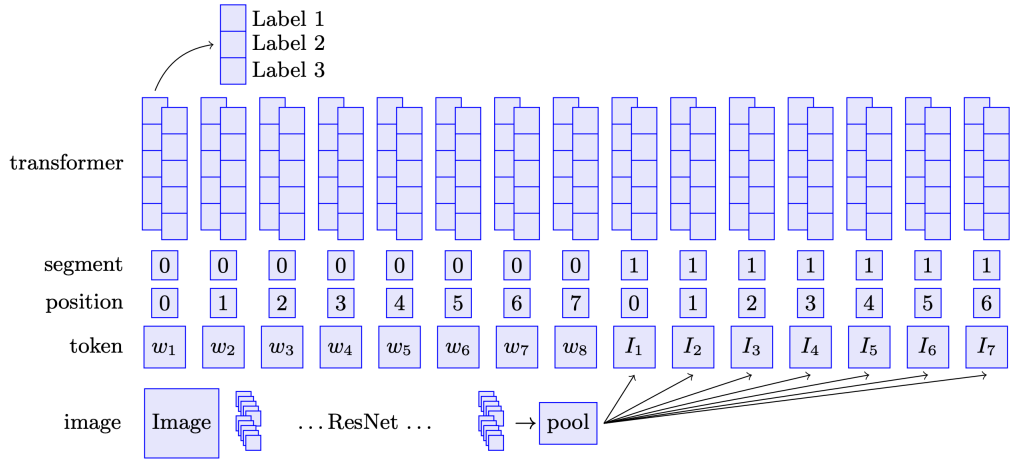


Figure 2.2: Diagram of the multimodal transformer architecture with concatenated sequence of subword and image embeddings. Image credit to (Douwe et al., 2020).

2.4.1 Input Transformations

The images and text from the image captioning dataset need to be transformed into a single sequence of embeddings that can be fed as input to the Vision Transformer contextual embedding function. The sequence of text embeddings for each caption are obtained through SentencePiece (Kudo & Richardson, 2018)[8] with 32000 sub-words into the range of integers [0, 32000) just as before. An image is tokenized and converted into a sequence of non-overlapping 16×16 patches in raster order (Dosovitskiy et al., 2020)[5]. Each pixel in the resulting image patches is normalized between $[1, 1]$ and divided by the square root of the patch dimension. The token patches are then embedded using a single ResNet[6] block to obtain a vector per patch. A learnable within-image position encoding vector is then added to the vector embedding obtained from the ResNet block, giving the final sequence of image embeddings. Since the original image is composed as a two dimensional grid of patches, each patch has a row and column position, which are normalized between $[0, 1]$ and then

discretized using uniform binning. The discretized positions are then used to index into learnable row and column positional embedding tables, giving positional embeddings that are added to the respective patch token embedding. The final result is a sequence of embeddings where each embedding represents an image patch.

2.4.2 Multimodal Input Transformation

Image captioning has both image and text data; when the Vision Transformer is used as a contextual embedding function, both modalities of data need to be fed as a single input sequence of embeddings. Hence, the final sequences of text and image embeddings described above, are concatenated together to form a single combined multimodal sequence (Douwe Kiela, 2020)[7] that is given to the Vision Transformer all at once as illustrated in Figure 2.2. This concatenated sequence is compatible with the self attention mechanism in each encoder block, since attention scores for each embedding of the input sequence with every other embedding in the same sequence are computed. Hence, the entire concatenated sequence is processed at once with the relationships (*image patches* $\longleftrightarrow$ *subwords*, *image patches* $\longleftrightarrow$ *image patches*, *subwords* $\longleftrightarrow$ *subwords*) being considered by the contextual embedding. In other words, the Vision Transformer will be able to interpret the input data at a granular level by relating specific subwords with specific image patches, making it possibly suitable for the image captioning tasks.

2.5 Classifier Models

Four different models are used for classification, including a linear support vector machine, non-linear support vector machine (polynomial kernel of degree 3), gaussian naive bayes, and a fully connected deep neural network.

A computer would deserve to be called intelligent if it
could deceive a human into believing that it was human.

Alan M. Turing

3

Experiments and Results

The classification efficacy on contextual embeddings for image captioning is described and analyzed in this chapter. Four different machine learning classifiers were employed on static and contextual embeddings of various dimensionalities. The false positive rate denotes the percentage of humans incorrectly classified as machines, while the false negative rate denotes the percentage of machines incorrectly classified as humans.

3.1 Disjoint SentencePiece and Residual Convolutional Neural Network Embeddings

The static caption embedding generated by SentencePiece has a dimension of $(1, 200)$. The contextual image embedding generated by ResNet-18 has a dimension of $(1, 512)$. The static caption embedding is concatenated with a separator with dimensions $(1, 10)$ containing the integer 1 and the contextual image embedding, forming a final static-contextual embedding with a dimension of $(1, 722)$.

	Linear SVM	**Non-Linear SVM**	**Naive Bayes**	**DNN**
Accuracy	**40.40%**	**74.29%**	**28.17%**	**71.43%**
False Positive	37.20%	79.20%	0.00%	100.00%
False Negative	68.56%	4.32%	99.76%	0.00%

Table 3.1: The classification accuracy, false positive rate and false negative rate for each model is listed in the table above. The non-linear SVM uses a polynomial kernel of degree 3. Each result is an average over 5 trials.

The non-linear SVM incorrectly classifies 79.20% of humans as machines compared to 37.20% for the linear SVM, a difference of 42% as shown in Table 3.1. However, the linear SVMincorrectly classifies 68.56% of machines as humans compared to 4.32% for the non-linear support vector machine, a difference of 64.24%. In other words, neither type of SVM performs well on classifying **both** humans and machines. The Gaussian Naive Bayes model essentially predicts all datapoints as humans and is unable to grasp any meaningful understanding of the dataset for more effective classification. Finally, the deep neural network predicts all datapoints as machines and similarly unable to learn any meaningful pattern within the dataset for effective classification.

3.2 Disjoint Encoder Transformer and Residual Convolutional Neural Network Contextual Embeddings

The contextual caption embedding generated by BERT has a dimension of (200, 768). The contextual image embedding generated by ResNet-18 has a dimension of (1, 512). The static caption embedding is concatenated with a separator with dimensions (1, 768) containing the integer 1 and the contextual image embedding, forming a final contextual embedding with a dimension of (1, 154880).

		Linear SVM	**Non-Linear SVM**	**Naive Bayes**	**DNN**
Original	Accuracy	**69.94%**	**69.54%**	**71.71%**	N/A
Original	False Positive	48.00%	100.00%	63.20%	N/A
Original	False Negative	22.88%	2.64%	14.32%	N/A
PCA 10	Accuracy	**71.03%**	**73.03%**	**74.69%**	**69.94%**
PCA 10	False Positive	92.80%	82.80%	67.20%	59.00%
PCA 10	False Negative	3.44%	4.64%	8.56%	18.48%
PCA 50	Accuracy	**73.66%**	**74.00%**	**71.31%**	**72.97%**
PCA 50	False Positive	79.80%	75.60%	65.60%	46.40%
PCA 50	False Negative	4.96%	6.16%	13.92%	19.28%
PCA 100	Accuracy	**73.71%**	**74.46%**	**71.49%**	**74.63%**
PCA 100	False Positive	79.20%	73.40%	66.60%	49.40%
PCA 100	False Negative	5.12%	6.40%	13.28%	15.76%

Table 3.2: The classification accuracy, false positive rate and false negative rate for each model is listed in the table above. The non-linear SVM uses a polynomial kernel of degree 3. Each result is an average over 5 trials. The DNN was not trained on the original contextual embeddings due to memory constraints, so the results are left as N/A.

The first column of Table 3.2 denotes the dimensionality of the contextual embedding, where original embedding has dimensions (1, 154880) and then the PCA reduced embeddings have dimensions (1, 10), (1, 50) and (1, 100). One of the first observations that can be made is that the original BERT and ResNet-18 contextual embedding seems to be much **more linearly separable** than its SentencePiece and ResNet-18 static-contextual counterpart. This is because the linear support vector machine has an accuracy of nearly 70% compared to 40.40% from before, where the false positive rate is 10.2% higher but the false negative rate is 45.68% lower. This seems to imply that contextual embeddings generated by pre-trained deep learning models are distilled with richer information about the original data than static embeddings, such that the differences between human and machine generated data are made clearer and therefore more linearly separable. Applying PCA dimensionality reduction on the contextual embeddings seems to cause information loss, removing the linear separability as the linear support vector machine ends up misclassifying the majority of humans as machines. We see that as more principle components are included, the linear support vector machine has a higher accuracy and a lower false positive rate.

We must note that the dataset contains 5 machine and 2 human generated captions for each image, so 71.43% of the dataset is comprised of machine labelled image-caption pairs. Hence, **two models that both have an accuracy of roughly** 70% **may have very different classification efficacies**, as one may only be predicting almost all datapoints as machines (e.g. non-linear SVM on the original embeddings with 100% false positive rate) while the other actually manages to correctly classify more than half the humans in addition to a large number of machines as well (e.g. linear SVM on original embeddings and deep neural network on PCA 50 and 100). With this concept of imbalanced data in mind,

the non-linear SVM performs much worse than the linear SVM on the original embeddings, overfitting and predicting nearly all datapoints as machines.

When training the fully connected deep neural network, applying PCA dimensionality reduction on the contextual embeddings is required in order to satisfy computation and memory constraints. The fully connected deep neural network is able to classify the PCA reduced embeddings with even better accuracy than the linear support vector machine on the original embeddings, obtaining an accuracy of 74.63% with 100 principle components and a false positive rate of 49.40%. Overall, the deep neural network trained on the PCA reduced data and the linear support vector machine trained on the original contextual embeddings have the best classification accuracy with the least lopsided false positive and false negative rates, outperforming both the non-linear support vector machine and naive bayes classifiers.

3.3 Combined Vision Transformer Contextual Embeddings

The final contextual embedding outputted by the Vision Transformer has a dimension of $(4299, 768)$, where there are 4299 token embeddings in the sequence and each token embedding has 768 values. Flattening this contextual embedding results in a vector of dimension $(1, 3301632)$ which introduces computational challenges for classifier training. Hence, we perform PCA dimensionality reduction with 20 principle components to reduce the contextual embedding size from $(4299, 768)$ to $(4299, 20)$, and then flattening it to $(1, 85980)$. The classifiers are also trained on embeddings obtained from a second round of PCA reduction, from $(1, 85980)$ to $(1, 100)$.

		Naive Bayes	DNN
PCA $(4299, 768) \rightarrow (4299, 20) \rightarrow (1, 85980)$	Accuracy	**26.57%**	**71.4%**
PCA $(4299, 768) \rightarrow (4299, 20) \rightarrow (1, 85980)$	False Positive	85.60%	100.00%
PCA $(4299, 768) \rightarrow (4299, 20) \rightarrow (1, 85980)$	False Negative	68.40%	0.00%
PCA $(1, 85980) \rightarrow (1, 100)$	Accuracy	**69.09%**	**70.06%**
PCA $(1, 85980) \rightarrow (1, 100)$	False Positive	99.80%	100.00%
PCA $(1, 85980) \rightarrow (1, 100)$	False Negative	3.34%	1.92%

Table 3.3: The classification accuracy, false positive rate and false negative rate for each model is listed in the table above. Each result is an average over 5 trials.

Both the Gaussian Naive Bayes and deep neural network classifier perform poorly on the VisionTransformer generated contextual embeddings, overfitting and predicting almost all datapoints as machines, simply demonstrating a lack of understanding of the underlying data. Two main factors contribute to these results. Firstly, the pre-trained weights for the VisionTransformer are only trained on image patch embeddings and do not contain any understanding of text data. Secondly, reducing the original contextual embedding with 3301632 values to 85980 and 100 values through PCA may have caused severe information loss in the resultant embeddings.

4

Conclusion

The application of deep learning models as contextual embedding functions for producing enriched embeddings to allow an AI-based Turing Test judge to more effectively differentiate between human and machine generated image captions was explored in this thesis. In particular, the Bidirectional Encoder Representations Transformer (BERT) model, residual convolutional neural network (ResNet-18), and Vision Transformer mod-

els were used to generate contextual embeddings image-caption pairs in the image captioning dataset. The classification results from four different machine learning models — namely, linear support vector machines, non-linear support vector machines, Gaussian Naive Bayes and fully connected deep neural networks — were then compared between various static and contextual embeddings. The experimental results showed that the disjoint BERT and ResNet-18 contextual embeddings were much more linearly separable than the disjoint SentencePiece and ResNet-18 static-contextual embeddings, indicating that contextual embeddings generated by pre-trained deep learning models more enriched with relevant information about the original data than static embeddings, such that the differences between human and machine generated data are made clearer and therefore more linearly separable. PCA dimensionality reduction is also applied on the contextual embedding space to alleviate memory and resource constraints for training the AI-based judge for human-machine classification. The experiments showed that the deep neural network trained on the PCA reduced data and the linear support vector machine trained on the original contextual embeddings had the best classification accuracy with the least lopsided false positive and false negative rates, outperforming both the non-linear support vector machine and gaussian naive bayes classifiers. Overall, classifiers perform better on the contextual embeddings than the static embeddings that are merely fixed representations for the input data that do not account for the contextual nuances of each datapoint.

There are several avenues of further research that can be taken. Firstly, the classification efficacy for the contextual embedding combination of disjoint BERT for captions and Vision Transformer for images can be evaluated and compared with the disjoint BERT for captions and ResNet-18 for images combination that was investigated in this thesis. Secondly, a multimodal transformer model that was pre-trained on both text and image data

should replace the Vision Transformer model that was pre-trained only on images to generate combined contextual embeddings. Finally, classification models such as decision trees and random forests can be employed for potentially more interpretable results.

References

[1] Harsh Agrawal, Karan Desai, Yufei Wang, Xinlei Chen, Rishabh Jain, Mark Johnson, Dhruv Batra, Devi Parikh, Stefan Lee, and Peter Anderson. nocaps: novel object captioning at scale. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, oct 2019.

[2] Corinna Cortes and Vladimir Naumovich Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995.

[3] Yin Cui, Guandao Yang, Andreas Veit, Xun Huang, and Serge Belongie. Learning to evaluate image captioning, 2018.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[5] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[7] Douwe Kiela, Suvrat Bhooshan, Hamed Firooz, Ethan Perez, and Davide Testuggine. Supervised multimodal bitransformers for classifying images and text, 2020.

[8] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing, 2018.

[9] Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation, 2022.

[10] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015.

[11] Ron Mokady, Amir Hertz, and Amit H. Bermano. Clipcap: Clip prefix for image captioning, 2021.

[12] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A generalist agent, 2022.

[13] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[15] Jianfeng Wang, Zhengyuan Yang, Xiaowei Hu, Linjie Li, Kevin Lin, Zhe Gan, Zicheng Liu, Ce Liu, and Lijuan Wang. Git: A generative image-to-text transformer for vision and language, 2022.

[16] Peng Wang, An Yang, Rui Men, Junyang Lin, Shuai Bai, Zhikang Li, Jianxin Ma, Chang Zhou, Jingren Zhou, and Hongxia Yang. Ofa: Unifying architectures, tasks, and modalities through a simple sequence-to-sequence learning framework, 2022.

[17] Mengmi Zhang, Giorgia Dellaferrera, Ankur Sikarwar, Marcelo Armendariz, Noga Mudrik, Prachi Agrawal, Spandan Madan, Andrei Barbu, Haochen Yang, Tanishq Kumar, Meghna Sadwani, Stella Dellaferrera, Michele Pizzochero, Hanspeter Pfister, and Gabriel Kreiman. Human or machine? turing tests for vision and language, 2022.