**Transformer Architecture Review**

Neal Ryan

Contents

[Abstract 3](#_Toc150209539)

[Transformer Overview 3](#_Toc150209540)

[Word Embedding 4](#_Toc150209541)

[Positional Encoding 4](#_Toc150209542)

[Attention 5](#_Toc150209543)

[Intuition 5](#_Toc150209544)

[Mathematical Explanation of Self-Attention 5](#_Toc150209545)

[Mathematical Explanation of Multi-Head Attention 6](#_Toc150209546)

[Intuition: MultiHead matrix 7](#_Toc150209547)

[Encoder 8](#_Toc150209548)

[Decoder 8](#_Toc150209549)

[Masking 8](#_Toc150209550)

[Decoder MultiHead Attention 9](#_Toc150209551)

[ChatGPT and the Transformer Decoder 9](#_Toc150209552)

[Conclusion 12](#_Toc150209553)

[Appendix 13](#_Toc150209554)

[Glossary of terms 13](#_Toc150209555)

[NLP: Natural Language Processing. 13](#_Toc150209556)

[Bag of words representation 13](#_Toc150209557)

[Token 13](#_Toc150209558)

[Word embedding 13](#_Toc150209559)

[Vector 14](#_Toc150209560)

[Matrix 14](#_Toc150209561)

[Tensor 14](#_Toc150209562)

[Matrix/Linear Algebra 15](#_Toc150209563)

[Weight Matrix 15](#_Toc150209564)

[Neural Network 16](#_Toc150209565)

[Feedforward and Backpropagation 17](#_Toc150209566)

[Normalization 18](#_Toc150209567)

[References 18](#_Toc150209568)

[Additional Sources 18](#_Toc150209569)

# Abstract

Transformer models have completely changed natural language processing (NLP), appearing much closer to general artificial intelligence than predecessor models like recurrent neural networks, LSTM’s, or GRUs. This review will discuss the Transformer model with little to no background required, explaining the components that make up this model, as well as the current state of the art (circa 2023) with a focus on Decoder only models (such as ChatGPT).

# Transformer Overview

In the seminal paper “Attention is all you need” researchers at Google laid out the fundamental building blocks for creating the transformer model (1: Viswani, A., et al.). While this architecture differs slightly from GPT’s, it is still a very informative model, and was the basis for transformer decoder models (like ChatGPT). The model architecture for the transformer is shown in figure 1 below:

A diagram of a software algorithm

Description automatically generated

Figure 1: Transformer model architecture

The transformer model has two subcomponents, an encoder (pictured on the left in figure 1) and the decoder (pictured on the right). We will walk through both in their respective sections below but let’s first discuss the preprocess steps to both units, namely, word embedding and positional encoding.

# Word Embedding

As laid out in the glossary, this is done to capture latent information in the word that may help to better represent the string’s meaning. Common techniques to perform this embedding are word2vec and GloVe. The dimensionality of the vector representation is up to the model creator, with longer vectors having more potential to grasp higher dimensions of latent information at the cost of extra compute.

It’s important to point out here that this word embedding dimensionality does not change between words. The same embedding dimensions are used for each word so that the input to the positional encoder is consistent.

Getting ahead of a misconception that may come up later, it is also important to note here that the sequence length given to the model can be viewed as the max sequence length the model can intake. Any portions of this sequence not used can then be padded with pad tokens. This is done to ensure that the sequence length of the input and output do not need to match.

# Positional Encoding

The main idea behind positional encoding is to allow the model to understand how far apart words are in the input. This is in stark contrast to older technologies like bag of words or LSTM’s/GRUs use of a sliding window (+/- some number of words from the current word). A slightly flawed pedological analogy is that this is akin to the sliding window if the window were infinite, allowing the model to see all other words that have come before this word.

This works by calculating a number that acts somewhat like an offset and adding this value to the value determined by the word embedding. Recall that our word embedding translates each word into a n-dimensional vector of our choosing. We then simply need to calculate an n-dimensional position value and add the two to output a new n-dimensional vector that contains both the word embedding and the positional encoding.

The Google researchers choose sinusoidal functions, alternating even (including 0) and odd dimensions of the vector, according to the following formulas:

Where dmodel=512 (the size of the word embedding)

The authors point out that this forms a geometric progression (ex: 1, 2, 4, 8, 16) of wavelengths from 2π to 20,000π. This means that as we go up in dimension of our positional encoding, the wavelength (distance between sinusoid peaks) increases steadily. Thus, the positionally encoding sinusoidal waves won’t be in phase with each other and we can ensure a unique encoding for our positions (though the value of these positional encodings may rarely overlap sometimes by coincidence). This is important to achieve the desired effect of adding learnable position distinguishment to the embeddings.

What does learnable mean? In this case, a mathematical function that generates the positional encoding value the neural network can learn to translate into position knowledge. The authors also point out that this choice of sinusoidal functions is not the only one capable of this job. Theoretically another geometric progression could be used, but these sinusoidal functions were likely chosen for their tendency to maintain amplitude (y) values between -1 and 1, allowing the values from this positional encoding to not completely dwarf the word embedding values at high i values.

# Attention

We now take what may seem like a diatribe to understand the key concept behind the transformer model: Attention. This concept actually predates the transformer(2: Bahdanau, D., et al.), and differs slightly from the MultiHead attention used in the transformer architecture but works well as a starting point. We will develop an intuition for what is going on, followed by explaining the math so figure 1 makes sense. This is really the key to understanding the transformer, and we will show that if attention is well understood, the rest of the transformer architecture sort of falls out as a byproduct.

## Intuition

Attention allows the model to “understand” sentences. By passing in the positionally encoded word vectors, we have abstracted representations of each word, as well as where it occurred in the sentence. This is great but doesn’t alone constitute a breakthrough. That comes from the idea of using this structure to learn from the data. Specifically, since we know the word vector representation carries the meaning of words at an n=512-dimensional level, we can then calculate where that word’s attention is. Take for example the phrase “The pizza came out of the oven hot, and it was delicious.” We know it represents another noun, but which? Using the old sliding window of an LSTM, we might assume oven, as that’s the closest noun. But wait, we have more information here. We know whatever “it” is also delicious. In our training corpus where we learned word embeddings, it’s unlikely that we ever saw “oven” and “delicious” in similar contexts (indicating they’d get similar embeddings). However, it’s very likely that “delicious” and “pizza” appeared in similar contexts. Thus, this positional encoding has “learned” that “it” should focus its attention on “pizza.”

## Mathematical Explanation of Self-Attention

As shown in figure 1, our input (position encoded + word embedded) now gets copied 3 times, giving us a total of 4 identical inputs. Three of these copies are sent to the attention unit (shown in figure 2 below), and are labeled Q (query), K (key), and V (value). To be clear, nothing about these matrices has changed, so they are still of dimension (sequence length, dmodel = 512). Focusing on just attention to start, the authors say Attention is given by the following expression:

Where KT is the transpose of the K matrix and dk is the model dimension of 512.

Since Q and K are the same matrix, multiplying Q by the transpose of K will yield a matrix of dimension (sequence length, sequence length). For every position in this matrix [i][j], the value represents a score of how similar the word[i] is to word[j]. Because we take the softmax (a function that often aids in classification by normalizing the output of multiclass calculations such that their sum adds to 1), each row of this matrix also adds to 1. As a result, this matrix can be thought of as a probabilistic similarity of word pairs, where for any row (word[i]), the similarities of all word[j]’s sums to 1. Readers who know some linear algebra will also note that this indicates the diagonal values of this matrix will be the largest in the matrix (corresponding to the idea that each word is most similar to itself).

It is also worth stopping here to ponder why the similarity matrix involves multiplying Q by KT. This intuition will again require some linear algebra knowledge but can be boiled down to the simple fact that the dot product of two vectors represents their similarity. This is a frequently employed tool in word to vector mapping to compute the similarity of two words. In this case, taking the transpose of the original Q matrix to yield KT and then performing Q\*KT is intuitively the same as taking each row of Q (a word embedding + position encoding), and finding it’s dot product with every other word embedding + position encoding (the columns of KT). Thus, each position in the Q\*KT yields the similarity of a word with another word as previously mentioned.

However, we’re not done. We now want to encode this attention back into our position encoded + word embedded input. We can’t currently do that, because our input vector is of shape (sequence length, sequence length) and our word embeddings + positional encodings are of shape (sequence length, dmodel). We can realign this shape through matrix multiplication, where again we notice that our similarity matrix (of dimension sequence length, sequence length) multiplied by V (of dimension sequence length, model size) will once again yield a matrix of size (sequence length, model size), which we call the attention matrix.

## Mathematical Explanation of Multi-Head Attention

As before, we split our input, this time into 4 units: Q, K, V. We will feed Q, K, and V into our model. Following a slightly different logic as self-attention described above, we multiply these matrices by three parameter matrices (WQ, WK, WV) of dimension (dmodel, dmodel), yielding transformed matrices Q’, K’, and V’ of dimension (sequence length, dmodel = 512). We will learn the weights to these parameter matrices (W matrices) in training. What does this do? As we covered earlier, without multiplying by these parameter matrices, the largest values will always lie along the diagonal (or with the word itself). This is not always desired, and these parameter matrices give the model a way to attend to other words, perhaps with higher scores than with itself.

A diagram of a product attention and multi-health

Description automatically generated

Figure 2 (Multi-Head Attention Unit)

Having generated Q’, K’, and V’, we now split them into smaller sub matrices, Q1-Qh. This split occurs over the dimension of the model, not the sequence, so the new dimension of each sub matrix (dk) will be the dimension of the model divided by the number of attention heads (h). In their work, the Google researchers used 8 heads, and 512/8 = dk = 64, so the dimension of the Q’, K’, and V’ submatrix becomes (sequence length, 64).

Now with these submatrices (Q1-Qh, K1-Kh, and V1-Vh), we can calculate their attention matrices, as described in the section above to arrive at the head matrix described in the paper:

where Q, K, and V are the original matrixes, W superscript Q, K, and V are the parameter matrices we learned from training, and the i represents the sub matrix splits we did for each attention head. Once again, this head attention matrix will have dimensions (sequence length, h\*dk = dmodel = 512). Finally, we will concatenate (join) all the head matrices, and multiply by WO another parameter matrix of dimension (h\*dk=dmodel, dmodel), as per the equation from the paper:

In the end, the MultiHead matrix is the same shape as our starting matrix (sequence length, dmodel).

## Intuition: MultiHead matrix

It may not be immediately clear why Q, K, and V are split into submatrices, so lets take a minute to explore this. Before this paper, attention models existed, but would use self-attention as described previously. This new multi-head model allows for a different level of attention. As noted in the mathematical explanation, Q’, K’, and V’ are split along the dimension of the model, which is the same as the dimension of the word embedding. Critically, also recall that no splitting is done on the dimension of the sequence length. What this means in practice is that each head sees all words in the sequence as well as a different part of each word’s embedding + position vector. This means that each head can focus on a different aspect of the ”meaning” of that word. In this sense, each head takes on an independent job helping to capture multiple types of possible relationships between the words (like syntactic relationships between subjects and verbs, or semantic relationships between words).

# Encoder

Now understanding the attention mechanism at play behind the curtain, explaining the rest of the encoder is straightforward. As previously mentioned, the input sequence comes into the encoder both embedded as well as positionally encoded. The transformer then computes the MultiHead matrix. Next this matrix is normalized and sent through a fully connected layer (the activation function of which is a ReLU) before this output is sent to the decoder. The weight matrix of this fully connected layer will be the parameters the encoder “learns,” and allows for tuning the emphasis on certain portions of the attention matrix that have importance. The ReLU or Rectified Linear Unit is a non-linear activation function that introduces non-linearity to the encoder. This increases the expressive power of the encoder because it enables learning more complex relationships within the MultiHead matrix.

It is worth stopping again here explore directionality. First, we note that the encoder is bidirectional (which conjures a somewhat antiquated idea that the input sequence is read both forward and backward). Bidirectional in this context means that the input sequence is read in in its entirety, and thus the encoder has access to all words at once. This is somewhat of a misnomer because it refers to older RNN/LSTM nomenclature where a sequence is read in word by word, so the encoder in these models only has access to words the proceed the current word being processed (thus making it unidirectional, because it only “sees” one way). The decoder is however unidirectional, the significance of which we will explore in the decoder section. This is a key point to understand, so it bears repeating: the encoder is bidirectional, and takes the entire input at once, while the decoder we will see is unidirectional, and therefore works token by token.

# Decoder

The decoder works fairly similarly to the encoder. We start by performing the same word embedding + position encoding process to the input. The input is still the same input the encoder received, so it is still of shape (sequence length, dmodel). This is where the decoder takes a small divergence from the encoder though. Despite receiving the entire input, the decoder generates tokens one at a time, taking the encoder output into consideration while predicting each token at t=1…sequence length time steps. To accomplish this, the input to the decoder is masked (hidden from the decoder). Let’s explore why.

## Masking

Note the stage following positional encoding in the decoder in figure 1 is a masked multi-head attention. This is typically done with a square matrix of shape (sequence length, sequence length) where a mask is applied at all values above the diagonal (ex: for a sequence length of three where represents an isMasked state, and that input will be set to ). As we mentioned in the encoder section, the decoder is unidirectional. This is because we cannot let the decoder see future output, otherwise we will run into reference problems (if the decoder can see future tokens, it might reference those tokens earlier in the sentence without context), as well as probability issues (future tokens will influence the selection of current tokens, but then could be changed in later steps, leading to nonsense generation).

Given our input matrix (shape: sequence length, dmodel), let’s think about what would happen when we perform masked attention. We would get the same MultiHead matrix as the encoder, but now we will multiply this matrix by our mask. The result of this is that at the first time step (the first token generation step), the decoder sees the all tokens of the input sequence, but any outputs generated beyond the first token are masked (hidden). Then at time step two, the decoder once again sees all of the input sequence but the outputs from the layer beyond the first two tokens are masked. This process continues for all time steps = sequence length.

## Decoder MultiHead Attention

With the masked MultiHead Attention computed, we now move to our second MultiHead Attention module in the decoder. If we look closely at figure 1, we can see that there are still three input tensors to this MultiHead Attention unit, but two are coming from the decoder. That math is still unchanged here, so don’t get confused. We are now just getting our keys and values (K and V from above) from the encoder output, and the queries (Q) comes from the masked MultiHead Attention. This is a special kind of attention called cross attention, the difference being that instead of the self attention we performed in the encoder (where the attention is on tokens in the same sequence), now the attention is on tokens in a different sequence (the decoders output sequence). Nothing changes other than the nomenclature though. We’ve seen this process before, and we know the output after all of the attention calculation will be an output of size (sequence length, dmodel).

That wasn’t really the goal though, right? The goal was to predict some token in our vocabulary at all time steps. This is where the last two operations come in. We first pass our output through a linear layer to undo our embedding projecting this back to our vocab\_size dimension. This result will be of shape (sequence length, vocab\_size), and roughly represents how likely a word is to be in each position. We follow this up by computing the SoftMax over the vocab\_size dimension to convert all the values in the dimension into probabilities. In English: we have computed a matrix where at time step t, we have t vocabulary probability vectors. We can now just find the maximum value in each of these vocabulary vectors to find the most likely token to occupy each spot in the sequence.

# ChatGPT and the Transformer Decoder

ChatGPT is a Transformer Decoder (or decoder-only transformer)[[1]](#endnote-1), meaning it does not use an encoder at all, as shown in figure 3(5: OpenAI). Other such models include PaLM from Google, Chinchilla from Deepmind, and LLaMa from Meta.

A screenshot of a computer

Description automatically generated

Figure 3: Transformer Decoder Architecture used by ChatGPT

Despite this difference, we have seen all these components before, so there is no new math or concepts to discuss here. Rather, we’ll focus on the intuition of what this difference means in practice.

The first notable difference is that without an encoder, we can see that the decoder receives only masked attention at each time point during inference. Contrast this with the general Transformer whose keys and values come from the encoder (which sees the whole unmasked sequence). In the literature, this one by one token generation using the tokens that proceed the current token to be generated is called autoregressive generation. It functions well on text generation tasks, and thus has risen to popularity via Open AI’s chat GPT models.

The natural question that arises is why use only the decoder? Surely there is a benefit to both the encoder and the decoder. This is unfortunately the point where the answers stop, and we’re left with only conjecture, as this is still an open question in the literature (4: Fu, Z. et al.). However, that isn’t to say we cannot develop some intuition. We know the encoder outputs a tensor of the same shape as the encoded input. Mathematically, this implies we are just messing with the numbers a bit, a regression task. Contrast this with the decoder which returns a tensor with shape sequence length, vocab size. This tells us the decoder transforms embeddings into words. We can see from this that the decoder is the one doing the generation, and thus is the only one strictly necessary for this task.

That’s just laziness though, right? Well, no. These models are already so complex that there really is a tradeoff adding features. Consider even a fairly “old” (as of the time of this writing: 2023) implementation of Open AI’s GPT model: GPT 3 (unfortunately Open AI is concealing more of the details for their latest implementation: GPT 4)(3: OpenAI). A table with numbers and letters

Description automatically generated

Figure 3: GPT model parameters

Chat GPT 3’s 175 billion parameters is already entirely unwieldy. In the same paper, OpenAI gives an indication of just how expensive and time consuming the training process is, shown in figure 4 below:

A table of numbers and a number of objects

Description automatically generated with medium confidence

Figure 4: Training cost in time and compute power for various large language models (PF-days is petaflop/s-day)

It’s no exaggeration to say that the training of these models is a multi-million-dollar expense. This brings us back to our original question: why decoder-only? At least for now, the exorbitant training cost of adding billions of parameters to the training through the implementation of an encoder just isn’t worth it. As mentioned previously though, this is an active research area (a real understatement), and there are plenty of implementations using encoder-decoder architectures, as shown by figure 5:

A computer screen shot of a tree

Description automatically generated

Figure 5: Visual summary of the architecture tree of notable large language models(6: Yang J., et al)

While work into encoder-only models has died down somewhat, the proliferation of interest in this area means we will see plenty of new iterations of the technology that hopefully help to answer questions on a dominant architecture more rigorously.

# Conclusion

In this review, we took a deep dive into the math behind the transformer architecture, as well as thoroughly examining the attention mechanism that is key to the model’s success. In the years since the original publication of “Attention is all you need,” the transformer model has been used in just about every type of NLP task imaginable. It is safe to say this technology is revolutionary and has exciting potential to continue for further iteration. As training costs come down and models are allowed to grow (number of training parameters) it will be interesting to see if there is a resurgence of the encoder-decoder transformer, and what continued generations of GPT-like models can achieve.

# Appendix

# Glossary of terms

To set the stage we must first define some terms required to discuss neural networks in general, and NLP and transformers more specifically. Terms will also include acronyms used throughout the review to clear up any potential confusion.

## NLP: Natural Language Processing.

This refers generally to the ability of a model to understand human speech, but as it is a huge field of research. We will focus on just a few sub fields of NLP that are relevant to the Transformer.

One incredibly important task that will occur in the attention mechanism (though will not be formally identified) is the concept of semantics. It involves understanding context and relationships between words that help to clarify language. One of the main ideas of the attention mechanism is that it can help parse these connections to develop a deeper understanding of language that previous machine learning implementations.

NLP also includes preprocessing tasks to parse speech (break down words into component forms – e.g. stop, stopping, stopped -> stop), tag words (stop - verb). While it will not be relevant to the Transformer as we discuss it, know that this process is occurring in the background in order to create embeddings (discussed later).

## Bag of words representation

A type of representation of text data where all positional information is stripped from a sentence, and only word counts remain. Tokens are then generated by selecting the next most likely word that continues the sentence. This technique is useful in some fields, search being a good example, because grammar is not often part of the domain, and a series of sentences is not often input.

## Token

We will switch frequently between using ‘token’ and ‘word’ to represent a prediction of some component of the output at each time step. This is done for readability, though strictly speaking, every mention of ‘word’ should always be replaced by ‘token’. What is a token? It’s a word or part of a word. For instance, the sentence Tom’s book is nice might be replaced by <S> <tom> <’s> <book> <is> <nice> <.> <END>. Where <S> and <END> are special tokens to denote the start and end of a sequence, and Tom’s is broken into tom ‘s (notice the lower case and ‘s indicating possession). This is done to help the model learn, but obfuscates some intuition while reading, and isn’t germane to this discussion.

## Word embedding

Word embeddings transform a word into a machine understandable vector (explained below). A word can often contain more information than just it’s surface definition. For instance, the word night is definitionally the portion of the a 24 hour period where the sun is not out. It is the opposite of day. It also conjures images of darkness, the moon, and time (specifically late in the day/early in the morning). A word embedding attempts to capture all these concepts in a single (finite dimension) vector in order to be able to use that word more effectively.

## Vector

A mathematical term for representing more than one variable. If we are given both an x and a y, this finitely represents a point in two-dimensional space. This can either be useful for defining a point (say 3, 4) or a direction (with respect to the origin – the 0,0 point). In most machine learning you might see the (x, y) coordinates (3, 4) represented as a vector:

Where implicitly, the top number is the x variable’s value, and the bottom number is the y variable's value. To quash some confusion before it arises: a vector is not limited to two dimensions. It can be n-dimensional, where n is some finite number. So we could also represent an x, y, z datapoint with a vector, but would require a new vector for each point in our now three dimensional space. Moving beyond a single point brings us to the next term.

## Matrix

A matrix is a collection of vectors. To represent the idea that we have two points x, y: (1, 5) and (4, 6) we could use the matrix:

## Tensor

A tensor is to a matrix what a matrix Is to a vector. To understand what that means, let’s extend the x, y analogy. Let’s map the position of two team’s football players on a field, team blue and team red. Team blue has players at (1, 4), (2, 3), and (0, 5) where as team red has players at (1, 5), (1, 4), and (2, 4).

Written as vectors then:

Team blue [ [1, 4], [2,3], [0,5] ] where the outer [ ] is a matrix that encompasses the entire team, and the inner [ ] represents the x, y vectors of the players

Team red: [ [1,5], [1,4], [2, 4] ]

Finally, the entire field:

[ [ [1, 4], [2,3], [0,5] ],

[ [1,5], [1,4], [2, 4] ] ] where [ ] is a tensor that encompasses the entire field

This creates a 2x3x2 tensor where:

The first two (**2**x3x2) represent the outer most [ ] and represents the entire field of options – namely the selection of team red or blue.

The three (2x**3**x2) represents the middle [ ] and represents team members – or more concretely the selection of a specific player within the red or blue team

The last two (2x3x**2**) represents the inner most [ ] and represents the location of the specified player (x, y).

## Matrix/Linear Algebra

At this point, an understanding of linear algebra would be helpful, but we can proceed without it assuming the reader understands some algebra. Lets set up a system of equations:

X + Y = 3

2X + 3Y = 7

Let’s say X represents the cost of pasta, and Y represents the cost of pasta sauce. You have forgetful friends, so while the remember what they paid total, the don’t quite remember how much each item cost.

The first equation is the price paid by one friend for 1 box of pasta (X) and 1 container of pasta sauce (Y).

The second equation is the price paid by a different friend for 2 boxes of pasta (X) and 3 containers of pasta sauce (Y).

We want to understand the price of the pasta (X) and pasta sauce (Y). We could employ traditional algebra techniques at this point, solving for x, then solving for y. However, there is a much more efficient approach: linear algebra. This will let us solve for both variables simultaneously. That may not seem valuable now, because we’re only working with a few unknowns. However, as we scale the number of unknowns we’re solving for exponentially, it becomes incredibly important that this process happens simultaneously.

To do this, we can set up some a linear algebra equation.

Where the first vector is the information we desire. We can employ traditional algebra rational here, dividing both sides by the weight matrix (a matrix of the scalars of the x and y variables). This is the same as multiplying by the inverse of the matrix

Simplifying to

As shown, this approach (while potentially more cognitively complex) does not require solving for individual variables. This may seems relatively unimportant in the above example, but gains serious benefit when talking about systems with billions of variables. When might those occur? Well that’s our next term: neural networks.

## Weight Matrix

A weight matrix is a matrix that allows us to transform an input into an output. This term is used very frequently in machine learning, where weight matrices are learned to fit datasets for a particular problem. We can explore this concept through the problem we just solved.

The weight matrix here is and it uniquely solves this problem for any dollar value paid by our two friends. Lets adjust the problem such that the items obtained by our two friends don’t change, but the dollar values do. Instead, friend one pays $5 and friend two pays $12. We can use the same matrix to calculate the cost of X and Y by the following equation:

Simplifying to

We can verify the validity of this solution on our first set of equations:

To see it checks out. Thus, our weight matrix can solve any problem of this kind. So long as the quantities of items are not changed represents a universal solution to yield the cost of X and Y. Should the quantities change, we must learn a new weight matrix to compute the result.

In this case the data fits our weight matrix perfectly (a property of all n unknowns, n equations system where in this case n=2), so there is no error. However, it is often the case that we will not know all the features/properties required to perfectly describe an outcome, so the model learns weights of the weight matrix to minimize the error as much as possible (though the backpropagation algorithm discussed later).

## Neural Network

A collection of vectors, matrices, and tensors that act as an optimized system of linear equations to perform some specified function(s). A neural network extends the linear algebra we just performed to as many dimensions as we would like (assuming we have the compute power to run such a network). The network is said to be composed of layers, which are really just vectors, matrices, or tensors. The network is composed of an input layer (the tensor we know the values of), hidden layers (layers of tensors whose values we don’t set, but rather are learned to perform the action we desire), and an output layer (a layer that is commonly just a vector that represents the output we intend). Let’s give a concrete example:

We might train a neural network to perform pricing optimization for ice cream sales. We feed in a series of datapoints, each of which contains many different variables (temperature, location, flavor, etc). and the output is a price. What happened in the hidden layer(s)? The model learned how much to weight each factor of the input given. For instance, perhaps the model learns something simple, like temperature and price are proportional (i.e., we can charge more when it’s hot outside). It may also learn more complex details (given the location is X, we can charge more when the flavor is vanilla, but in location Y, we can charge more for chocolate).

## Feedforward and Backpropagation

As described in the Neural Network section, training the weighting factors of each of the input variables is a combination of two algorithms: Feedforward and Backpropagation. In our example, we fed in a series of datapoints, each of which was a three-dimensional vector (temperature, location, and flavor), as well as the average price we charged that day. Let’s say we want to solve a much bigger set of equations though, say with 50 datapoints. We can design a dense (fully connected) neural network that takes in 3-dimensional input (temperature, location, flavor) and passes each of these values to multiple nodes in the hidden layer. These connections each have a weight, which can be thought of as an importance factor. The greater the weight, the more important that piece of data is to the output value. Each node then performs some calculation (think of this as learning some information that informs the output variable, i.e. helps predict the price) on this value and passes that forward to the next node. At some point the next node will be the output, and we will be done, having successfully turned our set of temperature, location, and flavor data into a price. This is the Feedforward algorithm, turning our data into the desired output. But is that price prediction any good? If we’ve only fed it one datapoint, no, definitely not. In order to tune the model to our data (often referred to as training), an algorithm called backpropagation is often used.

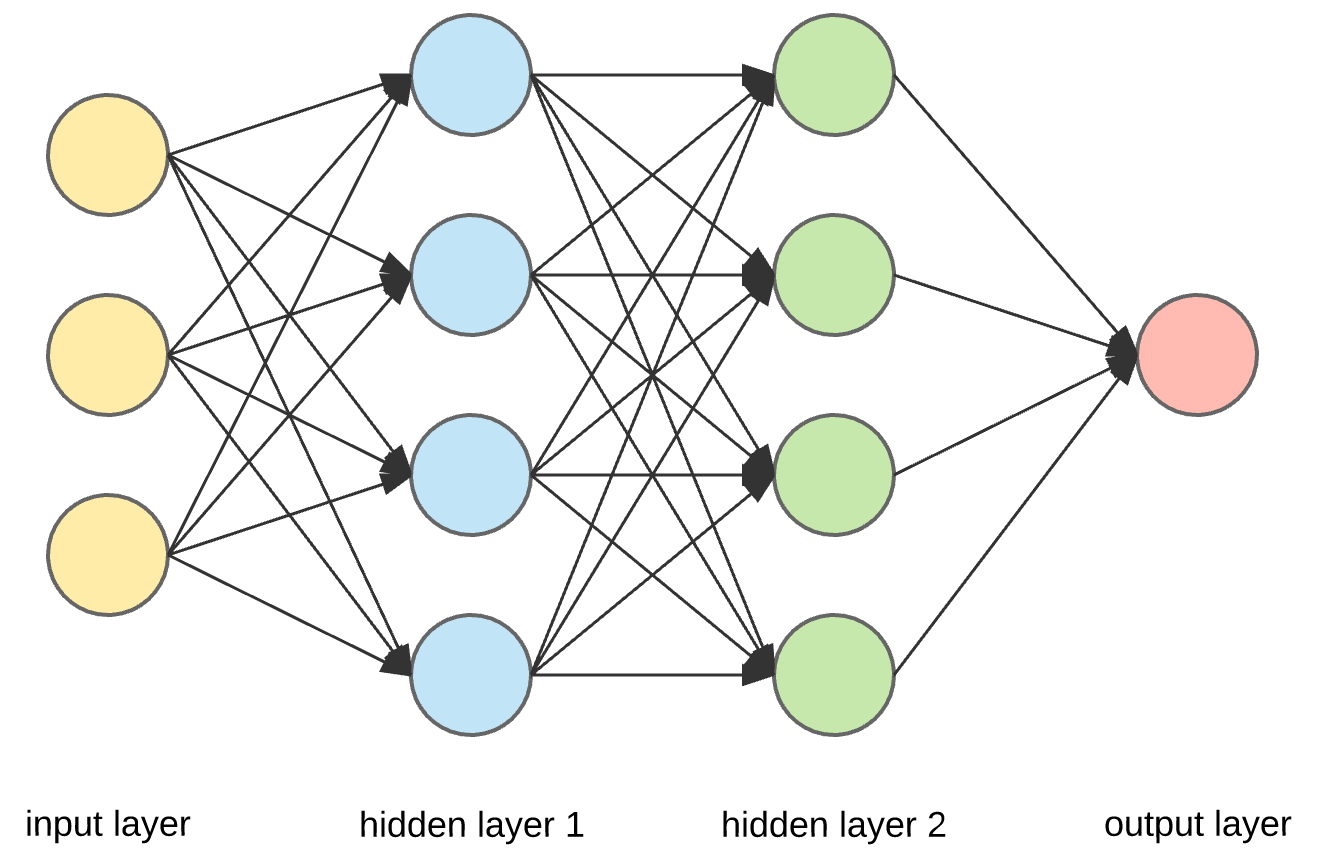


Figure 2: A neural network with 4 layers: 1 input, 2 hidden, and 1 output. Each circle represents a node. Note the network transforms a three dimensional vector (input) into a one dimensional vector (output). (Image source: https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a)

Backpropagation works by tuning the model to the data we have on hand to train it with. We mentioned before that we had 50 datapoints, where we know the temperature, location, and flavor as well as the average price we sold ice cream for. We can use that data to optimize the model. For each input set, we ask the model to predict a price. We then compare the prediction to the price we were able to get. If the model is spot on, we don’t need to adjust anything, that was a great prediction. However, if the model is off, we look at the magnitude of the error and adjust the weights of our model accordingly adjusting weights that contributed more to the error more by changing the values to a greater degree. By repeating this process on all 50 training examples, we can continuously tune the weights so that the prices given are as accurate as possible, leading to better results when we want to predict for real.

## Normalization

Normalization is the process of representing data in a consistent manner. When the feature vector for a model contains many different features sampled from different distributions, it is almost always the case that their variance is different. This poses a problem from simple calculations of weights in our weight matrix, as all statistical modeling techniques are basically an attempt to find a solution that minimizes the observed variance in the data. If variance means different things for different features, then features with very large variance will bias the modeling. There are a few ways to do this, but we will focus on the one used for the transformer work:

# References

1. Vaswani, A., et al. (2017). Attention Is All You Need. <https://arxiv.org/pdf/1706.03762.pdf>
2. Bahdanau, D., et al. (2016). Neural machine translation by jointly learning to align and translate. <https://arxiv.org/pdf/1409.0473.pdf>
3. Representations. OpenAI. (2022). Language Models are Few-Shot Learners. <https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf>
4. Fu, Z., et al. (2023). Decoder-Only or Encoder-Decoder? Interpreting Language Model as a Regularized Encoder-Decoder. <https://arxiv.org/pdf/2304.04052.pdf>
5. OpenAI. (2023). GPT-4 Technical Report. <https://arxiv.org/pdf/2303.08774.pdf>
6. Yang J., et al. (2023). Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond. https://arxiv.org/pdf/2304.13712.pdf

# Additional Sources

These videos were incredibly helpful and were used throughout this work to better understand the math and intuition behind each step of the transformer architecture

1. Jamil, U. (2023). Attention is all you need (Transformer) - Model explanation (including math), Inference and Training [Video]. YouTube. https://www.youtube.com/watch?v=bCz4OMemCcA&ab\_channel=UmarJamil
2. Starmer, J. (2023). Transformer Neural Networks, ChatGPT's foundation, Clearly Explained!!! [Video]. YouTube. <https://www.youtube.com/watch?v=zxQyTK8quyY>

1. [↑](#endnote-ref-1)