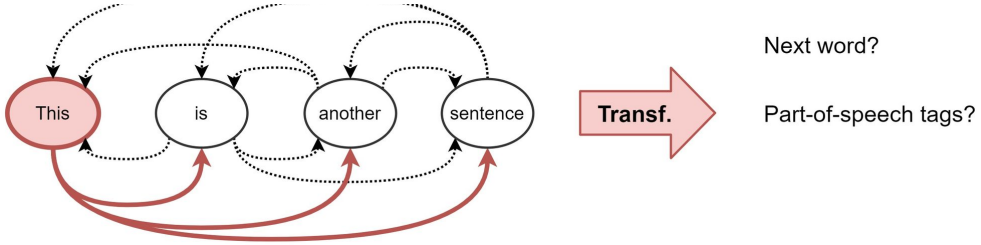
Primers • Transformers

Back to Top

Background: Representation Learning for NLP

- At a high level, all neural network architectures build representations of input data as vectors/embeddings, which encode useful syntactic and semantic information about the data. These latent or hidden representations can then be used for performing something useful, such as classifying an image or translating a sentence. The neural network learns to build better-and-better representations by receiving feedback, usually via error/loss functions.
- For Natural Language Processing (NLP), conventionally, Recurrent Neural Networks (RNNs) build representations of each word in a sentence in a sequential manner, i.e., one word at a time. Intuitively, we can imagine an RNN layer as a conveyor belt (as shown in the figure below; source), with the words being processed on it autoregressively from left to right. In the end, we get a hidden feature for each word in the sentence, which we pass to the next RNN layer or use for our NLP tasks of choice. Chris Olah's legendary blog for recaps on LSTMs and representation learning for NLP is highly recommend to develo background in this area

in the sentence are w.r.t. to the aforementioned word. Knowing this, the word's updated features are simply the sum of linear transformations of the features of all the words, weighted by their importance (as shown in the figure below; source). Back in 2017, this idea sounded very radical, because the NLP community was so used to the sequential–one-word-at-a-time–style of processing text with RNNs. As recommended reading, Lilian Weng's Attention? Attention! offers a great overview on various attention types and their pros/cons.



Enter the Transformer

- History:
  - LSTMs, GRUs and other flavors of RNNs were the essential building blocks of NLP models for two decades since 1990s.
  - CNNs were the essential building blocks of vision (and some NLP) models for three decades since the 1980s.
  - In 2017, Transformers (proposed in the "Attention Is All You Need" paper) demonstrated that recurrence and/or convolutions are not essential for building high-performance natural language models.
  - In 2020, Vision Transformer (ViT) (An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale) demonstrated that convolutions are not essential for building high-performance vision models.
- Transformers did not become a overnight success until GPT and BERT immensely popularized it. Here is a timeline of events:
  - Attention is all you need: 2017
  - ElMo (LSTM-based): 2018
  - ULMFiT (LSTM-based): 2018
  - GPT (Transformer-based): 2018

Back to Top

- The most advanced architectures in use before Transformers were Recurrent Neural Networks with LSTM/GRU. These architectures, however, have the following problems:
    - They struggle with really long sequences (despite using LSTM and GRU units).
    - They are fairly slow, as their sequential nature doesn't allow any kind of parallel computing.
- Transformers work differently:
    - They work on the entire sequence calculating attention across all word-pairs, which let them learn long-range dependencies.
    - Some parts of the architecture can be processed in parallel, making training much faster.
- Owing to their unique self-attention mechanism, transformer models offer a great deal of representational capacity/expressive power.
- Initially introduced for machine translation by Vaswani et al. (2017), Transformers have gradually replaced RNNs in mainstream NLP. The architecture takes a fresh approach to representation learning: Doing away with recurrence entirely, Transformers build features of each word using an attention mechanism to figure out how important **all the other words** in the sentence are w.r.t. the aforementioned word. As such, the word's updated features are simply the sum of linear transformations of the features of all the words, weighted by their importance.
- Back in 2017, this idea sounded very radical, because the NLP community was so used to the sequential – one-word-at-a-time – style of processing text with RNNs. The title of the paper probably added fuel to the fire! For a recap, Yannic Kilcher made an excellent video overview.
- Today, transformers are not just limited to language tasks but are used in vision, speech, and so much more. The following plot (source) shows the transformers family tree with prevalent models:

Back to Top

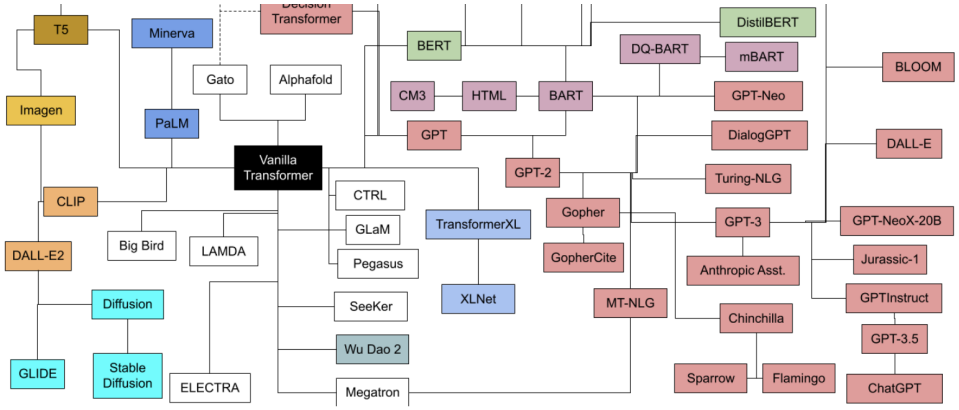


Figure 6: Transformers Family Tree

- And, the plots below (first plot source); (second plot source) show the timeline for prevalent transformer models:

Back to Top

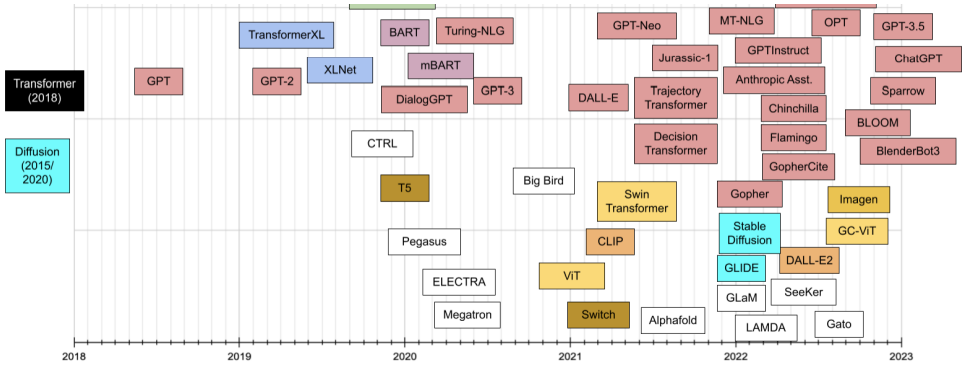
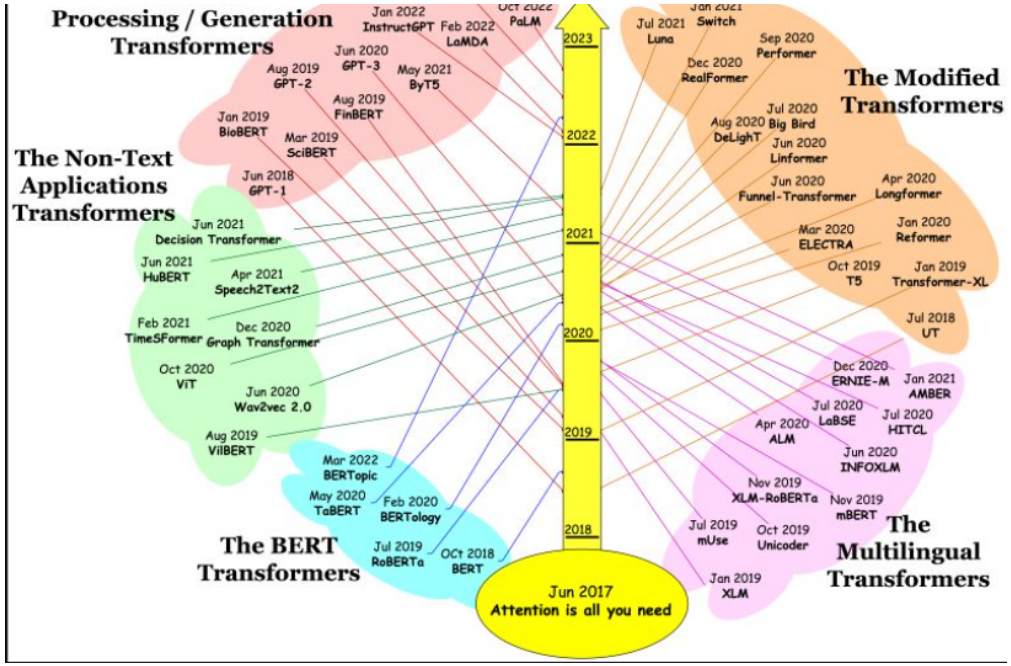


Figure 7: Transformer timeline. Colors describe Transformer family.

Back to Top



- Lastly, the plot below (source) shows the timeline vs. number of parameters for prevalent transformer models:

Back to Top

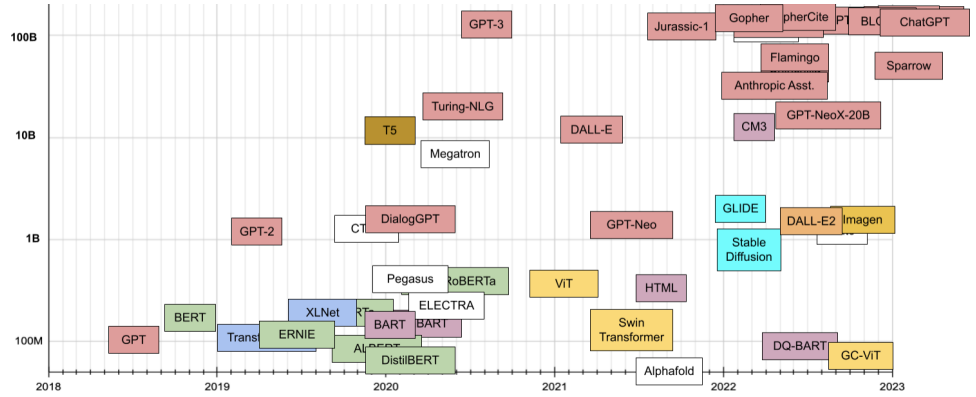


Figure 8: Transformer timeline. On the vertical axis, number of parameters. Colors describe Transformer family.

Transformers vs. CNNs

Language

- In a vanilla language model, for example, nearby words would first get grouped together. The transformer, by contrast, runs processes so that every element in the input data connects, or pays attention, to every other element. This is referred to as “self-attention.” This means that as soon as it starts training, the transformer can see traces of the entire data set.
- Before transformers came along, progress on AI language tasks largely lagged behind developments in other areas. Infact, in this deep learning revolution that happened in the past 10 years or so, natural language processing was a latecomer and NLP was, in a sense, behind computer vision, per the computer scientist Anna Rumshisky of the University of Massachusetts, Lowell.
- However, with the arrival of Transformers, the field of NLP has received a much-needed push and churned model after model that have beat the state-of-the-art in various NLP tasks.

words immediately around the “it”s would struggle, but a transformer connecting every word to every other word could discern that the owl did the grabbing, and the squirrel lost part of its tail.

Vision

> *In CNNs, you start off being very local and slowly get a global perspective. A CNN recognizes an image pixel by pixel, identifying features like corners or lines by building its way up from the local to the global. But in transformers, with self-attention, even the very first layer of information processing makes connections between distant image locations (just as with language). If a CNN’s approach is like starting at a single pixel and zooming out, a transformer slowly brings the whole fuzzy image into focus.*

- CNNs work by repeatedly applying filters on local patches of the input data, generating local feature representations (or “feature maps”) and incrementally increase their receptive field and build up to global feature representations. It is because of convolutions that photo apps can organize your library by faces or tell an avocado apart from a cloud. Prior to the transformer architecture, CNNs were thus considered indispensable to vision tasks.
- With the Vision Transformer (ViT), the architecture of the model is nearly identical to that of the first transformer proposed in 2017, with only minor changes allowing it to analyze images instead of words. Since language tends to be discrete, a lot of adaptations were to discretize the input image to make transformers work with visual input. Exactly mimicing the language approach and performing self-attention on every pixel would be prohibitively expensive in computing time. Instead, ViT divides the larger image into square units, or patches (akin to tokens in NLP). The size is arbitrary, as the tokens could be made larger or smaller depending on the resolution of the original image (the default is 16x16 pixels). But by processing pixels in groups, and applying self-attention to each, the ViT could quickly churn thro
enormous training data sets, spitting out increasingly accurate classifications.

Multimodal Tasks

- As discussed in the Enter the Transformer section, other architectures are “one trick ponies” while multimodal learning requires handling of modalities with different patterns within a streamlined architecture with a reasonably high relational inductive bias to even remotely reach human-like intelligence. In other words, we needs a single versatile architecture that seamlessly transitions between senses like reading/seeing, speaking, and listening.
- The potential to offer a universal architecture that can be adopted for multimodal tasks (that requires simultaneously handling multiple types of data, such as raw images, video and language) is something that makes the transformer architecture unique and popular.
- Because of the siloed approach with earlier architectures where each type of data had its own specialized model, this was a difficult task to accomplish. However, transformers offer an easy way to combine multiple input sources. For example, multimodal networks might power a system that reads a person’s lips in addition to listening to their voice using rich representations of both language and image information.
- With cross-attention where the query, key and value vectors are derived from different sources, transformers are able to lend themselves as a powerful tool for multimodal learning.
- The transformer thus offers be a big step toward achieving a kind of “convergence” for neural net architectures, resulting in a universal approach to processing data from multiple modalities.

Breaking Down the Transformer

- Before we pop open the hood of the Transformer and go through each component one by one, let’s first setup a background in underlying concepts such as one-hot vectors, dot product, matrix multiplication, embedding generation, and attention.

Back to Top

Overview

- Computers process numerical data. However, in most practical scenarios, the input data is not naturally numeric, for e.g., images (we model intensity values as pixels), speech (we model the audio signal as an oscillogram/spectrogram). Our first step is to convert all the words to numbers so we can do math on them.
- One hot encoding is a process by which categorical variables are converted into a form that could be provided to ML algorithms to do a better job in prediction.

Idea

- So, you're playing with ML models and you encounter this "one-hot encoding" term all over the place. You see the sklearn documentation for one-hot encoder and it says "encode categorical integer features using a one-hot aka one-of-K scheme." To demystify that, let's look at what one-hot encoding actually is, through an example.

Example: Basic Dataset

- Suppose the dataset is as follows:

CompanyName	Categoricalvalue	Price
VW	1	20000
Acura	2	10011
Honda	3	50000

to be another company in the dataset, it would have been given categorical value as 4. As the number of unique entries increases, the categorical values also proportionally increases.

- The previous table is just a representation. In reality, the categorical values start from 0 goes all the way up to $\mathrm{N}-1$ categories.
- As you probably already know, the categorical value assignment can be done using sklearn's LabelEncoder.
- Now let's get back to one-hot encoding: Say we follow instructions as given in the sklearn's documentation for one-hot encoding and follow it with a little cleanup, we end up with the following:

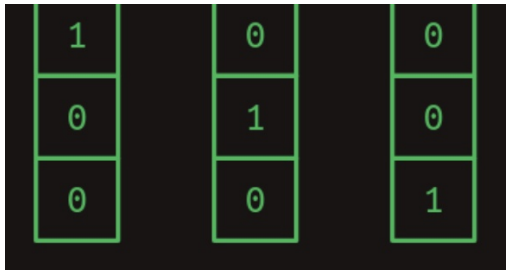
VW	Acura	Honda	Price
1	0	0	20000
0	1	0	10011
0	0	1	50000
0	0	1	10000

  - where `0` indicates non-existent while `1` indicates existent.
- Before we proceed further, could you think of one reason why just label encoding is not sufficient to provide to the model for training? Why do you need one-hot encoding?
- Problem with label encoding is that it assumes higher the categorical value, better the category. Specifically, what this form of organization presupposes is `VW > Acura > Honda` based on the categorical values. Say supposing your model internally calculates average, then accordingly we get, `1+3 = 4/2 = 2`. This implies that: Average of VW and Honda is Acura. This is definitely a recipe for disaster. This model's prediction would have a lot of errors.
- This is why we use one-hot encoder to perform "binarization" of the category and include it as a featu train the model.

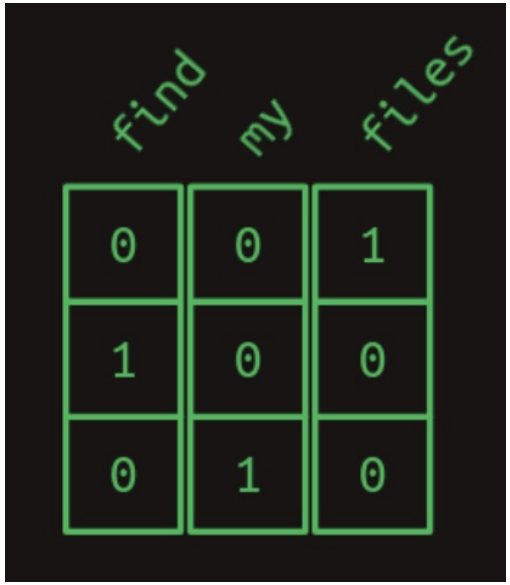
Example: NLP

- Inspired by Brandon Rohrer's Transformers From Scratch, let's consider another example in the context of natural language processing. Imagine that our goal is to create the computer that processes text, say a Machine Translation system that translates computer commands from one language to another. Such a model would ingest the input text and convert (or transduce) a sequence of sounds to a sequence of words.
- We start by choosing our vocabulary, the collection of symbols that we are going to be working with in each sequence. In our case, there will be two different sets of symbols, one for the input sequence to represent vocal sounds and one for the output sequence to represent words.
- For now, let's assume we're working with English. There are tens of thousands of words in the English language, and perhaps another few thousand to cover computer-specific terminology. That would give us a vocabulary size that is the better part of a hundred thousand. One way to convert words to numbers is to start counting at one and assign each word its own number. Then a sequence of words can be represented as a list of numbers.
- For example, consider a tiny language with a vocabulary size of three: files, find, and my. Each word could be swapped out for a number, perhaps `files = 1`, `find = 2`, and `my = 3`. Then the sentence "Find my files", consisting of the word sequence `[find, my, files]` could be represented instead as the sequence of numbers `[2, 3, 1].`
- This is a perfectly valid way to convert symbols to numbers, but it turns out that there's another format that's even easier for computers to work with, one-hot encoding. In one-hot encoding a symbol is represented by an array of mostly zeros, the same length of the vocabulary, with only a single element having a value of one. Each element in the array corresponds to a separate symbol.
- Another way to think about one-hot encoding is that each word still gets assigned its own number, but now that number is an index to an array. Here is our example above, in one-hot notation.

Back to Top



- So the phrase `find my files` becomes a sequence of one-dimensional arrays, which, after squeezing together, looks like a two-dimensional array.



- The terms “one-dimensional array” and “vector” are typically used interchangeably (in this article otherwise). Similarly, “two-dimensional array” and “matrix” can be interchanged as well.

Back to Top

Algebraic Definition

- The dot product of two vectors $\mathbf{a} = [a_1, a_2, \ldots, a_n]$ and $\mathbf{b} = [b_1, b_2, \ldots, b_n]$ is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

  - where Σ denotes summation and n is the dimension of the vector space.
- For instance, in three-dimensional space, the dot product of vectors $[1, 3, -5]$ and $[4, -2, -1]$ is:

$$\begin{aligned}[1, 3, -5] \cdot [4, -2, -1] &= (1 \times 4) + (3 \times -2) + (-5 \times -1) \\ &= 4 - 6 + 5 \\ &= 3\end{aligned}$$

- The dot product can also be written as a product of two vectors, as below.

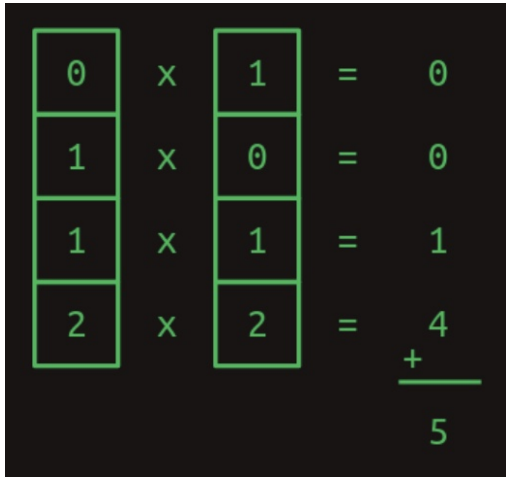
$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}\mathbf{b}^\top$$

  - where $\mathbf{b}^\top$ denotes the transpose of $\mathbf{b}$.
- Expressing the above example in this way, a 1×3 matrix (row vector) is multiplied by a 3×1 matrix (column vector) to get a 1×1 matrix that is identified with its unique entry:

$$\begin{bmatrix} 1 & 3 & -5 \end{bmatrix} \begin{bmatrix} 4 \\ -2 \\ -1 \end{bmatrix} = 3$$

- **Key takeaway**:

Back to Top



Geometric Definition

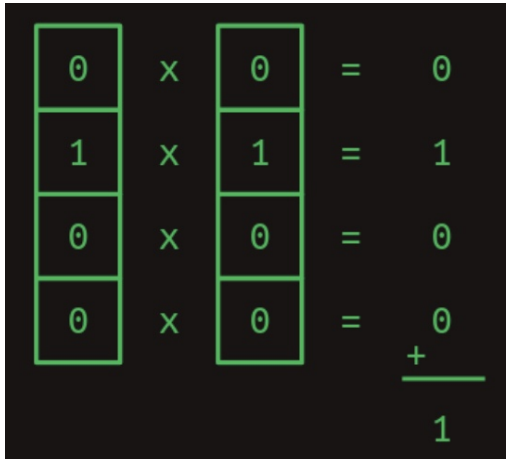
- In Euclidean space, a Euclidean vector is a geometric object that possesses both a magnitude and a direction. A vector can be pictured as an arrow. Its magnitude is its length, and its direction is the direction to which the arrow points. The magnitude of a vector a is denoted by $\|\mathbf{a}\|$. The dot product of two Euclidean vectors $\mathbf{a}$ and $\mathbf{b}$ is defined by,

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta$$

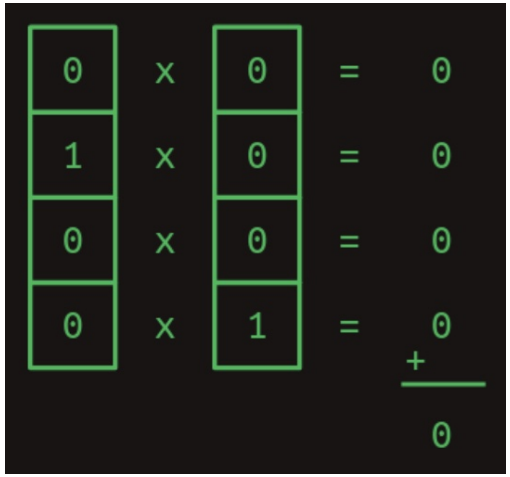
  - where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.
- The above equation establishes the relation between dot product and cosine similarity.

Properties of the Dot Product

Back to Top

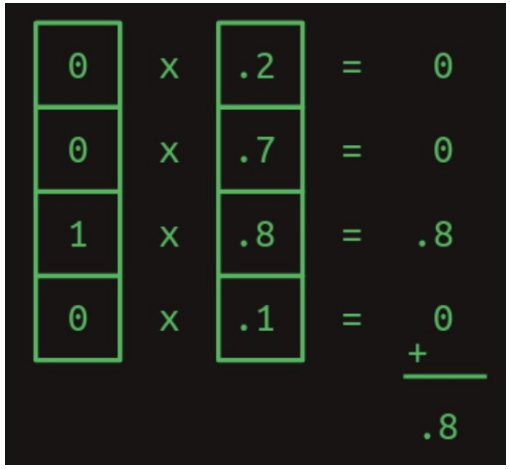


- The dot product of any one-hot vector with another one-hot vector is zero.



Back to Top

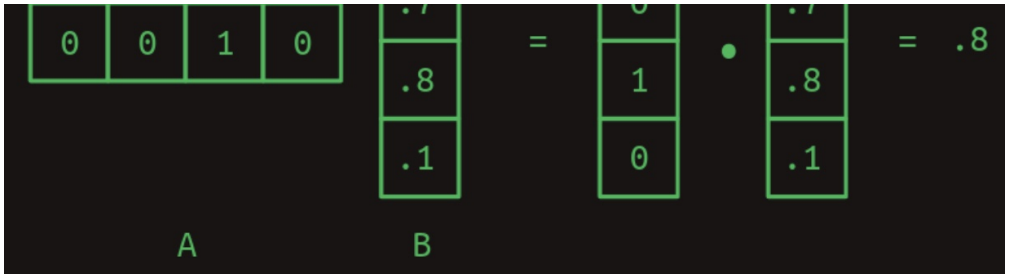
of calculating the dot product.



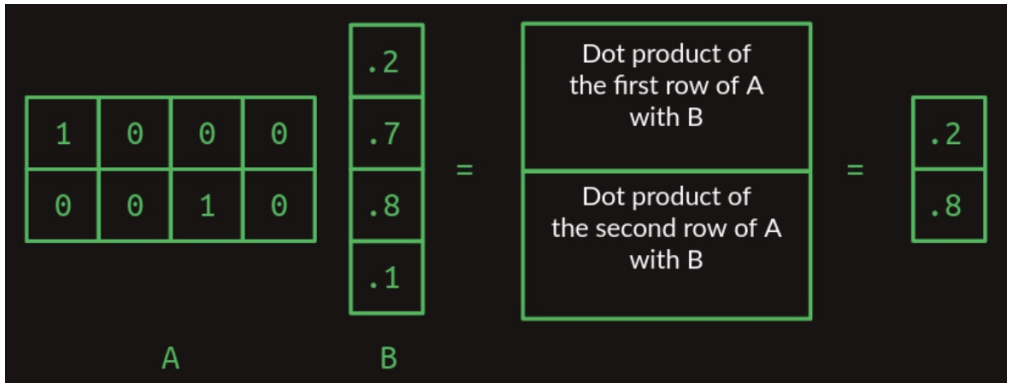
Matrix Multiplication As a Series of Dot Products

- The dot product is the building block of matrix multiplication, a very particular way to combine a pair of two-dimensional arrays. We'll call the first of these matrices $\mathbf{A}$ and the second one $\mathbf{B}$. In the simplest case, when $\mathbf{A}$ has only one row and $\mathbf{B}$ has only one column, the result of matrix multiplication is the dot product of the two. The following figure shows the multiplication of a single row matrix and a single column matrix.

Back to Top

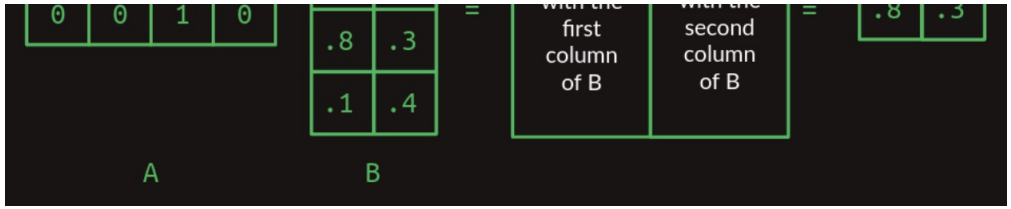


- Notice how the number of columns in A and the number of rows in $\mathbf{B}$ needs to be the same for the two arrays to match up and for the dot product to work out.
- When $\mathbf{A}$ and $\mathbf{B}$ start to grow, matrix multiplication starts to increase quadratically in time complexity. To handle more than one row in $\mathbf{A}$, take the dot product of $\mathbf{B}$ with each row separately. The answer will have as many rows as A does. The following figure shows the multiplication of a two row matrix and a single column matrix.



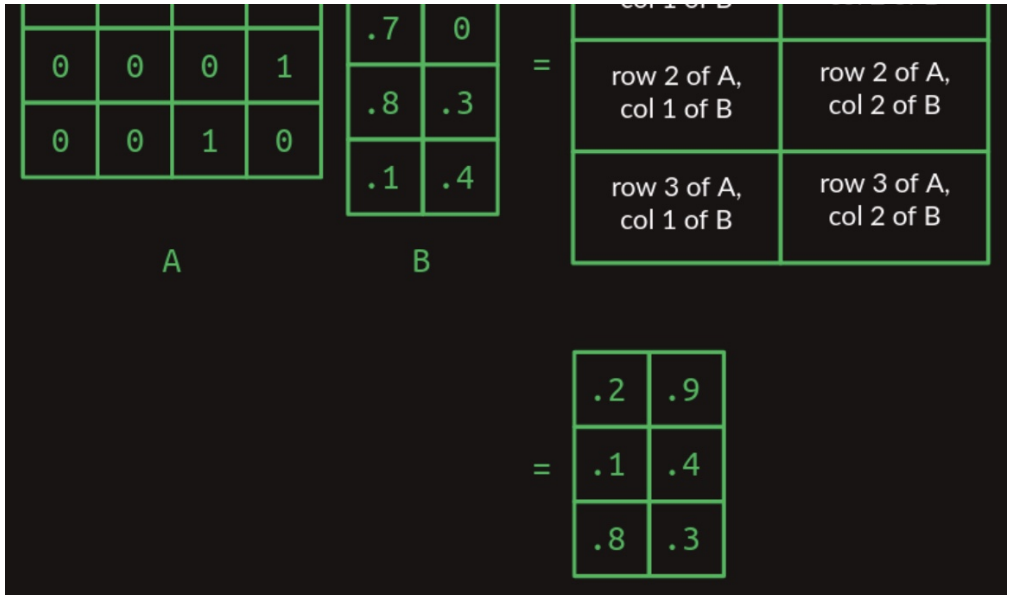
- When $\mathbf{B}$ takes on more columns, take the dot product of each column with $\mathbf{A}$ and stack the results in successive columns. The following figure shows the multiplication of a one row matrix and a two col matrix:

Back to Top



- Now we can extend this to mutliplying any two matrices, as long as the number of columns in $\mathbf{A}$ is the same as the number of rows in $\mathbf{B}$. The result will have the same number of rows as $\mathbf{A}$ and the same number of columns as $\mathbf{B}$. The following figure shows the multiplication of a one three matrix and a two column matrix:

Back to Top



Matrix Multiplication As a Table Lookup

- In the above section, we saw how matrix multiplication acts as a lookup table.
- The matrix $\mathbf{A}$ is made up of a stack of one-hot vectors. They have ones in the first column, the fourth column, and the third column, respectively. When we work through the matrix multiplication, this serves to pull out the first row, the fourth row, and the third row of the $\mathbf{B}$ matrix, in that order. This trick of using a one-hot vector to pull out a particular row of a matrix is at the core of how transformers work.

First Order Sequence Model

- We can set aside matrices for a minute and get back to what we really care about, sequences of wo
  Imagine that as we start to develop our natural language computer interface we want to handle just t

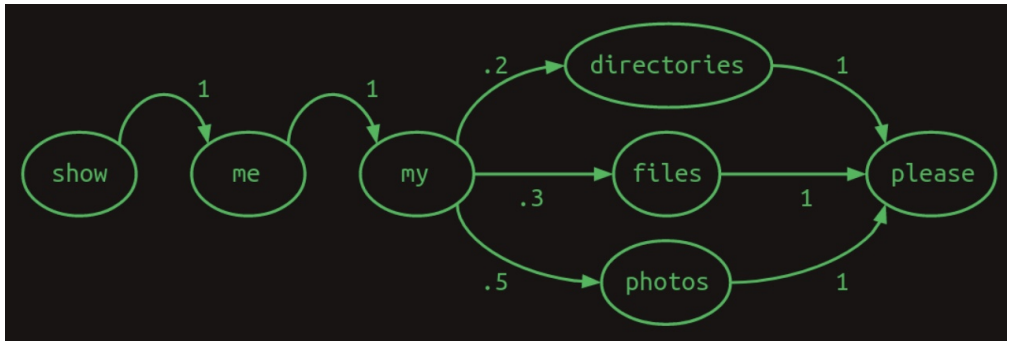
Back to Top

```
Show me my photos please.
```

- Our vocabulary size is now seven:

```
{directories, files, me, my, photos, please, show}
```

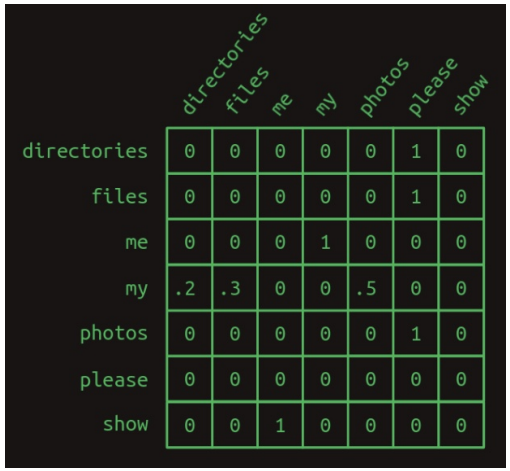
- One useful way to represent sequences is with a transition model. For every word in the vocabulary, it shows what the next word is likely to be. If users ask about photos half the time, files 30% of the time, and directories the rest of the time, the transition model will look like this. The sum of the transitions away from any word will always add up to one. The following figure shows a Markov chain transition model.



- This particular transition model is called a **Markov chain**, because it satisfies the Markov property that the probabilities for the next word depend only on recent words. More specifically, it is a first order Markov model because it only looks at the single most recent word. If it considered the two most recent words it would be a second order Markov model.

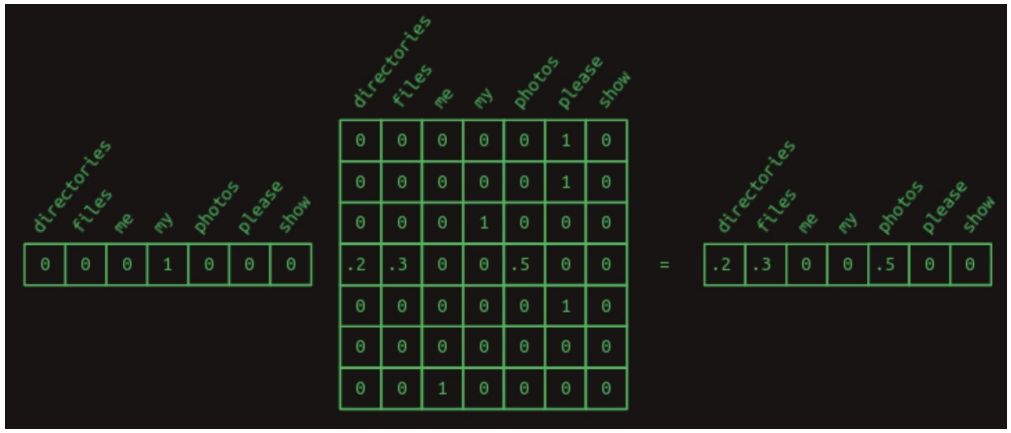
Back to Top

shows the probability of that word coming next. Because the value of each element in the matrix represents a probability, they will all fall between zero and one. Because probabilities always sum to one, the values in each row will always add up to one. The following diagram from Brandon Rohrer's Transformers From Scratch shows a transition matrix:



- In the transition matrix here we can see the structure of our three sentences clearly. Almost all of the transition probabilities are zero or one. There is only one place in the Markov chain where branching happens. After `my`, the words `directories`, `files`, or `photos` might appear, each with a different probability. Other than that, there's no uncertainty about which word will come next. That certainty is reflected by having mostly ones and zeros in the transition matrix.
- We can revisit our trick of using matrix multiplication with a one-hot vector to pull out the transition probabilities associated with any given word. For instance, if we just wanted to isolate the probabilities of which word comes after `my`, we can create a one-hot vector representing the word my and multiply it by our transition matrix. This pulls out the row the relevant row and shows us the probability distributio

Back to Top



Second Order Sequence Model

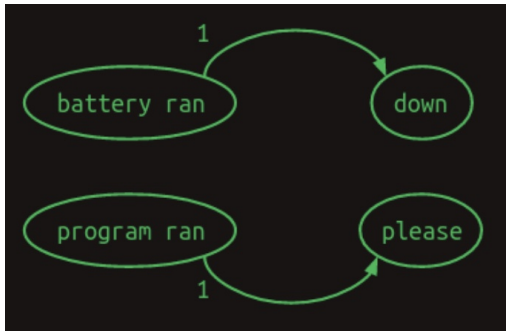
- Predicting the next word based on only the current word is hard. That's like predicting the rest of a tune after being given just the first note. Our chances are a lot better if we can at least get two notes to go on.
- We can see how this works in another toy language model for our computer commands. We expect that this one will only ever see two sentences, in a 40/60 proportion.

```
Check whether the battery ran down please.
Check whether the program ran please.
```

- A Markov chain illustrates a first order model for this. The following diagram from Brandon Rohrer's Transformers From Scratch shows a first order Markov chain transition model.

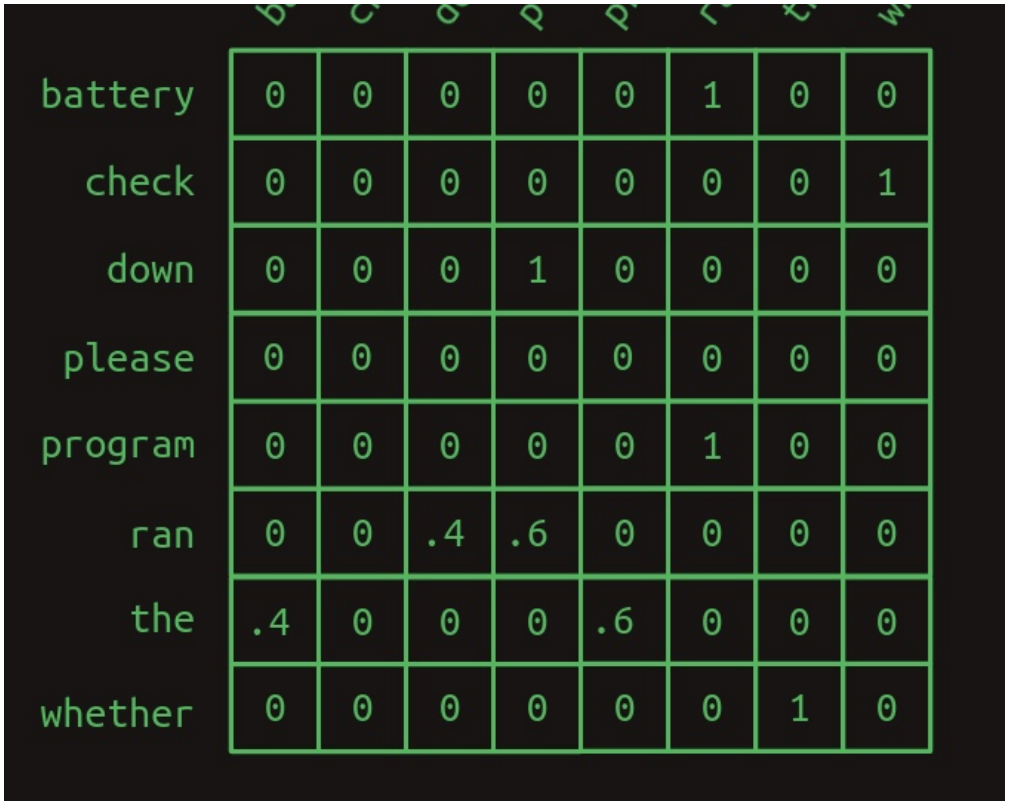
Back to Top

- Here we can see that if our model looked at the two most recent words, instead of just one, that it could do a better job. When it encounters `battery ran`, it knows that the next word will be down, and when it sees program ran the next word will be `please`. This eliminates one of the branches in the model, reducing uncertainty and increasing confidence. Looking back two words turns this into a second order Markov model. It gives more context on which to base next word predictions. Second order Markov chains are more challenging to draw, but here are the connections that demonstrate their value. The following diagram from Brandon Rohrer's Transformers From Scratch shows a second order Markov chain.



- To highlight the difference between the two, here is the first order transition matrix,
  - Here's a first order transition matrix:

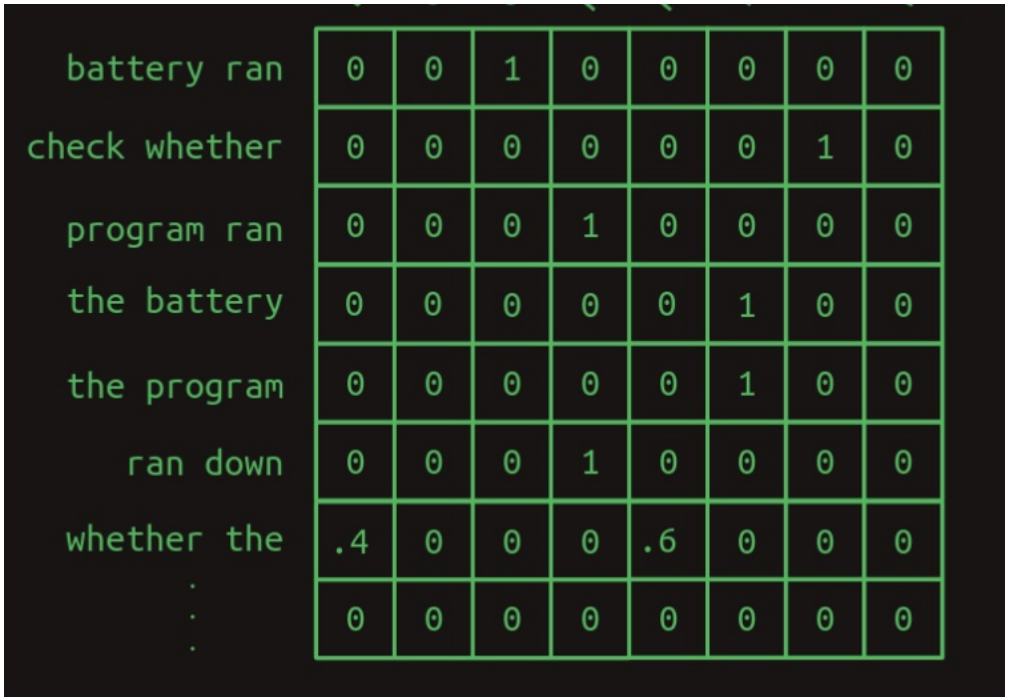
Back to Top



battery	0	0	0	0	0	1	0	0
check	0	0	0	0	0	0	0	1
down	0	0	0	1	0	0	0	0
please	0	0	0	0	0	0	0	0
program	0	0	0	0	0	1	0	0
ran	0	0	.4	.6	0	0	0	0
the	.4	0	0	0	.6	0	0	0
whether	0	0	0	0	0	0	1	0

- … and here is the second order transition matrix:

Back to Top



battery ran	0	0	1	0	0	0	0	0
check whether	0	0	0	0	0	0	1	0
program ran	0	0	0	1	0	0	0	0
the battery	0	0	0	0	0	1	0	0
the program	0	0	0	0	0	1	0	0
ran down	0	0	0	1	0	0	0	0
whether the	.4	0	0	0	.6	0	0	0
.								
.	0	0	0	0	0	0	0	0
.								

- Notice how the second order matrix has a separate row for every combination of words (most of which are not shown here). That means that if we start with a vocabulary size of $\mathbf{N}$ then the transition matrix has $\mathbf{N}^2$ rows.
- What this buys us is more confidence. There are more ones and fewer fractions in the second order model. There's only one row with fractions in it, one branch in our model. Intuitively, looking at two words instead of just one gives more context, more information on which to base a next word guess.

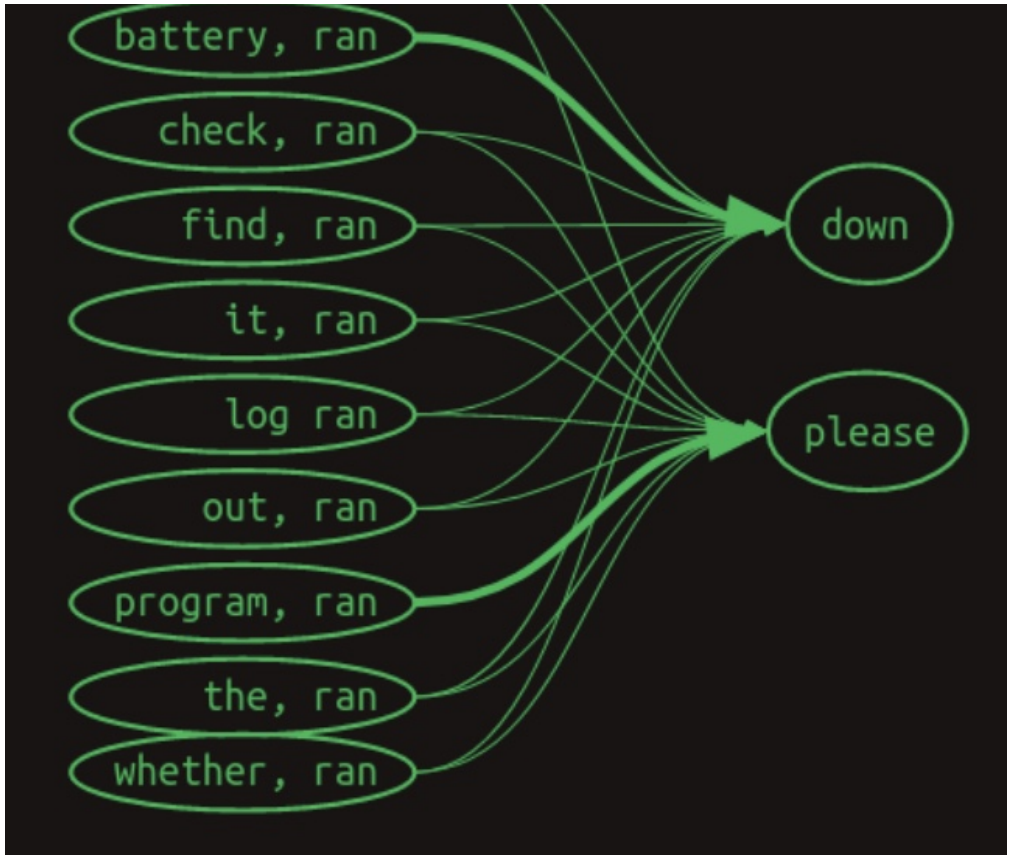
Second Order Sequence Model with Skips

- A second order model works well when we only have to look back two words to decide what word co next. What about when we have to look back further? Imagine we are building yet another langu

Back to Top

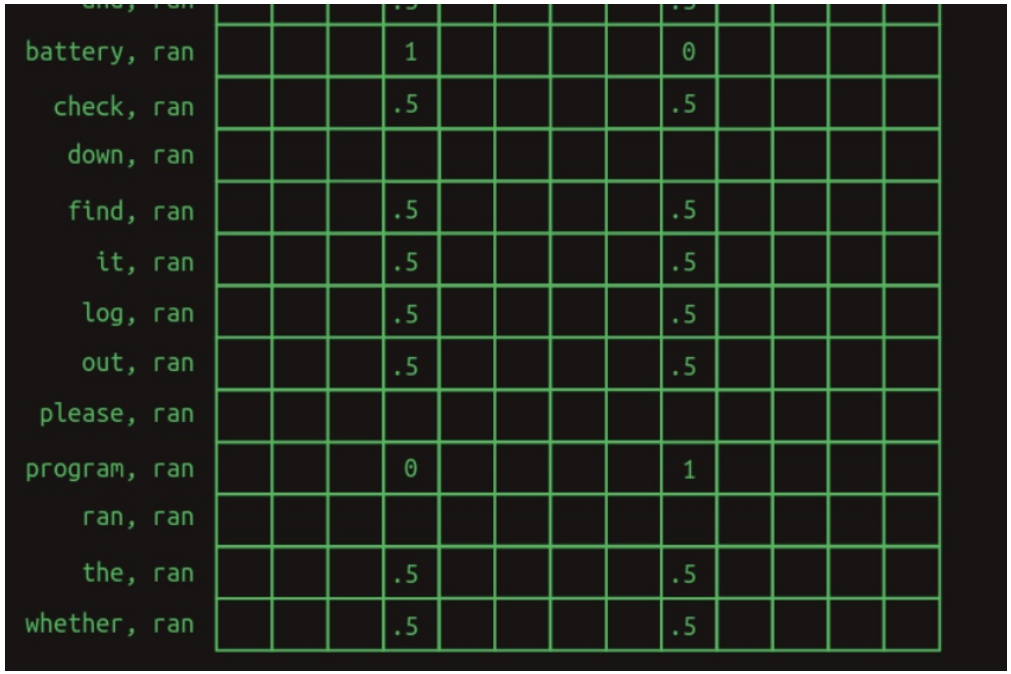
- In this example, in order to determine which word should come after ran, we would have to look back 8 words into the past. If we want to improve on our second order language model, we can of course consider third- and higher order models. However, with a significant vocabulary size this takes a combination of creativity and brute force to execute. A naive implementation of an eighth order model would have N^8 rows, a ridiculous number for any reasonable vocabulary.
- Instead, we can do something sly and make a second order model, but consider the combinations of the most recent word with each of the words that came before. It's still second order, because we're only considering two words at a time, but it allows us to reach back further and capture **long range dependencies**. The difference between this second-order-with-skips and a full umpteenth-order model is that we discard most of the word order information and combinations of preceding words. What remains is still pretty powerful.
- Markov chains fail us entirely now, but we can still represent the link between each pair of preceding words and the words that follow. Here we've dispensed with numerical weights, and instead are showing only the arrows associated with non-zero weights. Larger weights are shown with heavier lines. The following diagram from Brandon Rohrer's Transformers From Scratch shows a second order sequence model with skips feature voting.

Back to Top



- Here's what it might look like in a second order with skips transition matrix.

Back to Top



- This view only shows the rows relevant to predicting the word that comes after `ran`. It shows instances where the most recent word (`ran`) is preceded by each of the other words in the vocabulary. Only the relevant values are shown. All the empty cells are zeros.
- The first thing that becomes apparent is that, when trying to predict the word that comes after `ran`, we no longer look at just one line, but rather a whole set of them. We've moved out of the Markov realm now. Each row no longer represents the state of the sequence at a particular point. Instead, each row represents one of many **features** that may describe the sequence at a particular point. The combination of the most recent word with each of the words that came before makes for a collection of applicable rows, maybe a large collection. Because of this change in meaning, each value in the matrix no longer represents a probability, but rather a vote. Votes will be summed and compared to determine next w
predictions.

and that `battery` occurred somewhere earlier in the sentence. This feature has a weight of 1 associated with down and a weight of 0 associated with please. Similarly, the feature program, ran has the opposite set of weights. This structure shows that it is the presence of these two words earlier in the sentence that is decisive in predicting which word comes next.

- To convert this set of word-pair features into a next word estimate, the values of all the relevant rows need to be summed. Adding down the column, the sequence Check the program log and find out whether it ran generates sums of 0 for all the words, except a 4 for `down` and a 5 for `please`. The sequence `Check the battery log and find out whether it ran` does the same, except with a 5 for `down` and a 4 for `please`. By choosing the word with the highest vote total as the next word prediction, this model gets us the right answer, despite having an eight word deep dependency.

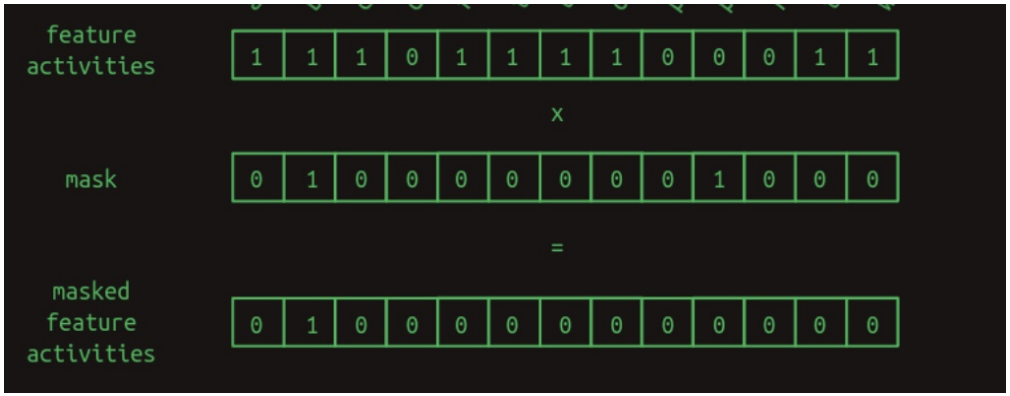
Masking Features

- On more careful consideration, this is unsatisfying – the difference between a vote total of 4 and 5 is relatively small. It suggests that the model isn't as confident as it could be. And in a larger, more organic language model it's easy to imagine that such a slight difference could be lost in the statistical noise.
- We can sharpen the prediction by weeding out all the uninformative feature votes. With the exception of battery, ran and program, ran. It's helpful to remember at this point that we pull the relevant rows out of the transition matrix by multiplying it with a vector showing which features are currently active. For this example so far, we've been using the implied feature vector shown here. The following diagram from Brandon Rohrer's Transformers From Scratch shows a feature selection vector.

Back to Top

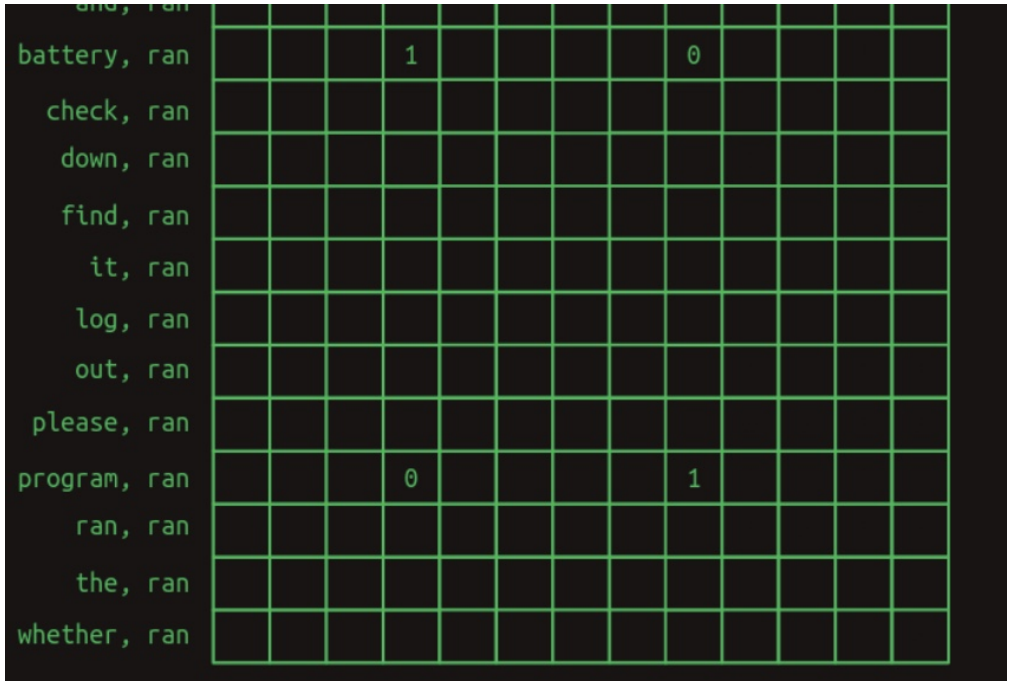
- It includes a one for each feature that is a combination of ran with each of the words that come before it. Any words that come after it don't get included in the feature set. (In the next word prediction problem these haven't been seen yet, and so it's not fair to use them predict what comes next.) And this doesn't include all the other possible word combinations. We can safely ignore these for this example because they will all be zero.
- To improve our results, we can additionally force the unhelpful features to zero by creating a **mask**. It's a vector full of ones except for the positions you'd like to hide or mask, and those are set to zero. In our case we'd like to mask everything except for `battery`, `ran` and `program`, `ran`, the only two features that have been of any help. The following diagram from Brandon Rohrer's Transformers From Scratch shows a masked feature vector.

Back to Top



- To apply the mask, we multiply the two vectors element by element. Any feature activity value in an unmasked position will be multiplied by one and left unchanged. Any feature activity value in a masked position will be multiplied by zero, and thus forced to zero.
- The mask has the effect of hiding a lot of the transition matrix. It hides the combination of ran with everything except `battery` and `program`, leaving just the features that matter. The following diagram from Brandon Rohrer's Transformers From Scratch shows a masked transition matrix.

Back to Top



- After masking the unhelpful features, the next word predictions become much stronger. When the word battery occurs earlier in the sentence, the word after ran is predicted to be down with a weight of 1 and please with a weight of 0. What was a weight difference of 25 percent has become a difference of infinity percent. There is no doubt what word comes next. The same strong prediction occurs for please when program occurs early on.
- This process of selective masking is the attention called out in the title of the original paper on transformers. So far, what we've described is a just an approximation of how attention is implemented in the paper.

Back to Top

> *information retrieval, an attention function is the mapping between a query and a set of key-value pairs to an output. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function (referred to as the "alignment" function in Bengio's original paper that introduced attention) of the query with the corresponding key. While this captures a top-level overview of the important concepts, the details are discussed in the section on Attention.*

From Feature Vectors to Transformers

- The selective-second-order-with-skips model is a useful way to think about what transformers do, at least in the decoder side. It captures, to a first approximation, what generative language models like OpenAI's GPT-3 are doing. It doesn't tell the complete story, but it represents the central gist of it.
- The next sections cover more of the gap between this intuitive explanation and how transformers are implemented. These are largely driven by three practical considerations:
    1. **Computers are especially good at matrix multiplications.** There is an entire industry around building computer hardware specifically for fast matrix multiplications, with CPUs being good at matrix multiplications owing to it being modeled as a multi-threaded algorithm, GPUs being even faster at it owing to them have massively parallelizable/multi-threaded dedicated cores on-chip that are especially suited. Any computation that can be expressed as a matrix multiplication can be made shockingly efficient. It's a bullet train. If you can get your baggage into it, it will get you where you want to go real fast.

Back to Top

2.
3. **Each step needs to be differentiable.** So far we've just been working with toy examples, and have had the luxury of hand-picking all the transition probabilities and mask values—the model **parameters**. In practice, these have to be learned via **backpropagation**, which depends on each computation step being differentiable. This means that for any small change in a parameter, we can calculate the corresponding change in the model error or **loss**.
4. **The gradient needs to be smooth and well conditioned.** The combination of all the derivatives for all the parameters is the loss **gradient**. In practice, getting backpropagation to behave well requires gradients that are smooth, that is, the slope doesn't change very quickly as you make small steps in any direction. They also behave much better when the gradient is well conditioned, that is, it's not radically larger in one direction than another. If you picture a loss function as a landscape, The Grand Canyon would be a poorly conditioned one. Depending on whether you are traveling a

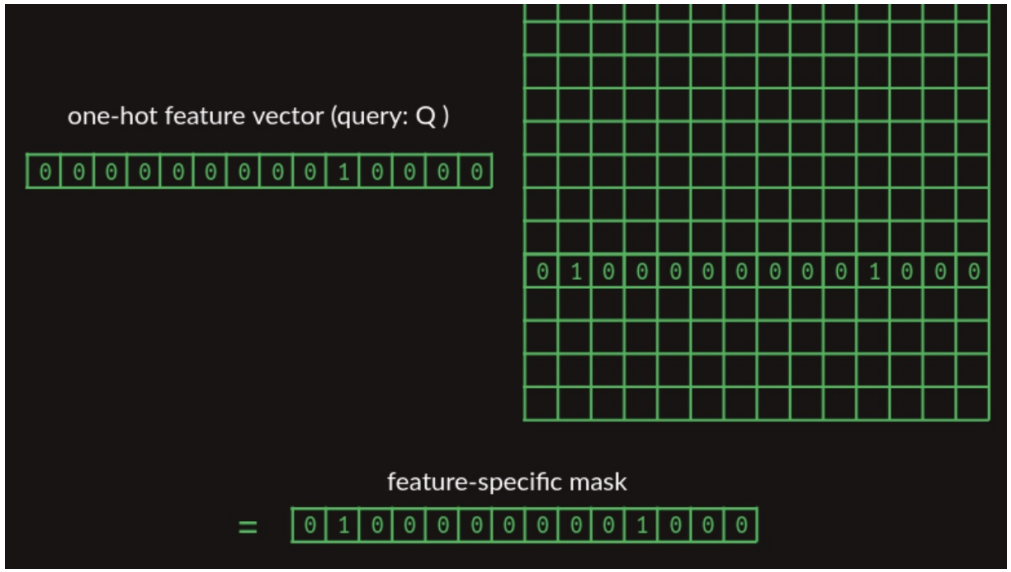
Back to Top

magnitude in every direction.

Attention As Matrix Multiplication

- Feature weights could be straightforward to build by counting how often each word pair/next word transition occurs in training, but attention masks are not. Up to this point, we've pulled the mask vector out of thin air. How transformers find the relevant mask matters. It would be natural to use some sort of lookup table, but now we are focusing hard on expressing everything as matrix multiplications.
- We can use the same lookup method we introduced above by stacking the mask vectors for every word into a matrix and using the one-hot representation of the most recent word to pull out the relevant mask.

Back to Top



- In the matrix showing the collection of mask vectors, we've only shown the one we're trying to pull out, for clarity.
- We're finally getting to the point where we can start tying into the paper. This mask lookup is represented by the QK^{T} term in the attention equation (below), described in the details are discussed in the section on Single Head Attention Revisited.

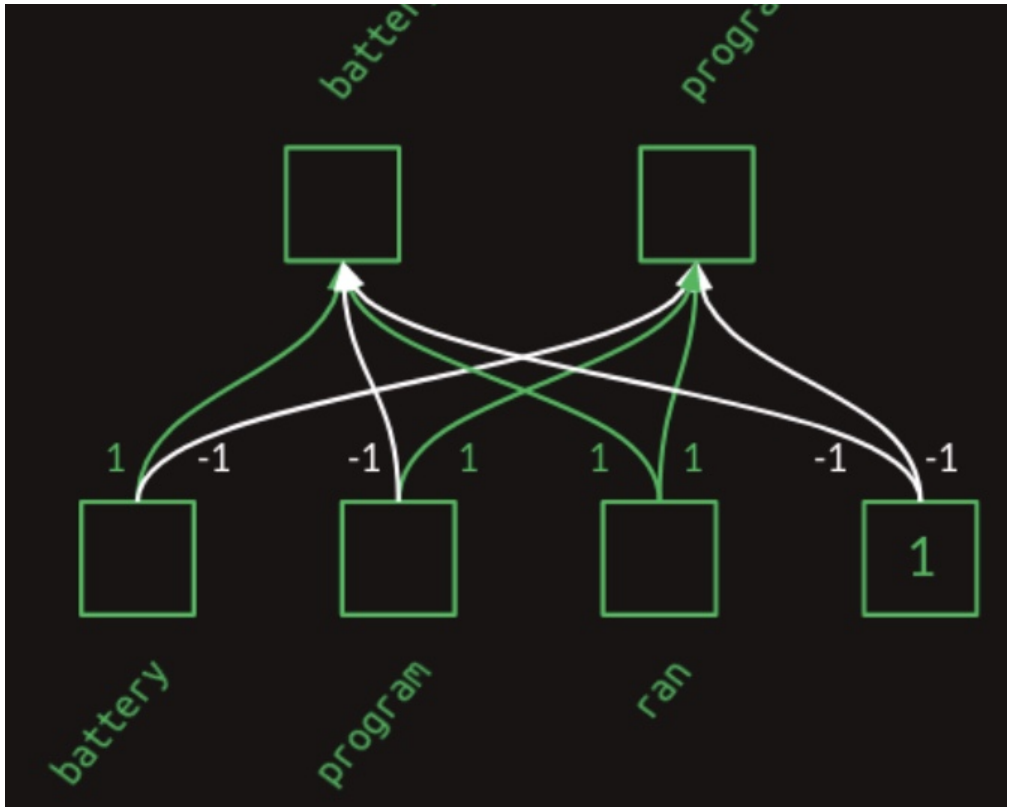
$$\text{Attention}(\mathrm{Q}, \mathrm{K}, \mathrm{V}) = \text{softmax}\left(\frac{\mathrm{QK}^{\mathrm{T}}}{\sqrt{\mathrm{d_k}}}\right)\mathrm{V}$$

- The query Q represents the feature of interest and the matrix K represents the collection of masks. Because it's stored with masks in columns, rather than rows, it needs to be transposed (with the T operator) before multiplying. By the time we're all done, we'll make some important modifications to this, but at this level it captures the concept of a differentiable lookup table that transformers make use of.

Back to Top

- Another step that we have been hand wavy about so far is the construction of transition matrices. We have been clear about the logic, but not about how to do it with matrix multiplications.
- Once we have the result of our attention step, a vector that includes the most recent word and a small collection of the words that have preceded it, we need to translate that into features, each of which is a word pair. Attention masking gets us the raw material that we need, but it doesn't build those word pair features. To do that, we can use a single layer fully connected neural network.
- To see how a neural network layer can create these pairs, we'll hand craft one. It will be artificially clean and stylized, and its weights will bear no resemblance to the weights in practice, but it will demonstrate how the neural network has the expressivity necessary to build these two word pair features. To keep it small and clean, will focus on just the three attended words from this example, `battery`, `program`, `ran`. The following diagram from Brandon Rohrer's Transformers From Scratch shows a neural network layer for creating multi-word features.

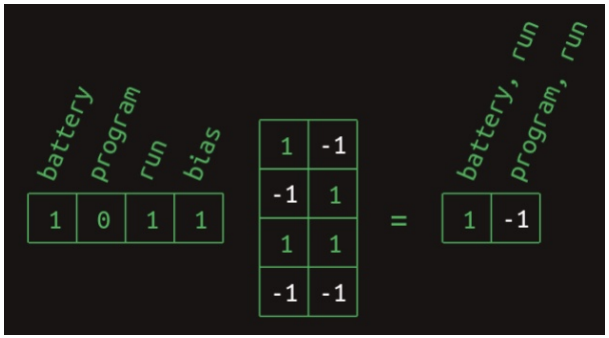
Back to Top



- In the layer diagram above, we can see how the weights act to combine the presence and absence of each word into a collection of features. This can also be expressed in matrix form. The following diagram from Brandon Rohrer's Transformers From Scratch shows a weight matrix for creating multi word features.

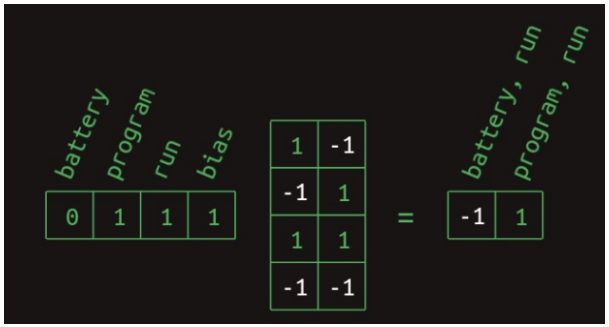
Back to Top

- And it can be calculated by a matrix multiplication with a vector representing the collection of words seen so far. The following diagram from Brandon Rohrer's Transformers From Scratch shows the calculation of the `battery, ran` feature.



- The battery and ran elements are 1 and the program element is 0. The bias element is always 1, a fea of neural networks. Working through the matrix multiplication gives a 1 for the element represen

Back to Top



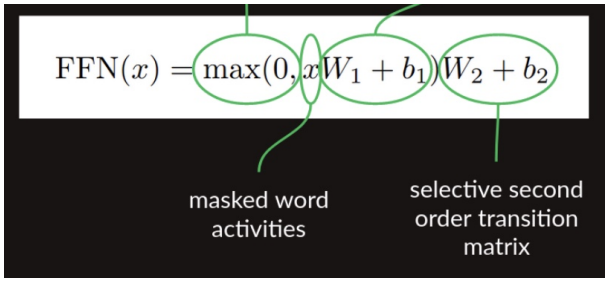
- The final step in calculating these word combo features is to apply a rectified linear unit (ReLU) nonlinearity. The effect of this is to substitute any negative value with a zero. This cleans up both of these results so they represent the presence (with a 1) or absence (with a 0) of each word combination feature.
- With those gymnastics behind us, we finally have a matrix multiplication based method for creating multiword features. Although I originally claimed that these consist of the most recent word and one earlier word, a closer look at this method shows that it can build other features too. When the feature creation matrix is learned, rather than hard coded, other structures can be learned. Even in this toy example, there's nothing to stop the creation of a three-word combination like `battery, program, ran`. If this combination occurred commonly enough it would probably end up being represented. There wouldn't be any way to indicated what order the words occurred in (at least not yet), but we could absolutely use their co-occurrence to make predictions. It would even be possible to make use of word combos that ignored the most recent word, like `battery, program`. These and other types of features are probably created in practice, exposing the over-simiplification we made when we claimed that transformers are a selective-second-order-with-skips sequence model. There's more nuance to it than that, and now you can see exactly what that nuance is. This won't be the last time we'll change the story to incorporate more subtlety.
- In this form, the multiword feature matrix is ready for one more matrix multiplication, the second order sequence model with skips we developed above. All together, the following sequence of feedforw

Back to Top

4.

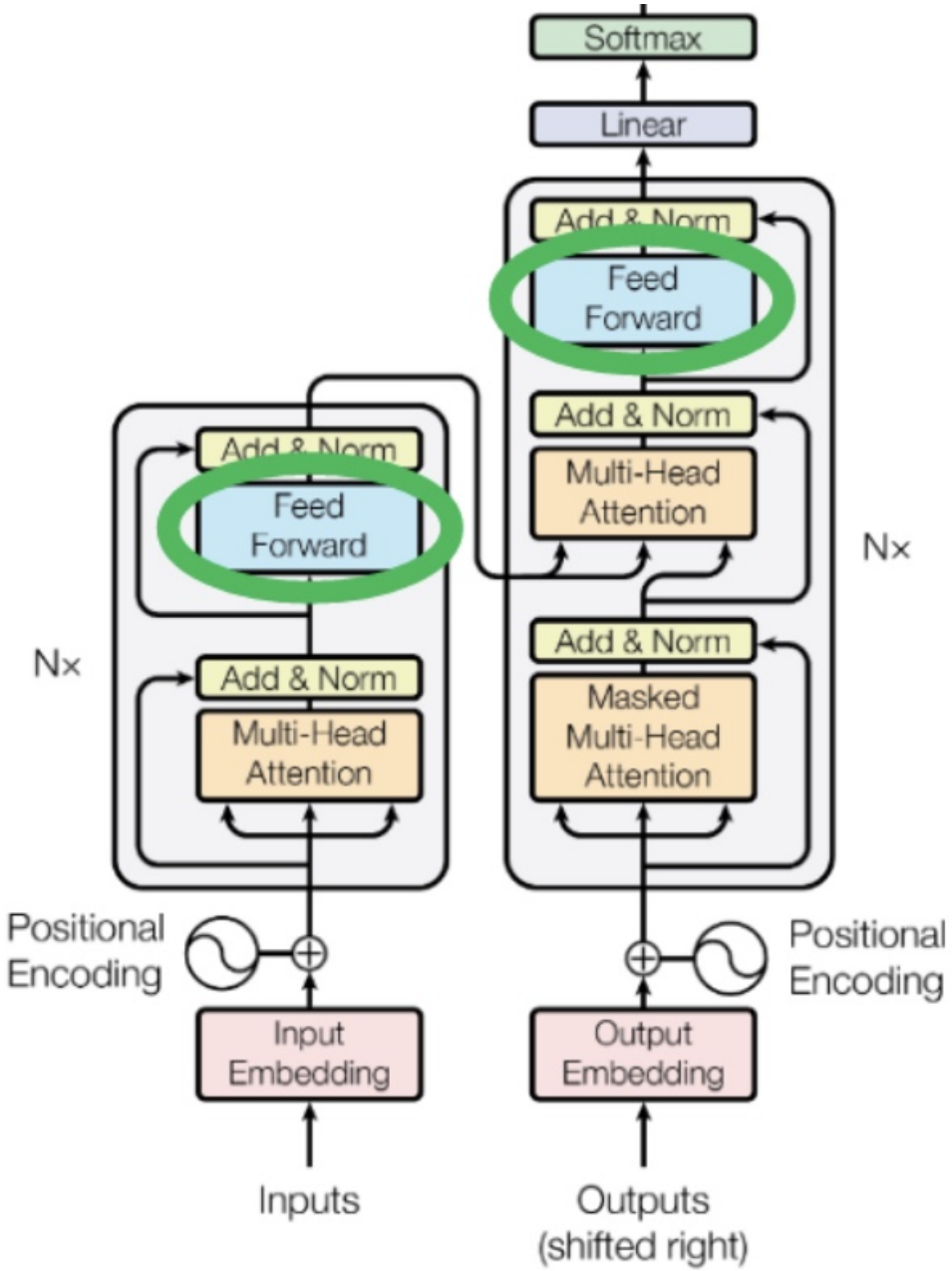
- The following equation from the paper shows the steps behind the Feed Forward block in a concise mathematical formulation.

Back to Top



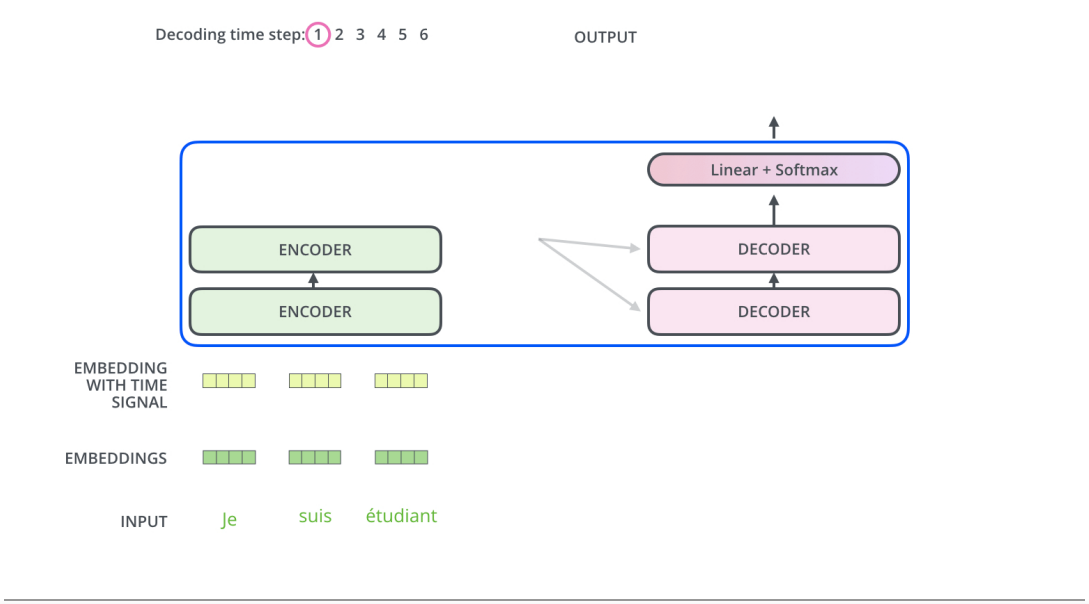
- The architecture diagram (below) from the Transformers paper shows these lumped together as the Feed Forward block.

Back to Top


Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

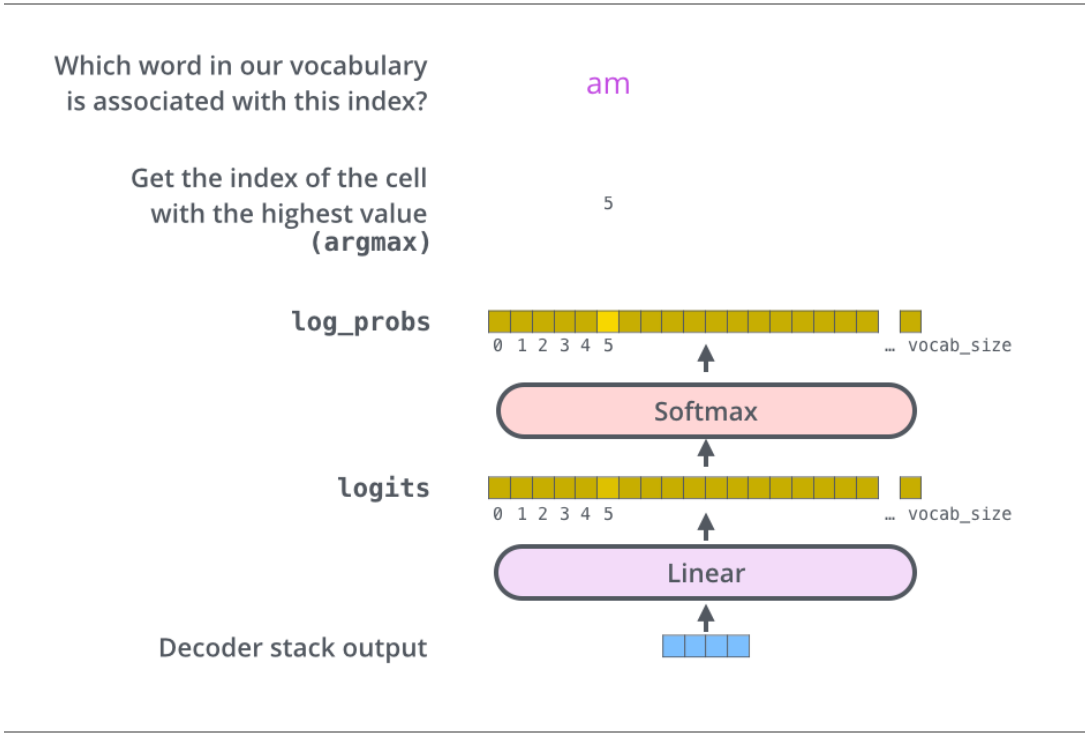
Back to Top

- So far we've only talked about next word prediction. There are a couple of pieces we need to add to get our decoder to generate a long sequence. The first is a **prompt**, some example text to give the transformer running start and context on which to build the rest of the sequence. It gets fed in to decoder, the column on the right in the image above, where it's labeled "Outputs (shifted right)". Choosing a prompt that gives interesting sequences is an art in itself, called prompt engineering. It's also a great example of humans modifying their behavior to support algorithms, rather than the other way around.
- The decoder is fed a `<START>` token to generate the first word. The token serves as a signal to tell itself to start decoding using the compact representation of collected information from the encoder (more on this in the section on Cross-Attention). The following animation from Jay Alammar's The Illustrated Transformer shows: (i) the process of parallel ingestion of tokens by the encoder (leading to the generation of key and value matrices from the last encoder layer), and (ii) the decoder producing the first token (the `<START>` token is missing from the animation).



Back to Top

output of the linear layer) to a probability distribution (at the output of the softmax layer) and finally, to an output word as shown below in the below illustration from Jay Alammar's The Illustrated Transformer.



Role of the Final Linear and Softmax Layer

- The linear layer is a simple fully connected layer that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.
- A typical NLP model knows about 40,000 unique English words (our model's "output vocabulary") that it's learned from its training dataset. This would make the logits vector 40,000 dimensional, with each cell

Back to Top

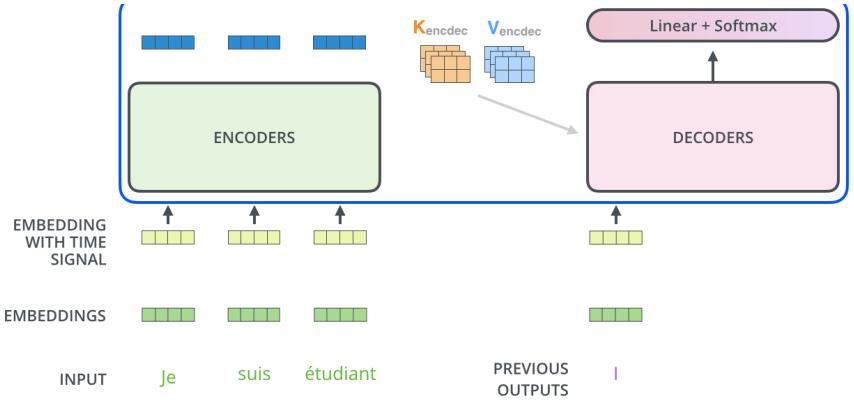
negative, i.e, $\in [0, 1]$, and (ii) all values add up to 1.0.

- At each position, the distribution shows the predicted probabilities for each next word in the vocabulary. We don't care about predicted probabilities for existing words in the sequence (since they're already established) – what we really care about are the predicted probabilities for the next word after the end of the prompt.
- The cell with the highest probability is chosen (more on this in the section on Greedy Decoding), and the word associated with it is produced as the output for this time step.

Greedy Decoding

- There are several ways to go about choosing what that word should be, but the most straightforward is called **greedy decoding**, which involves picking the word with the highest probability.
- The new next word then gets added to the sequence and fed in as input to the decoder, and the process is repeated until you either reach an `<EOS>` token or once you sample a fixed number of tokens. The following animation from Jay Alammar's The Illustrated Transformer shows the decoder auto-regressively generating the next token (by absorbing the previous tokens).

Back to Top



- The one piece we're not quite ready to describe in detail is yet another form of masking, ensuring that when the transformer makes predictions it only looks behind, not ahead. It's applied in the block labeled "Masked Multi-Head Attention". We'll revisit this later in the section on Single Head Attention Revisited to understand how it is implemented.

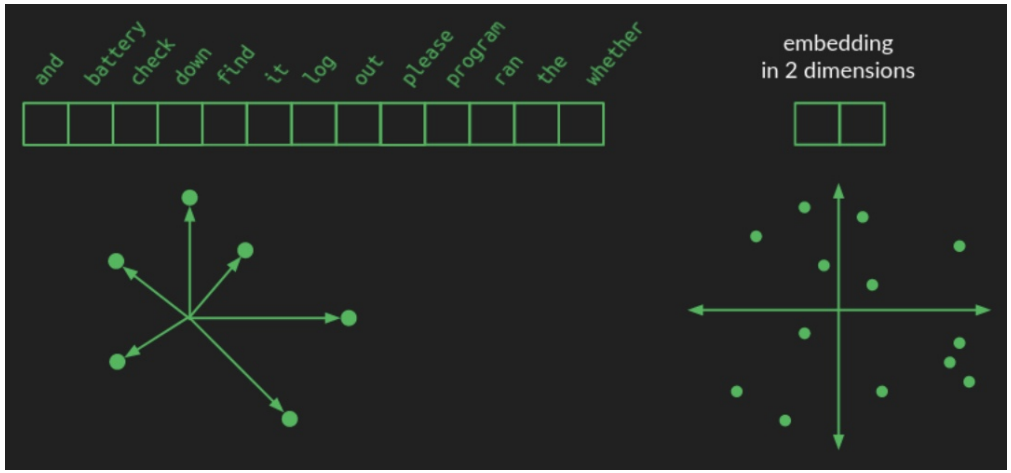
Transformer Core

Embeddings

- As we've described them so far, transformers are too big. For a vocabulary size N of say 50,000, the transition matrix between all pairs of words and all potential next words would have 50,000 columns and 50,000 squared (2.5 billion) rows, totaling over 100 trillion elements. That is still a stretch, even for modern hardware.
- It's not just the size of the matrices that's the problem. In order to build a stable transition language model we would have to provide training data illustrating every potential sequence several times at least.

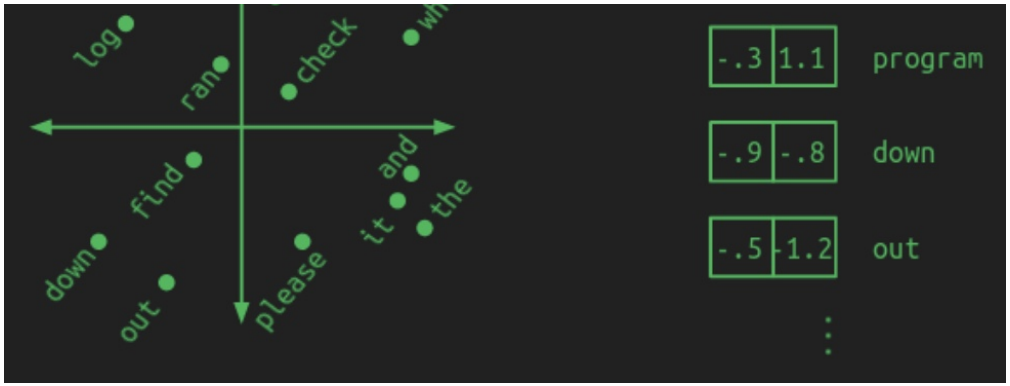
Back to Top

from the origin along one of the many axes. A crude representation of a high dimensional space is as below.

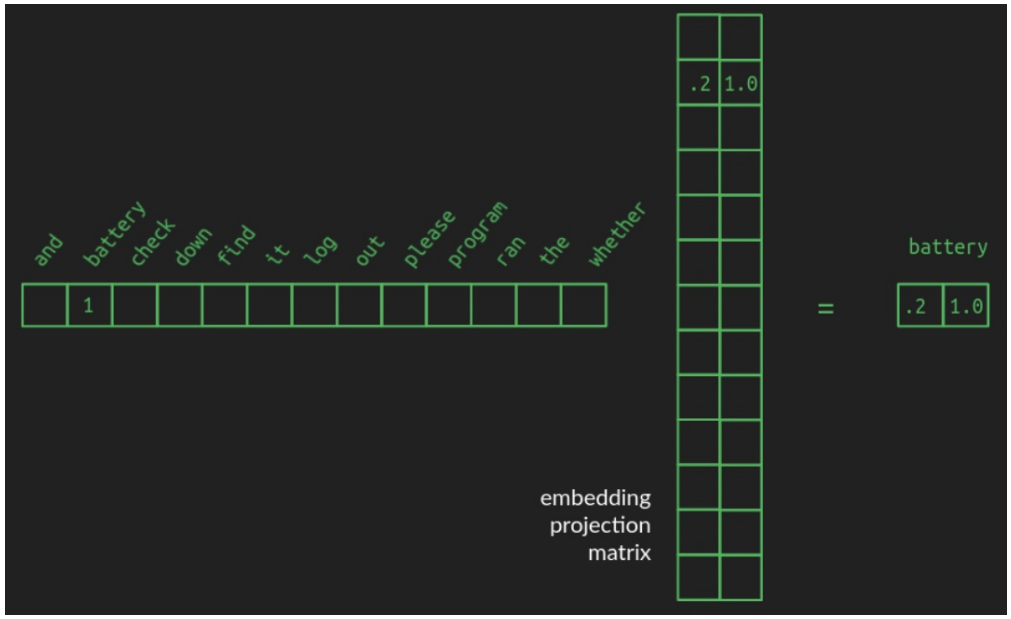


- In an embedding, those word points are all taken and rearranged into a lower-dimensional space. In linear algebra terminology, this refers to the **projecting** data points. The picture above shows what they might look like in a 2-dimensional space for example. Now, instead of needing N numbers to specify a word, we only need 2. These are the (x, y) coordinates of each point in the new space. Here's what a 2-dimensional embedding might look like for our toy example, together with the coordinates of a few of the words.

Back to Top

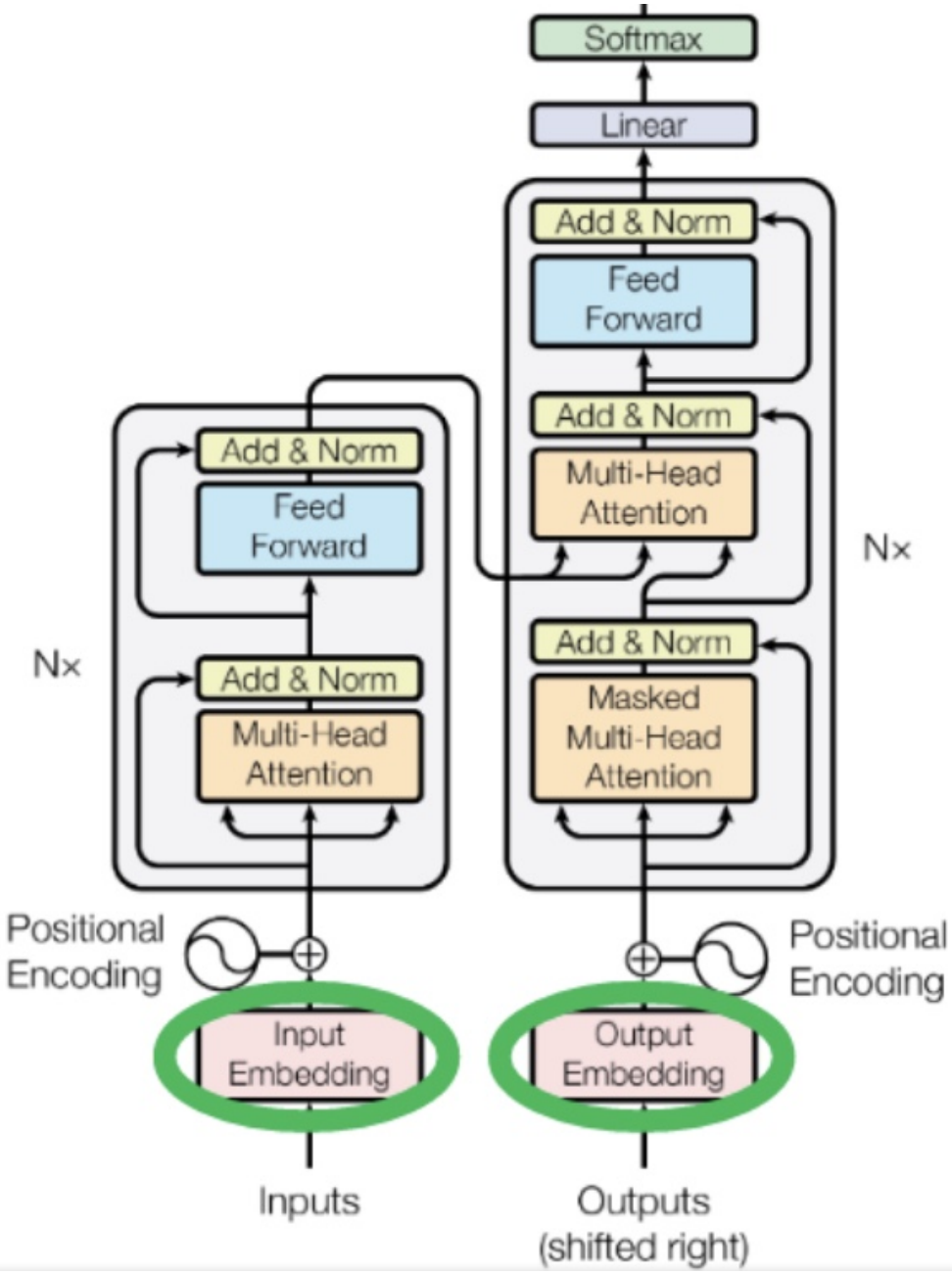


- A good embedding groups words with similar meanings together. A model that works with an embedding learns patterns in the embedded space. That means that whatever it learns to do with one word automatically gets applied to all the words right next to it. This has the added benefit of reducing the amount of training data needed. Each example gives a little bit of learning that gets applied across a whole neighborhood of words.
- The illustration shows that by putting important components in one area (`battery`, `log`, `program`), prepositions in another (`down`, `out`), and verbs near the center (`check`, `find`, `ran`). In an actual embedding the groupings may not be so clear or intuitive, but the underlying concept is the same. Distance is small between words that behave similarly.
- An embedding reduces the number of parameters needed by a tremendous amount. However, the fewer the dimensions in the embedded space, the more information about the original words gets discarded. The richness of a language still requires quite a bit of space to lay out all the important concepts so that they don't step on each other's toes. By choosing the size of the embedded space, we get to trade off computational load for model accuracy.
- It will probably not surprise you to learn that projecting words from their one-hot representation to an embedded space involves a matrix multiplication. Projection is what matrices do best. Starting with a one hot matrix that has one row and N columns, and moving to an embedded space of two dimensions,



- This example shows how a one-hot vector, representing for example battery, pulls out the row associated with it, which contains the coordinates of the word in the embedded space. In order to make the relationship clearer, the zeros in the one-hot vector are hidden, as are all the other rows that don't get pulled out of the projection matrix. The full projection matrix is dense, each row containing the coordinates of the word it's associated with.
- Projection matrices can convert the original collection of one-hot vocabulary vectors into any configuration in a space of whatever dimensionality you want. The biggest trick is finding a useful projection, one that has similar words grouped together, and one that has enough dimensions to spread them out. There are some decent pre-computed embeddings for common langauges, like English. Also, like everything else in the transformer, it can be learned during training.
- The architecture diagram from the Transformers paper shows where the embeddings are generated:

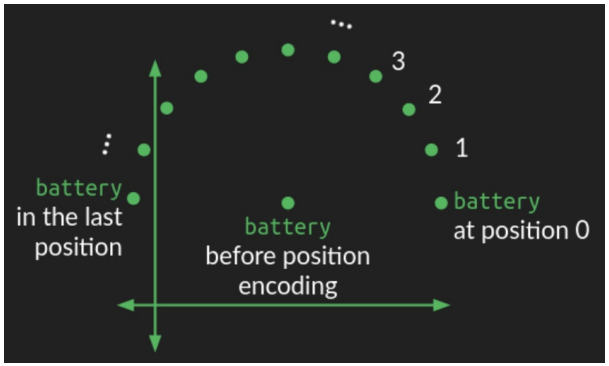
Back to Top


Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Back to Top

before the very most recent word. Now we get to fix that using positional embeddings, which offer a gateway to embed spatial information as an input to the transformer.

- There are several ways that position information could be introduced into our embedded representation of words, but the way it was done in the original transformer was to add a circular wiggle by using sinusoidal positional embeddings. Newer positional encoding schemes that utilize sophisticated schemes such as Rotary Position Embeddings, which encode absolute positional information with a rotation matrix and naturally incorporate explicit relative position dependency in the self-attention formulation, have been recently proposed.
- The following diagram from Brandon Rohrer's Transformers From Scratch shows that positional encoding introduces a circular wiggle, owing to the addition of sinusoidal positional embeddings:



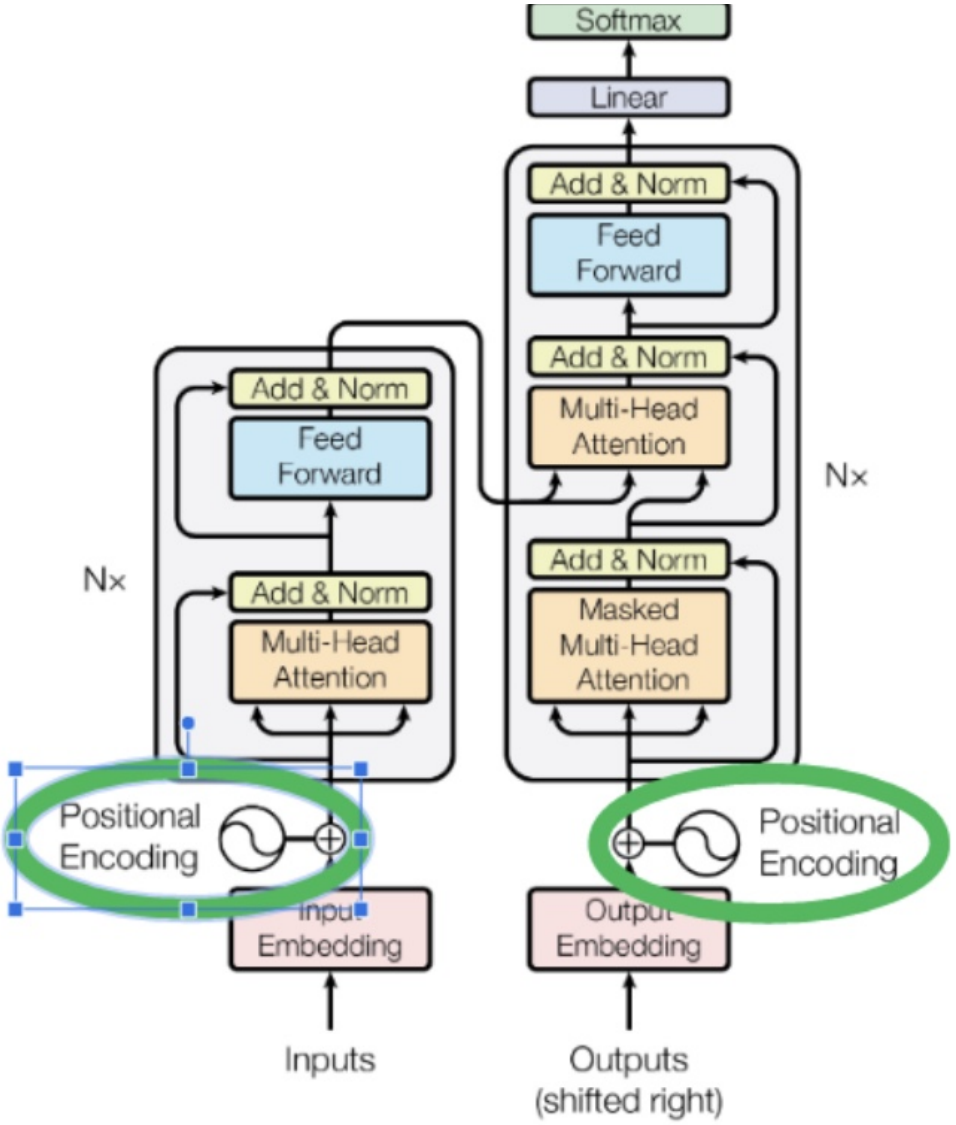
- The position of the word in the embedding space acts as the center of a circle. A perturbation is added to it, depending on where it falls in the order of the sequence of words. For each position, the word is moved the same distance but at a different angle, resulting in a circular pattern as you move through the sequence. Words that are close to each other in the sequence have similar perturbations, but words that are far apart are perturbed in different directions.

Back to Top

dimension pairs, the wiggle will sweep out many rotations of the circle. In other pairs, it will only sweep out a small fraction of a rotation. The combination of all these circular wiggles of different frequencies gives a good representation of the absolute position of a word within the sequence.

- In the architecture diagram from the Transformers paper, these blocks show the generation of the position encoding and its addition to the embedded words:

Back to Top

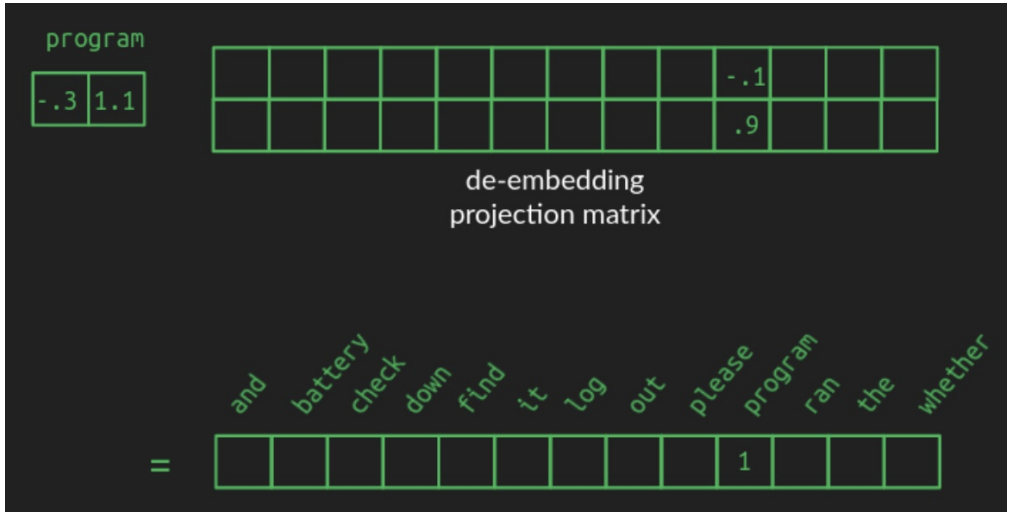


Back to Top

Why Sinusoidal Positional Embeddings Work

Decoding Output Words / De-embeddings

- Embedding words makes them vastly more efficient to work with, but once the party is over, they need to be converted back to words from the original vocabulary. De-embedding is done the same way embeddings are done, with a projection from one space to another, that is, a matrix multiplication.
- The de-embedding matrix is the same shape as the embedding matrix, but with the number of rows and columns flipped. The number of rows is the dimensionality of the space we're converting from. In the example we've been using, it's the size of our embedding space, two. The number of columns is the dimensionality of the space we're converting to — the size of the one-hot representation of the full vocabulary, 13 in our example. The following diagram shows the de-embedding transform:



- The values in a good de-embedding matrix aren't as straightforward to illustrate as those from the embedding matrix, but the effect is similar. When an embedded vector representing, say, the w

Back to Top

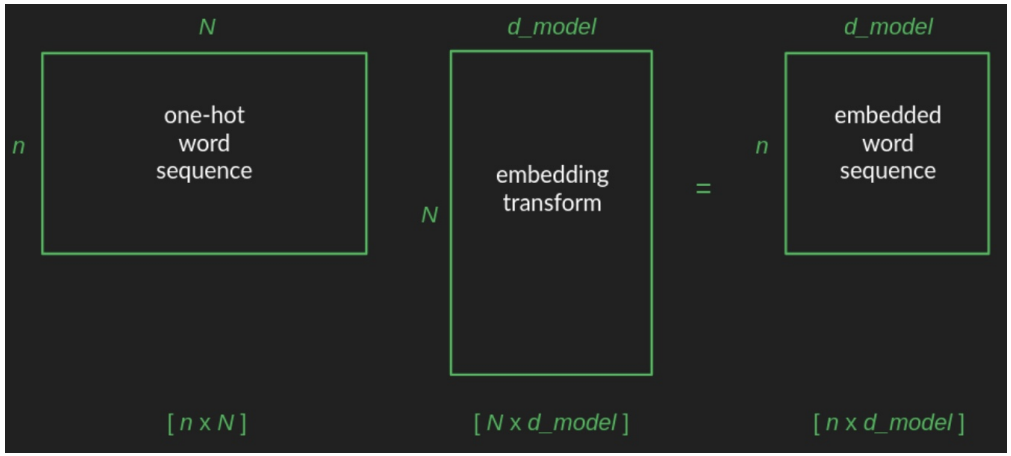
negative values. The output vector in vocabulary space will no longer be one-hot or sparse. It will be dense, with nearly all values non-zero. The following diagram shows the representative dense result vector from de-embedding:

- We can recreate the one-hot vector by choosing the word associated with the highest value. This operation is also called **argmax**, the argument (element) that gives the maximum value. This is how to do greedy sequence completion, as mentioned in the section on sampling a sequence of output words. It's a great first pass, but we can do better.
- If an embedding maps very well to several words, we might not want to choose the best one every time. It might be only a tiny bit better choice than the others, and adding a touch of variety can make the result more interesting. Also, sometimes it's useful to look several words ahead and consider all the directions the sentence might go before settling on a final choice. In order to do these, we have to first convert our de-embedding results to a probability distribution.

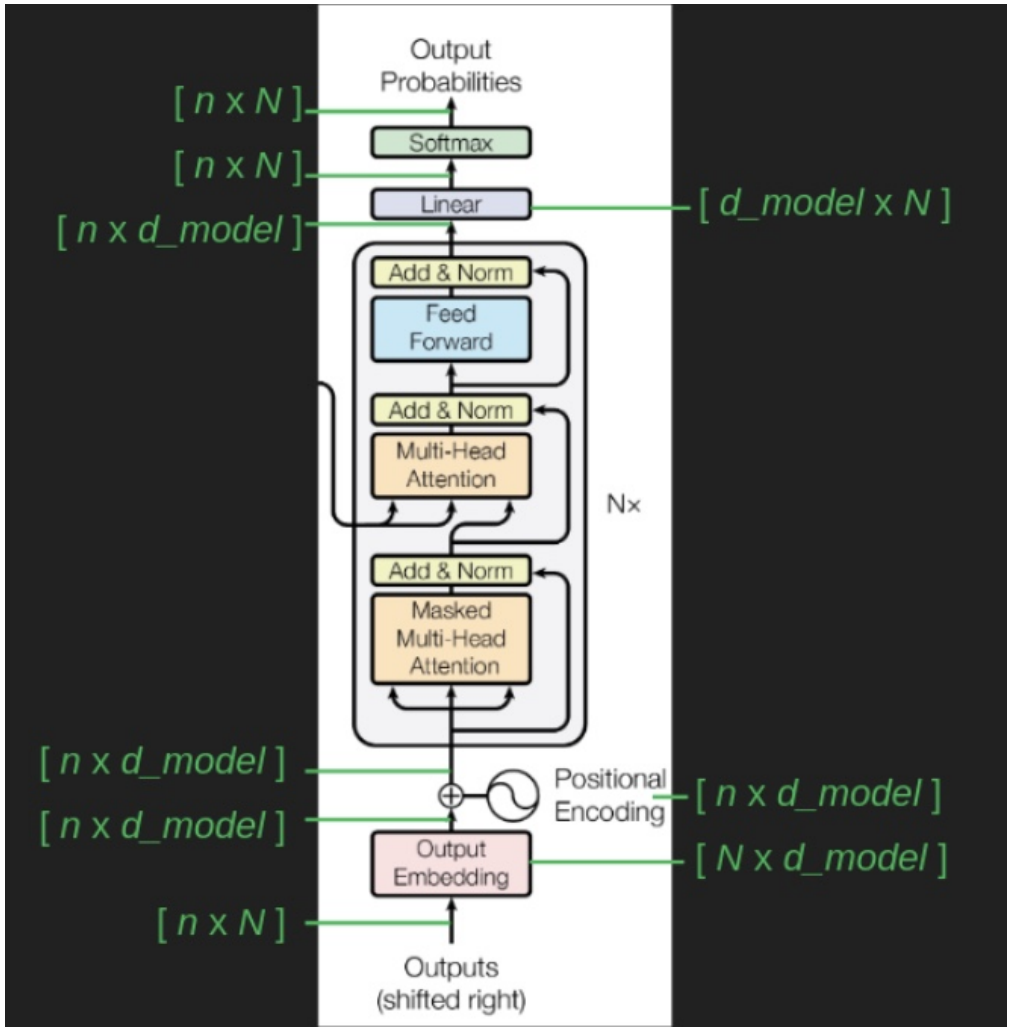
Attention

- Now that we've made peace with the concepts of projections (matrix multiplications) and spaces (vector sizes), we can revisit the core attention mechanism with renewed vigor. It will help clarify the algorith

don't specify.) 2048 in GPT-3.
    - $\mathbf{d_{model}}$: number of dimensions in the embedding space used throughout the model (512 in the paper).
- The original input matrix is constructed by getting each of the words from the sentence in their one-hot representation, and stacking them such that each of the one-hot vectors is its own row. The resulting input matrix has $\mathbf{n}$ rows and $\mathbf{N}$ columns, which we can abbreviate as $\mathbf{[n \times N]}$.



- As we illustrated before, the embedding matrix has $\mathbf{N}$ rows and $\mathbf{d_{model}}$ columns, which we can abbreviate as $\mathbf{[N \times d_{model}]}$. When multiplying two matrices, the result takes its number of rows from the first matrix, and its number of columns from the second. That gives the embedded word sequence matrix a shape of $\mathbf{[n \times d_{model}]}$.
- We can follow the changes in matrix shape through the transformer as a way to track what's going on (c.f. figure below; source). After the initial embedding, the positional encoding is additive, rather than a multiplication, so it doesn't change the shape of things. Then the embedded word sequence goes into attention layers, and comes out the other end in the same shape. (We'll come back to the inner workings



Why Attention? Contextualized Word Embeddings

Back to Top

History

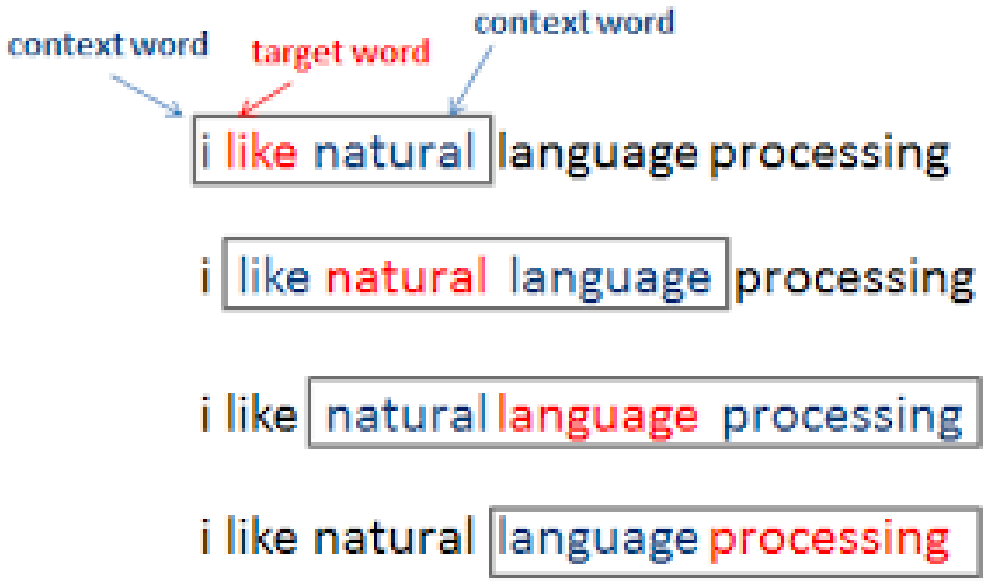
	1 This	2 movie	3 is	4 very	5 scary	6 and	7 long	8 not	9 slow	10 spooky	11 good	Length of the review(in words)
Review 1	1	1	1	1	1	1	1	0	0	0	0	7
Review 2	1	1	2	0	0	1	1	0	1	0	0	8
Review 3	1	1	1	0	0	0	1	0	0	1	1	6

- However, this suggests that when all words are considered equally important, significant words like “crisis” which carry important meaning in the text can be drowned out by insignificant words like “and”, “for”, or “the” which add little information but are commonly used in all types of text.
- To address this issue, **TF-IDF (Term Frequency-Inverse Document Frequency)** assigns weights to each word based on its frequency across all documents. The more frequent the word is across all documents, the less weight it carries.
- However, this method is limited in that it treats each word independently and does not account for the fact that the meaning of a word is highly dependent on its context. As a result, it can be difficult to accurately capture the meaning of the text. This limitation was addressed with the use of deep learning techniques.

Enter Word2Vec: Neural Word Embeddings

- Word2Vec revolutionized embeddings by using a neural network to transform texts into vectors.
- Two popular approaches are the Continuous Bag of Words (CBOW) and Skip-gram models, which are trained using raw text data in an unsupervised manner. These models learn to predict the center word given context words or the context words given the center word, respectively. The resulting trained weights encode the meaning of each word relative to its context.

Back to Top



- However, Word2Vec and similar techniques (such as GloVe, FastText, etc.) have their own limitations. After training, each word is assigned a unique embedding. Thus, polysemous words (i.e, words with multiple distinct meanings in different contexts) cannot be accurately encoded using this method. As an example:

> *"The man was accused of robbing a **bank**." "The man went fishing by the **bank** of the river."*

- As another example:

> *"Time **flies** like an arrow." "Fruit **flies** like a banana."*

- This limitation gave rise to contextualized word embeddings.

Back to Top

- Note that while Transformer-based architectures (e.g., BERT) learn contextualized word embeddings, prior work (ELMo) originally proposed this concept.
- As indicated in the prior section, contextualized word embeddings help distinguish between multiple meanings of the same word, in case of polysemous words.
- The process begins by encoding each word as an embedding (i.e., a vector that represents the word and that LLMs can operate with). A basic one is one-hot encoding, but we typically use embeddings that encode meaning (the Transformer architecture begins with a randomly-initialized `nn.Embedding` instance that is learnt during the course of training). However, note that the embeddings at this stage are non-contextual, i.e., they are fixed per word and do not incorporate context surrounding the word.
- As we will see in the section on Single Head Attention Revisited, self-attention transforms the embedding to a weighted combination of the embeddings of all the other words in the text. This represents the contextualized embedding that packs in the context surrounding the word.
- Considering the example of the word **bank** above, the embedding for **bank** in the first sentence would have contributions (and would thus be influenced significantly) from words like “accused”, “robbing”, etc. while the one in the second sentence would utilize the embeddings for “fishing”, “river”, etc. In case of the word **flies**, the embedding for **flies** in the first sentence will have contributions from words like “go”, “soars”, “pass”, “fast”, etc. while the one in the second sentence would depend on contributions from “insect”, “bug”, etc.
- The following figure (source) shows an example for the word **flies**, and computing the new embeddings involves a linear combination of the representations of the other words, with the weight being proportional to the relationship (say, similarity) of other words compared to the current word. In other words, the output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key (also called the “alignment” function in Bengio’s original paper that introduced attention in the context of neural networks).

Back to Top

Types of Attention: Additive, Multiplicative (Dot-product), and Scaled

- The Transformer is based on “Scaled Dot-Product Attention”.
- The two most commonly used attention functions are additive attention (Neural Machine Translation by Jointly Learning to Align and Translate), and dot-product (multiplicative) attention. Dot-product attention is identical to their algorithm, except for the scaling factor of $\frac{1}{\sqrt{\mathrm{d_k}}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.
- While for small values of $\mathrm{d_k}$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $\mathrm{d_k}$ (Massive Exploration of Neural Machine Translation Architectures). We suspect that for large values of $\mathrm{d_k}$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients (To illustrate why the dot products get large, assume that the components of q and k are independent random

Back to Top

Attention Calculation

- Let's develop an intuition about the architecture using the language of mathematical symbols and vectors.
- We update the hidden feature h of the i^{th} word in a sentence $\square$ from layer ℓ to layer $\ell + 1$ as follows:

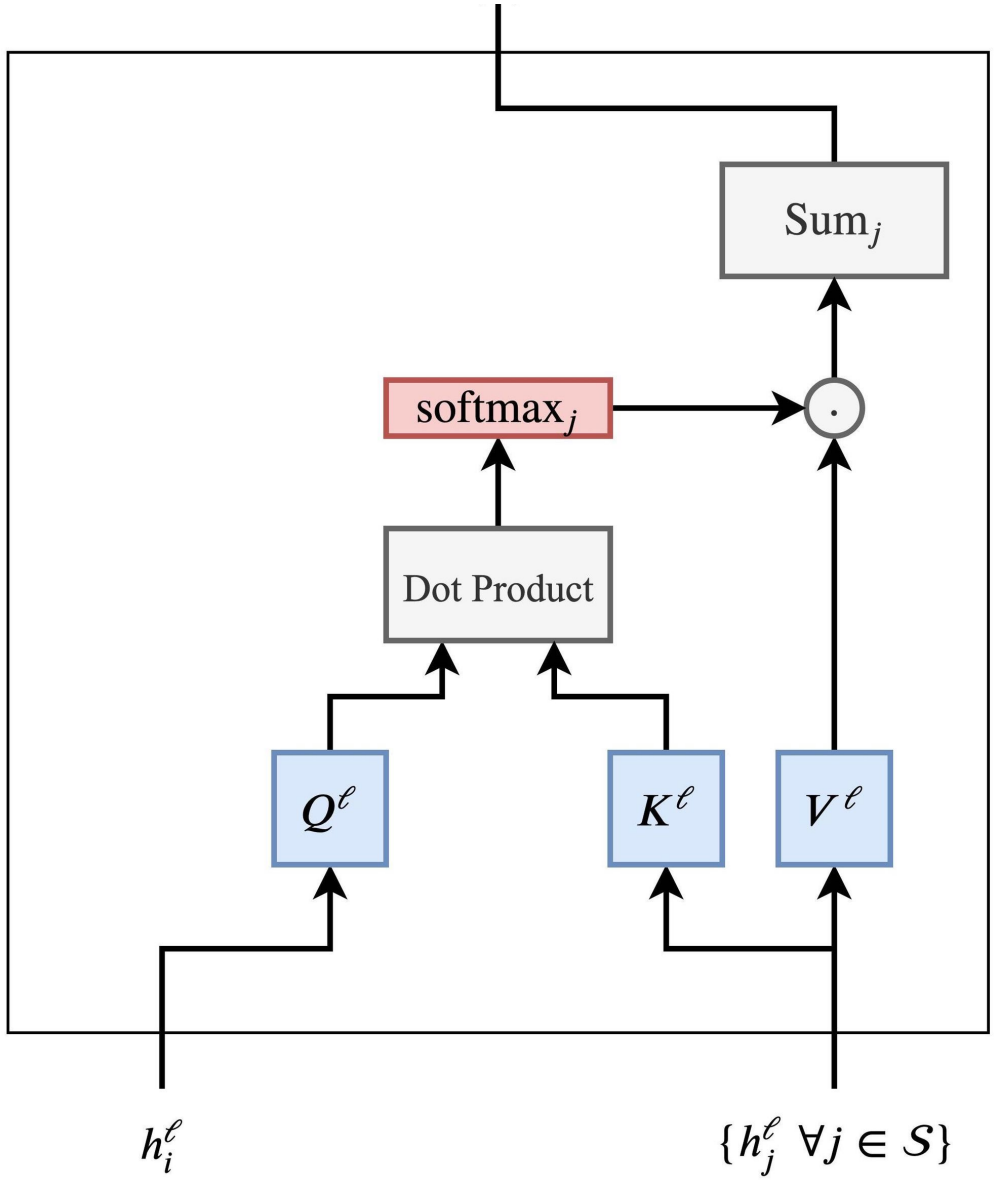
$$h_i^{\ell+1} = \text{Attention}\left(Q^\ell h_i^\ell, K^\ell h_j^\ell, V^\ell h_j^\ell\right)$$

  - i.e.,

$$h_i^{\ell+1} = \Sigma_{j \in \square} w_{ij} \left(V^\ell h_j^\ell\right)$$
$$\text{where } w_{ij} = \text{softmax}_j\left(Q^\ell h_i^\ell \cdot K^\ell h_j^\ell\right)$$

  - where $j \in \square$ denotes the set of words in the sentence and Q^ℓ, K^ℓ, V^ℓ are learnable linear weights (denoting the **Q**uery, **K**ey and **V**alue for the attention computation, respectively).
- Since the queries, keys, and values are all drawn from the same source, we refer to this attention as **self-attention** (we use "attention" and "self-attention" interchangeably in this topic). Self-attention forms the core component of Transformers. Also, given the use of the dot-product to ascertain similarity between the query and key vectors, the attention mechanism is also called **dot-product self-attention**.
- Note that one of the benefits of self-attention over recurrence is that it's highly parallelizable. In other words, the attention mechanism is performed in parallel for each word in the sentence to obtain their updated features in one shot. This is a **big advantage for Transformers** over RNNs, which update features word-by-word. In other words, Transformer-based deep learning models don't require sequential data to be processed in order, allowing for much more parallelization and reduced training time on GPUs than RNNs.
- We can understand the attention mechanism better through the following pipeline (source):

Back to Top


Sum$_j$
softmax$_j$
Dot Product
Q^ℓ
K^ℓ
V^ℓ
h_i^ℓ
$\{h_j^\ell \ \forall j \in S\}$

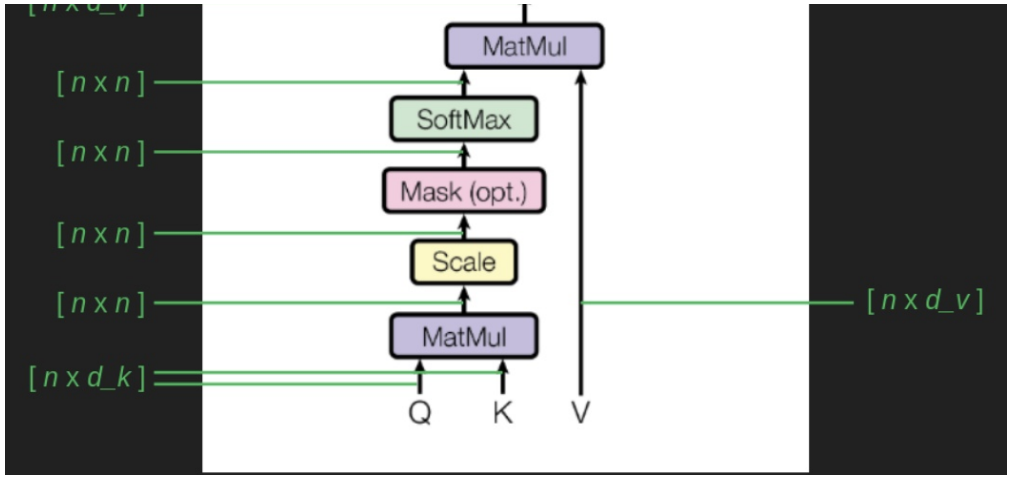
Back to Top

- Finally, we produce the updated word feature $\mathbf{x}_i'$ for word $\mathbf{i}$ by summing over all $\{\mathbf{x}_j'\}$'s weighted by their corresponding $\mathbf{w}_{ij}$. Each word in the sentence parallelly undergoes the same pipeline to update its features.

Single Head Attention Revisited

- We already walked through a conceptual illustration of attention in Attention As Matrix Multiplication above. The actual implementation is a little messier, but our earlier intuition is still helpful. The queries and the keys are no longer easy to inspect and interpret because they are all projected down onto their own idiosyncratic subspaces. In our conceptual illustration, one row in the queries matrix represents one point in the vocabulary space, which, thanks the one-hot representation, represents one and only one word. In their embedded form, one row in the queries matrix represents one point in the embedded space, which will be near a group of words with similar meanings and usage. The conceptual illustration mapped one query word to a set of keys, which in turn filtered out all the values that are not being attended to. Each attention head in the actual implempentation maps a query word to a point in yet another lower-dimensional embedded space. The result of this that that attention becomes a relationship between word groups, rather than between individual words. It takes advantage of semantic similarities (closeness in the embedded space) to generalize what it has learned about similar words.
- Following the shape of the matrices through the attention calculation helps to track what it's doing (source):

Back to Top



- The queries and keys matrices, $\mathbf{Q}$ and $\mathbf{K}$, both come in with shape $[\mathrm{n} \times \mathrm{d_k}]$. Thanks to $\mathbf{K}$ being transposed before multiplication, the result of $\mathbf{QK}^{\mathrm{T}}$ gives a matrix of $[\mathrm{n} \times \mathrm{d_k}] * [\mathrm{d_k} \times \mathrm{n}] = [\mathrm{n} \times \mathrm{n}]$. Dividing every element of this matrix by the square root of $\mathrm{d_k}$ has been shown to keep the magnitude of the values from growing wildly, and helps backpropagation to perform well. The softmax, as we mentioned, shoehorns the result into an approximation of an argmax, tending to focus attention one element of the sequence more than the rest. In this form, the $[\mathrm{n} \times \mathrm{n}]$ attention matrix roughly maps each element of the sequence to one other element of the sequence, indicating what it should be watching in order to get the most relevant context for predicting the next element. It is a filter that finally gets applied to the values matrix $\mathbf{V}$, leaving only a collection of the attended values. This has the effect of ignoring the vast majority of what came before in the sequence, and shines a spotlight on the one prior element that is most useful to be aware of.

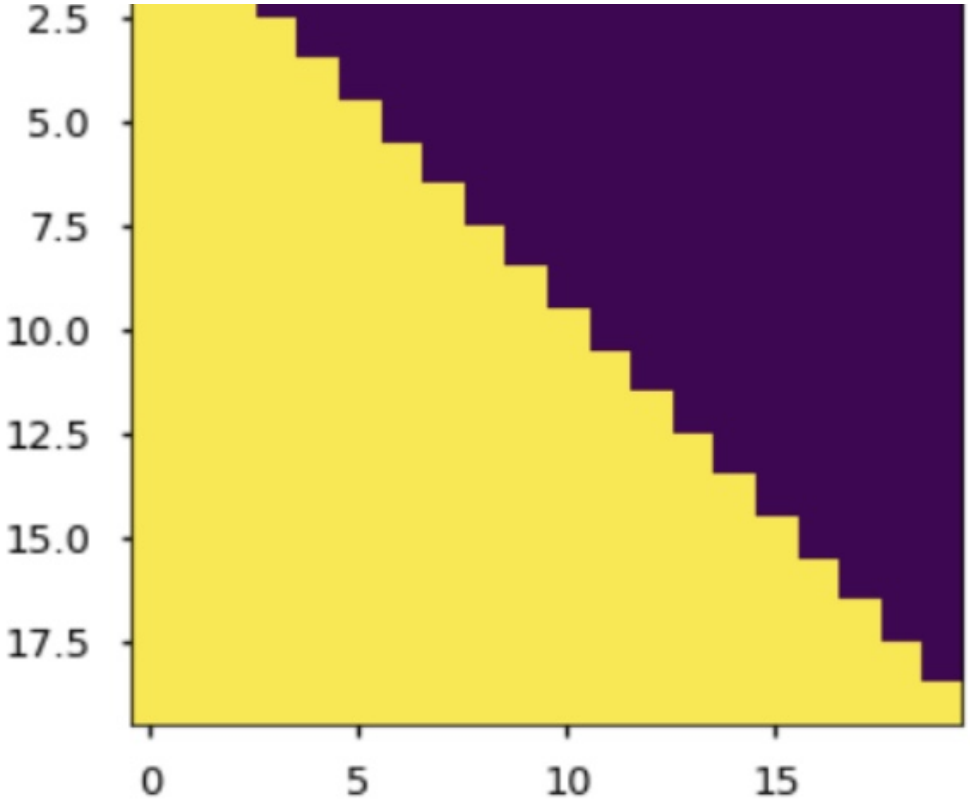
$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{QK}^{\mathrm{T}}}{\sqrt{\mathrm{d_k}}}\right)\mathbf{V}$$

Back to Top

> *corresponding key (also called the "alignment" function in Bengio's original paper that introduced attention in the context of neural networks).*

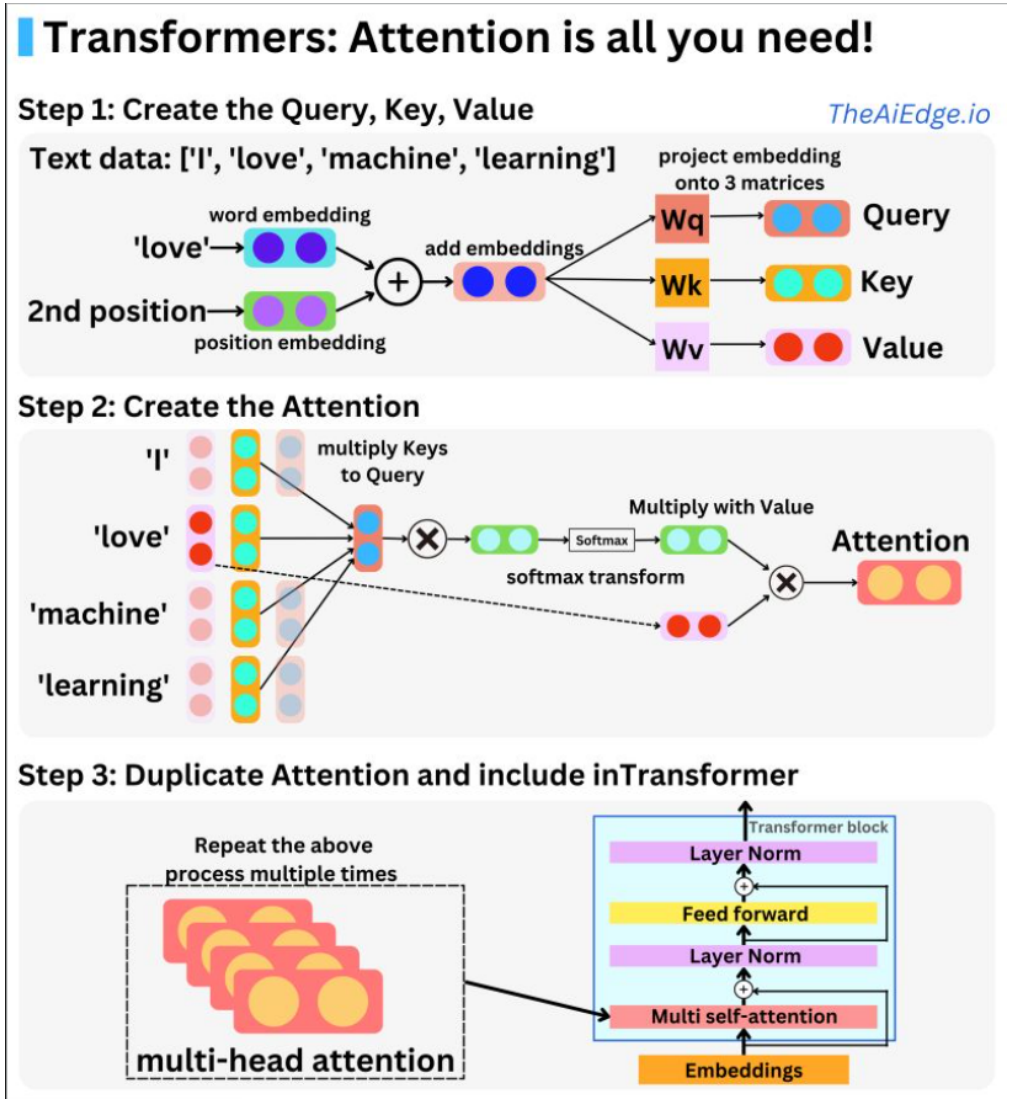
- One tricky part about understanding this set of calculations is keeping in mind that it is calculating attention for every element of our input sequence, for every word in our sentence, not just the most recent word. It's also calculating attention for earlier words. We don't really care about these because their next words have already been predicted and established. It's also calculating attention for future words. These don't have much use yet, because they are too far out and their immediate predecessors haven't yet been chosen. But there are indirect paths through which these calculations can effect the attention for the most recent word, so we include them all. It's just that when we get to the end and calculate word probabilities for every position in the sequence, we throw away most of them and only pay attention to the next word.
- The masking block enforces the constraint that, at least for this sequence completion task, we can't look into the future. It avoids introducing any weird artifacts from imaginary future words. It is crude and effective – manually set the attention paid to all words past the current position to negative infinity to prevent attention to subsequent positions. In The Annotated Transformer, an immeasurably helpful companion to the paper showing line by line Python implementation, the mask matrix is visualized. Purple cells show where attention is disallowed. Each row corresponds to an element in the sequence. The first row is allowed to attend to itself (the first element), but to nothing after. The last row is allowed to attend to itself (the final element) and everything that comes before. The Mask is an $[n \times n]$ matrix. It is applied not with a matrix multiplication, but with a more straightforward element-by-element multiplication. This has the effect of manually going in to the attention matrix and setting all of the purple elements from the mask to negative infinity. The following diagram shows an attention mask for sequence completion (source):

Back to Top



- Another important difference in how attention is implemented is that it makes use of the order in which words are presented to it in the sequence, and represents attention not as a word-to-word relationship, but as a position-to-position relationship. This is evident in its $[n \times n]$ shape. It maps each element from the sequence, indicated by the row index, to some other element(s) of the sequence, indicated by the column index. This helps us to visualize and interpret what it is doing more easily, since it is operating in the embedding space. We are spared the extra step of finding nearby word in the embedding space to represent the relationships between queries and keys.

Back to Top


Transformers: Attention is all you need!
Step 1: Create the Query, Key, Value
TheAiEdge.io
Text data: ['I', 'love', 'machine', 'learning']
project embedding onto 3 matrices
word embedding
'love'
2nd position
position embedding
add embeddings
Wq
Wk
Wv
Query
Key
Value
Step 2: Create the Attention
'I'
'love'
'machine'
'learning'
multiply Keys to Query
Softmax
softmax transform
Multiply with Value
Attention
Step 3: Duplicate Attention and include inTransformer
Repeat the above process multiple times
multi-head attention
Transformer block
Layer Norm
Feed forward
Layer Norm
Multi self-attention
Embeddings

Back to Top

- In an attention-based model like the transformer, the word embeddings are combined with attention weights that are learned during training. These weights indicate how much attention should be given to each word in the input sequence when making predictions. By dynamically adjusting the attention weights, the model can focus on different parts of the input sequence and better capture the context in which a word appears. As the paper states, the attention mechanism is what has revolutionized Transformers to what we see them to be today.
- Upon encoding a word as an embedding vector, we can also encode the position of that word in the input sentence as a vector (positional embeddings), and add it to the word embedding. This way, the same word at a different position in a sentence is encoded differently.
- The attention mechanism works with the inclusion of three vectors: key, query, value. Attention is the mapping between a query and a set of key-value pairs to an output. We start off by taking a dot product of query and key vectors to understand how similar they are. Next, the Softmax function is used to normalize the similarities of the resulting query-key vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key. (source)
- Thus, the basis behind the concept of attention is: "How much attention a word should pay to another word in the input to understand the meaning of the sentence?"
- As indicated in the section on Attention Calculation, one of the benefits of self-attention over recurrence is that it's highly parallelizable. In other words, the attention mechanism is performed in parallel for each word in the sentence to obtain their updated features in one shot. Furthermore, learning long-term/long-range dependencies in sequences is another benefit.

Averaging is Equivalent to Uniform Attention

- On a side note, it is worthwhile noting that the averaging operation is equivalent to uniform attention with the weights being all equal to $\frac{1}{n}$, where n is the number of words in the input sequence. In other wo

Back to Top

- The transformer does not use an activation function following the multi-head attention layer, but does use the ReLU activation post the two fully-connected layers that form the feed-forward network.
- The reason behind this goes back to the purpose of self-attention. The measure between word-vectors is generally computed through cosine-similarity because in the dimensions word tokens exist, it's highly unlikely for two words to be collinear even if they are trained to be closer in value if they are similar. However, two trained tokens will have higher cosine-similarity if they are semantically closer to each other than two completely unrelated words.
- This fact is exploited by the self-attention mechanism; after several of these matrix multiplications, the dissimilar words will zero out or become negative due to the dot product between them, and the similar words will stand out in the resulting matrix.
- Thus, self attention can be viewed as a weighted average, where less similar words become averaged out faster (toward the zero vector, on average), thereby achieving groupings of important and unimportant words (i.e. attention). The weighting happens through the dot product. If input vectors were normalized, the weights would be exactly the cosine similarities.
- The important thing to take into consideration is that within the self-attention mechanism, there are no inherent parameters; those linear operations are just there to capture the relationship between the different vectors by using the properties of the vectors used to represent them, leading to attention weights.

Attention in Transformers: What's New and What's Not?

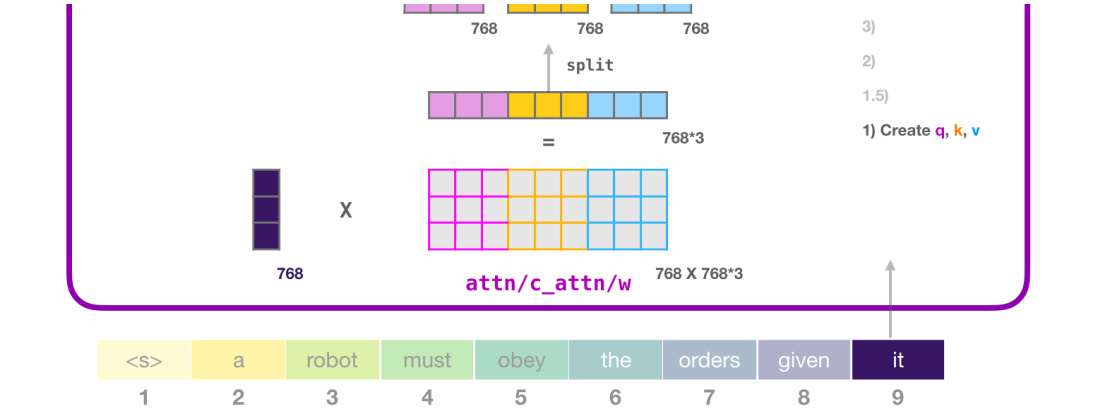
- The seq2seq encoder-decoder architecture that Vaswani et al. used is an idea adopted from one of Bengio's papers, Neural Machine Translation by Jointly Learning to Align and Translate.
- Further, Transformers use scaled dot-product attention (based on Query, Key and Value matrices) which is a concept inspired from the field of information retrieval (note that Bengio's seq2seq architecture group used Bahdanau attention in Neural Machine Translation by Jointly Learning to Align and Translate which is a more relatively basic form of attention compared to what Transformers use).

Back to Top

Calculating $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ Matrices in the Transformer Architecture

- Each word is embedded into a vector of size 512 and is fed into the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512 – in the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. The size of this list is hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.
- In the self-attention layers, multiplying the input vector (which is the word embedding for the first block of the encoder/decoder stack, while the output of the previous block for subsequent blocks) by the attention weights matrix (which are the $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ matrices stacked horizontally) and adding a bias vector afterwards results in a contcatenated key, value, and query vector for this token. This long vector is split to form the $\mathbf{q}$, $\mathbf{k}$, and $\mathbf{v}$ vectors for this token (which actually respresent the concatenated output for multiple attention heads and is thus, further reshaped into $\mathbf{q}$, $\mathbf{k}$, and $\mathbf{v}$ outputs for each attention head — more on this in the section on Multi-head Attention). From Jay Alammar's: The Illustrated GPT-2:

Back to Top



Applications of Attention in Transformers

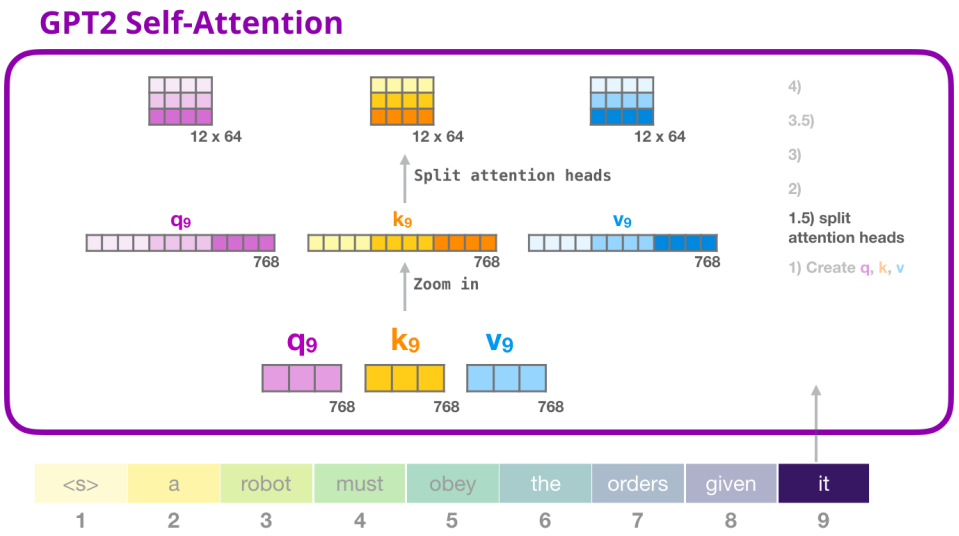
- From the paper, the Transformer uses multi-head attention in three different ways:
  - The encoder contains self-attention layers. In a self-attention layer, all of the keys, values, and queries are derived from the same source, which is the word embedding for the first block of the encoder stack, while the output of the previous block for subsequent blocks. Each position in the encoder can attend to all positions in the previous block of the encoder.
  - Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out all values (by setting to a very low value, such as $-\infty$) in the input of the softmax which correspond to illegal connections.
  - In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models such as Neural Machine Translation

Multi-Head Attention

- Let's confront some of the simplistic assumptions we made during our first pass through explaining the attention mechanism. Words are represented as dense embedded vectors, rather than one-hot vectors. Attention isn't just 1 or 0, on or off, but can also be anywhere in between. To get the results to fall between 0 and 1, we use the softmax trick again. It has the dual benefit of forcing all the values to lie in our [0, 1] attention range, and it helps to emphasize the highest value, while aggressively squashing the smallest. It's the differential almost-argmax behavior we took advantage of before when interpreting the final output of the model.
- An complicating consequence of putting a softmax function in attention is that it will tend to focus on a single element. This is a limitation we didn't have before. Sometimes it's useful to keep several of the preceding words in mind when predicting the next, and the softmax just robbed us of that. This is a problem for the model.
- To address the above issues, the Transformer paper refined the self-attention layer by adding a mechanism called "multi-headed" attention. This improves the performance of the attention layer in two ways:
    - It expands the model's ability to focus on different positions. It would be useful if we're translating a sentence like "The animal didn't cross the street because it was too tired", we would want to know which word "it" refers to.
    - It gives the attention layer multiple "representation subspaces". As we'll see next, with multi-headed attention we have not only one, but multiple sets of $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.
    - Further, getting the straightforward dot-product attention mechanism to work can be tricky. Bad random initializations of the learnable weights can de-stabilize the training process.

Back to Top

- To accomplish multi-head attention, self attention is simply conducted multiple times on different parts of the $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ matrices (each part corresponding to each attention head). Each $\mathbf{q}$, $\mathbf{k}$, and $\mathbf{v}$ vector generated at the output contains concatenated output corresponding to contains each attention head. To obtain the output corresponding to each attention heads, we simply reshape the long $\mathbf{q}$, $\mathbf{k}$, and $\mathbf{v}$ self-attention vectors into a matrix (with each row corresponding to the output of each attention head). From Jay Alammar's: The Illustrated GPT-2:



- Mathematically,

$$\mathbf{h}_i^{\ell+1} = \text{Concat}\ (\text{head}_1, \ldots, \text{head}_K)\, \mathbf{O}^{\ell}$$

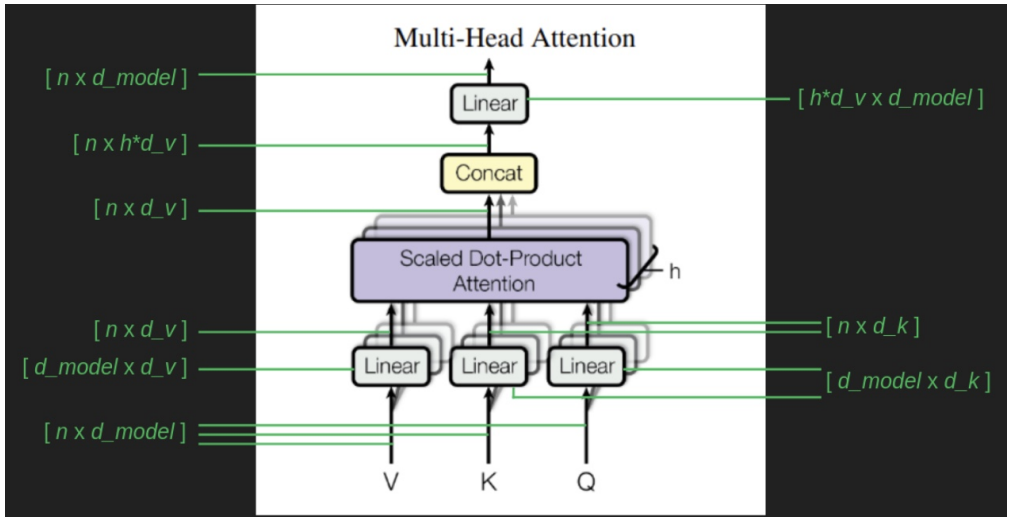
$$\text{head}_k = \text{Attention}\ \left(\mathbf{Q}^{k,\ell}\mathbf{h}_i^{\ell}, \mathbf{K}^{k,\ell}\mathbf{h}_j^{\ell}, \mathbf{V}^{k,\ell}\mathbf{h}_j^{\ell}\right)$$

Back to Top

Why Multiple Heads of Attention? Why Attention?.

Managing Computational Load Due to Multi-head Attention

- Unfortunately, multi-head attention really increases the computational load. Computing attention was already the bulk of the work, and we just multiplied it by however many heads we want to use. To get around this, we can re-use the trick of projecting everything into a lower-dimensional embedding space. This shrinks the matrices involved which dramatically reduces the computation time.
- To see how this plays out, we can continue looking at matrix shapes. Tracing the matrix shape through the branches and weaves of the multi-head attention blocks requires three more numbers.
  - $\mathbf{d_k}$: dimensions in the embedding space used for keys and queries (64 in the paper).
  - $\mathbf{d_v}$: dimensions in the embedding space used for values (64 in the paper).
  - $\mathbf{h}$: the number of heads (8 in the paper).



Back to Top

$[n \times d_v]$. It confuses things a little that d_k and d_v are the same in the paper, but they don't have to be. An important aspect of this setup is that each attention head has its own W_v, W_q, and W_k transforms. That means that each head can zoom in and expand the parts of the embedded space that it wants to focus on, and it can be different than what each of the other heads is focusing on.

- The result of each attention head has the same shape as V. Now we have the problem of h different result vectors, each attending to different elements of the sequence. To combine these into one, we exploit the powers of linear algebra, and just concatenate all these results into one giant $[n \times h * d_v]$ matrix. Then, to make sure it ends up in the same shape it started, we use one more transform with the shape $[h * d_v \times d_{model}]$.
- Here's all of the that from the paper, stated tersely.

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \\ \text{where head} &= \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \end{aligned}$$

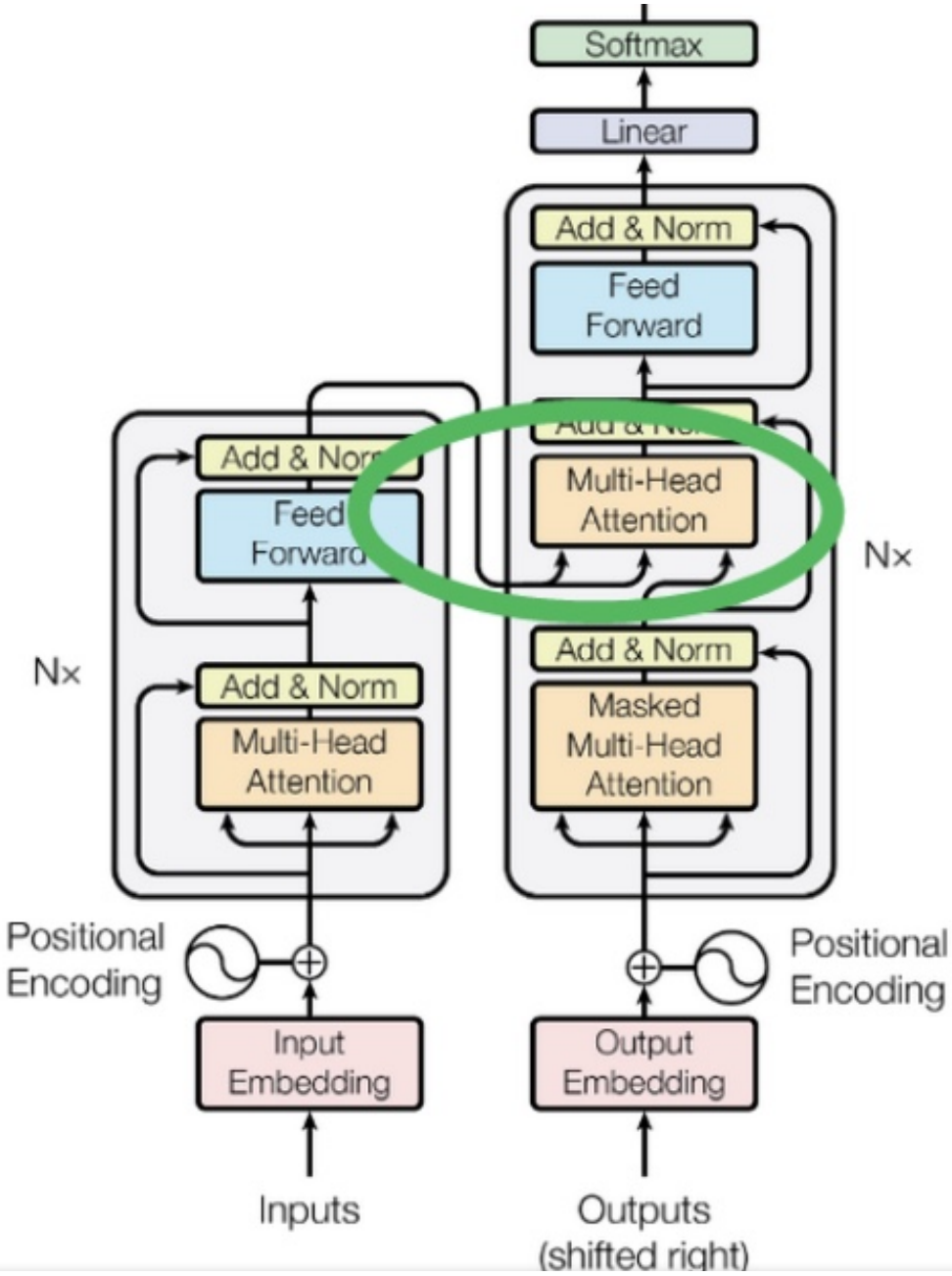
  - where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

Cross-Attention

- The final step in getting the full transformer up and running is the connection between the encoder and decoder stacks, the cross attention block. We've saved it for last and, thanks to the groundwork we've laid, there's not a lot left to explain.
- Cross-attention works just like self-attention with the exception that the key matrix K and value matrix V are based on the output of the encoder stack (i.e., the final encoder layer), rather than the output of the previous decoder layer. The query matrix Q is still calculated from the results of the previous decoder layer. This is the channel by which information from the source sequence makes its way into the ta

architecture.

Back to Top

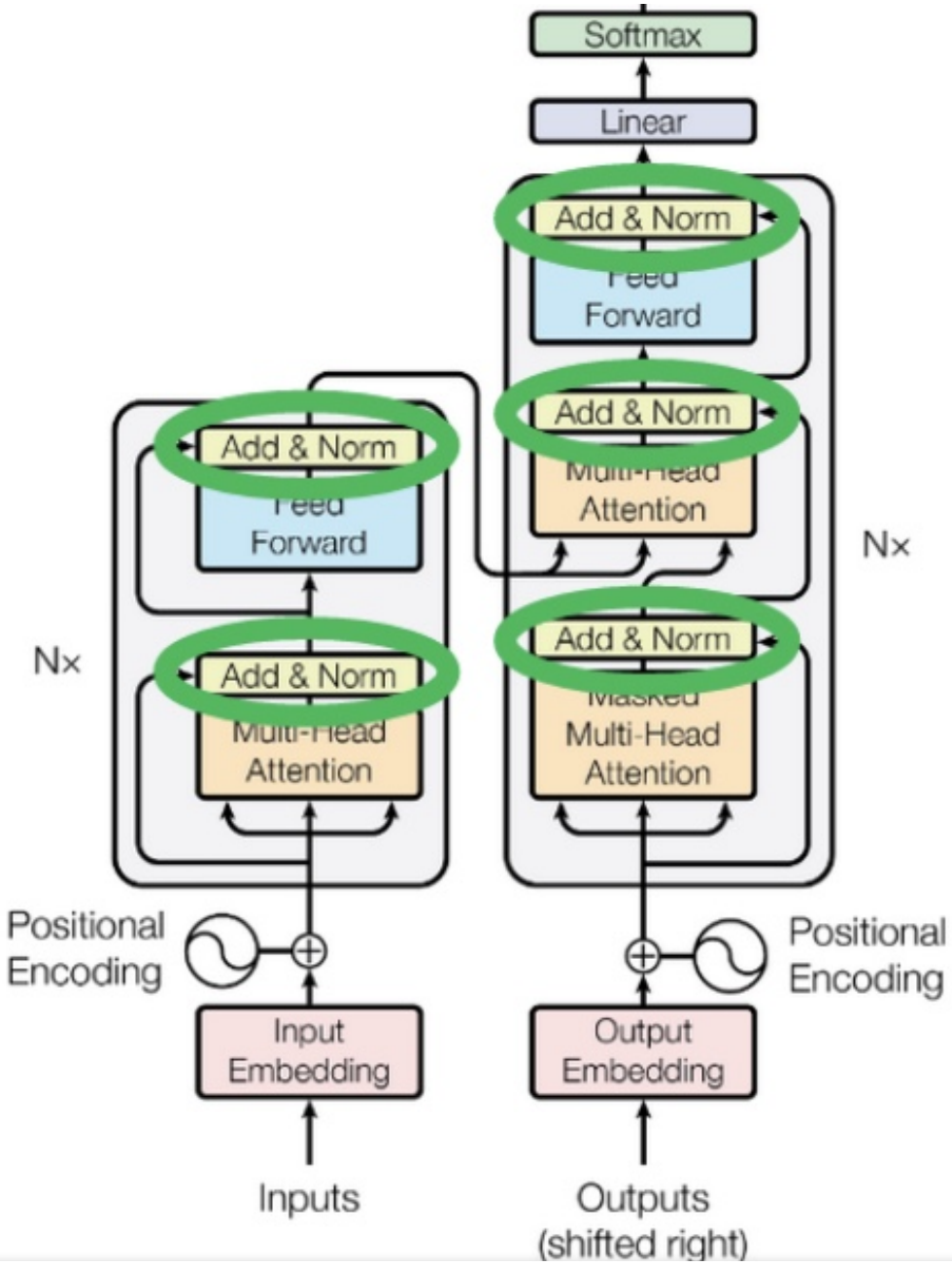

Softmax
Linear
Add & Norm
Feed Forward
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Back to Top

traversed it had a pretty concrete level. Everything from here on out is the plumbing necessary to make it work well. It's the rest of the harness that lets attention pull our heavy workloads.

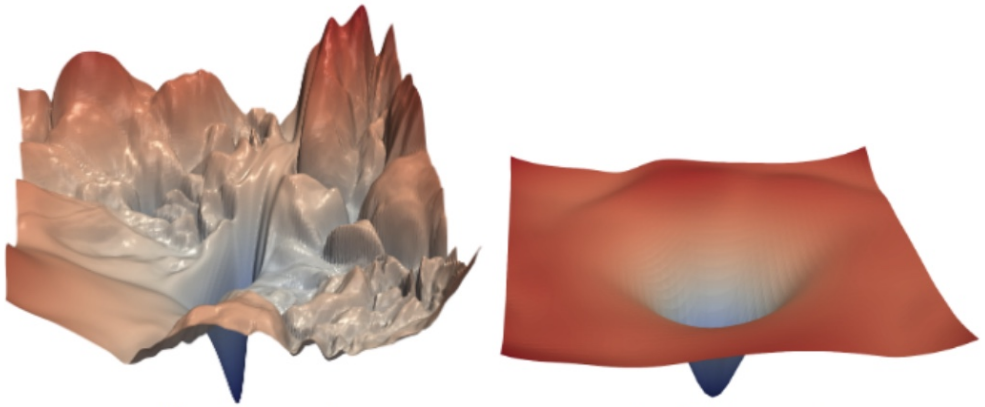
- One piece we haven't explained yet are skip connections. These occur around the Multi-Head Attention blocks, and around the element wise Feed Forward blocks in the blocks labeled "Add and Norm". In skip connections, a copy of the input is added to the output of a set of calculations. The inputs to the attention block are added back in to its output. The inputs to the element-wise feed forward block are added to its outputs. The following diagram shows the Transformer architecture showing add and norm blocks.

Back to Top


Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Back to Top

which means that when it's working correctly it will block most of what tries to pass through it. The result of this is that small changes in a lot of the inputs may not produce much change in the outputs if they happen to fall into channels that are blocked. This produces dead spots in the gradient where it is flat, but still nowhere near the bottom of a valley. These saddle points and ridges are a big tripping point for backpropagation. Skip connections help to smooth these out. In the case of attention, even if all of the weights were zero and all the inputs were blocked, a skip connection would add a copy of the inputs to the results and ensure that small changes in any of the inputs will still have noticeable changes in the result. This keeps gradient descent from getting stuck far away from a good solution. Skip connections have become popular because of how they improve performance since the days of the ResNet image classifier. They are now a standard feature in neural network architectures. The figure below (source) shows the effect that skip connections have by comparing a ResNet with and without skip connections. The slopes of the loss function hills are are much more moderate and uniform when skip connections are used. If you feel like taking a deeper dive into how the work and why, there's a more in-depth treatment in this post. The following diagram shows the comparison of loss surfaces with and without skip connections.



(a) without skip connections (b) with skip connections

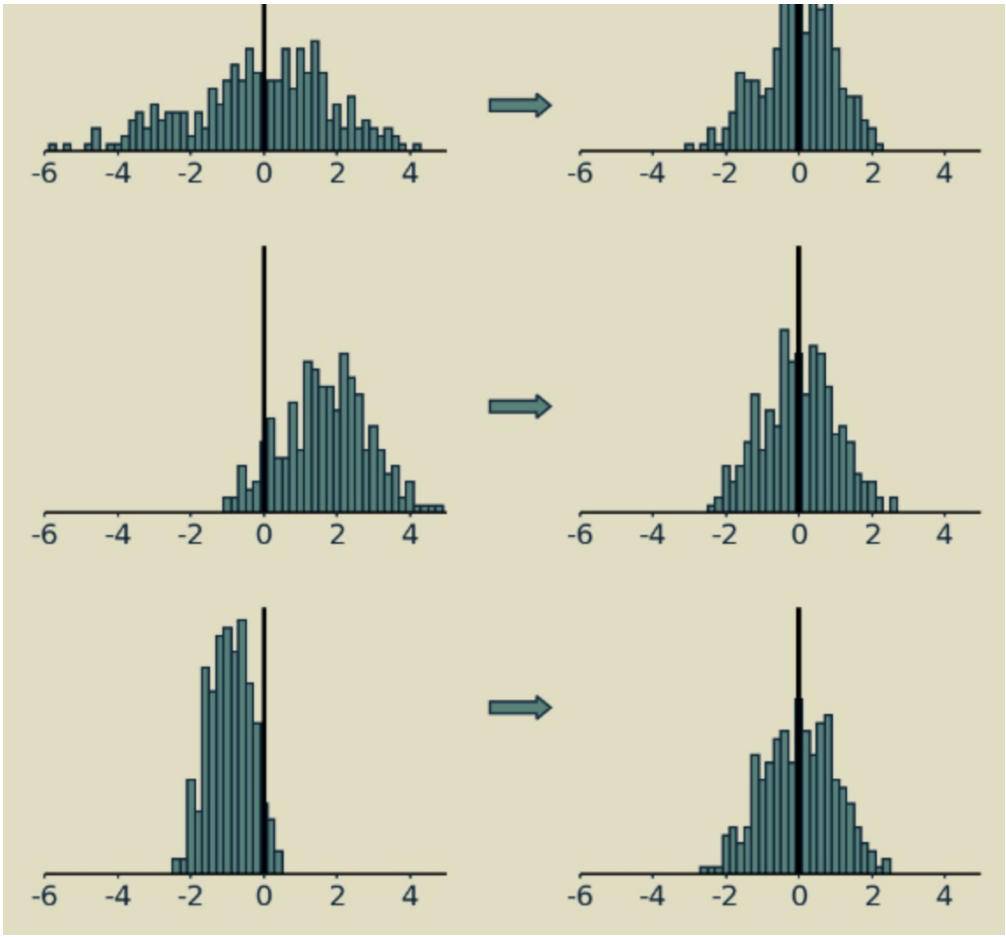
Back to Top

and manually adds it back into the signal, so that there's no way it can be dropped or forgotten. This source of robustness may be one of the reasons for transformers' good behavior in so many varied sequence completion tasks.

Layer Normalization

- Normalization is a step that pairs well with skip connections. There's no reason they necessarily have to go together, but they both do their best work when placed after a group of calculations, like attention or a feed forward neural network.
- The short version of layer normalization is that the values of the matrix are shifted to have a mean of zero and scaled to have a standard deviation of one. The following diagram shows several distributions being normalized.

Back to Top



- The longer version is that in systems like transformers, where there are a lot of moving pieces and some of them are something other than matrix multiplications (such as softmax operators or rectified linear units), it matters how big values are and how they're balanced between positive and negative. If everything is linear, you can double all your inputs, and your outputs will be twice as big, and everything will work just fine. Not so with neural networks. They are inherently nonlinear, which makes them very expressive but also sensitive to signals' magnitudes and distributions. Normalization is a technique that has proven useful

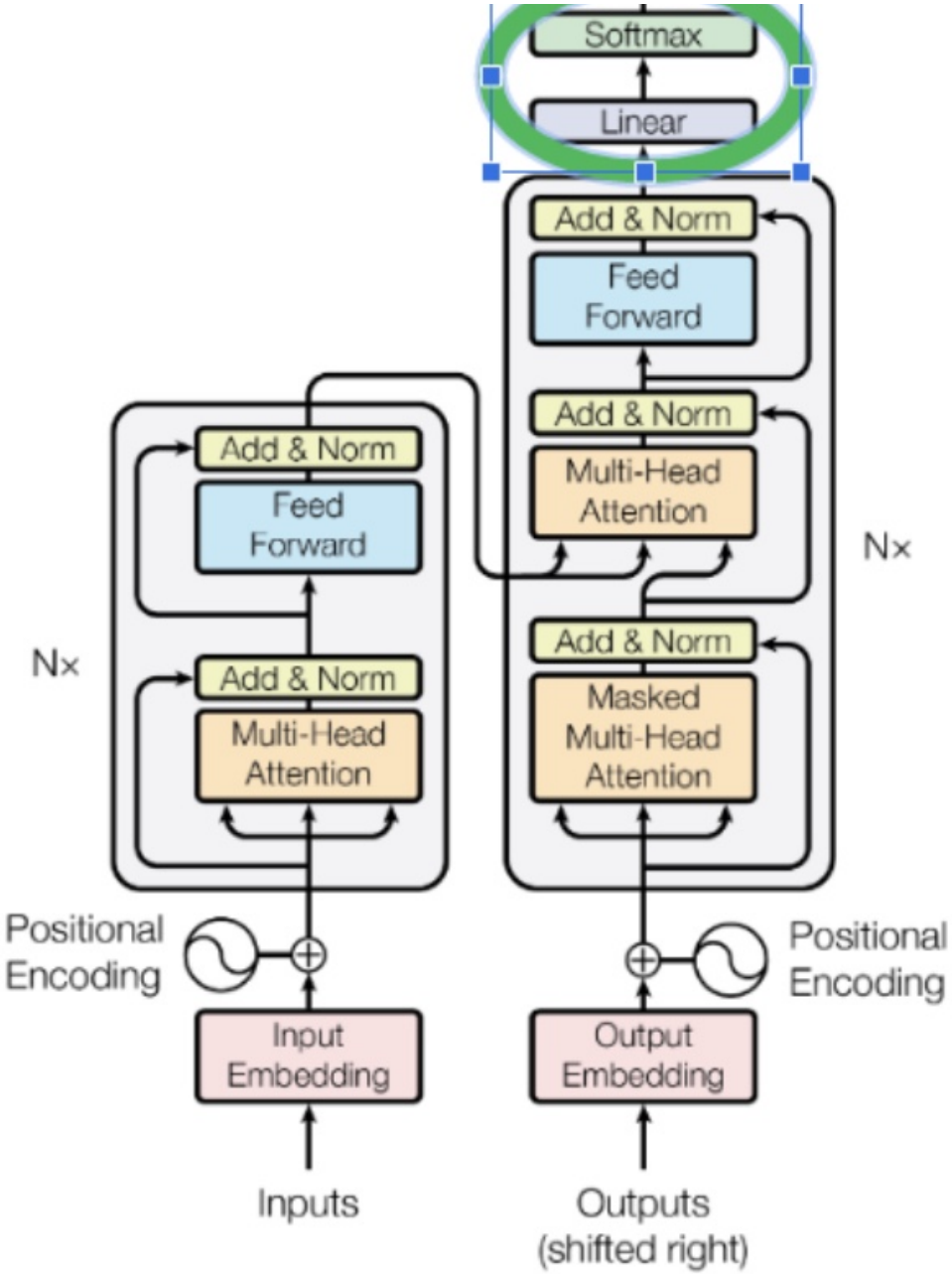
includes batch normalization, a close cousin of the layer normalization used in transformers.

Softmax

- The argmax function is “hard” in the sense that the highest value wins, even if it is only infinitesimally larger than the others. If we want to entertain several possibilities at once, it’s better to have a “soft” maximum function, which we get from **softmax**. To get the softmax of the value $\mathbf{x}$ in a vector, divide the exponential of $\mathbf{x}$, $\mathbf{e}^{\mathbf{x}}$, by the sum of the exponentials of all the values in the vector. This converts the (unnormalized) logits/energy values into (normalized) probabilities $\in [0, 1]$, with all summing up to 1.
- The softmax is helpful here for three reasons. First, it converts our de-embedding results vector from an arbitrary set of values to a probability distribution. As probabilities, it becomes easier to compare the likelihood of different words being selected and even to compare the likelihood of multi-word sequences if we want to look further into the future.
- Second, it thins the field near the top. If one word scores clearly higher than the others, softmax will exaggerate that difference (owing to the “exponential” operation), making it look almost like an argmax, with the winning value close to one and all the others close to zero. However, if there are several words that all come out close to the top, it will preserve them all as highly probable, rather than artificially crushing close second place results, which argmax is susceptible to. You might be thinking what the difference between standard normalization and softmax is – after all, both rescale the logits between 0 and 1. By using softmax, we are effectively “approximating” argmax as indicated earlier while gaining differentiability. Rescaling doesn’t weigh the max significantly higher than other logits, whereas softmax does due to its “exponential” operation. Simply put, softmax is a “softer” argmax.
- Third, softmax is differentiable, meaning we can calculate how much each element of the results will change, given a small change in any of the input elements. This allows us to use it with backpropagation to train our transformer.

Back to Top

Back to Top

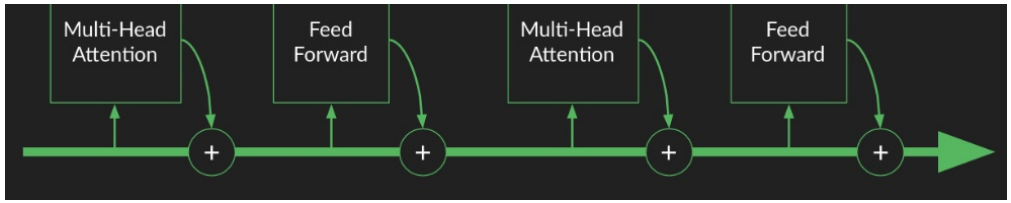

Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Back to Top

with carefully chosen weights were enough to make a decent language model. Most of the weights were zeros in our examples, a few of them were ones, and they were all hand picked. When training from raw data, we won't have this luxury. At the beginning the weights are all chosen randomly, most of them are close to zero, and the few that aren't probably aren't the ones we need. It's a long way from where it needs to be for our model to perform well.

- Stochastic gradient descent through backpropagation can do some pretty amazing things, but it relies a lot on trial-and-error. If there is just one way to get to the right answer, just one combination of weights necessary for the network to work well, then it's unlikely that it will find its way. But if there are lots of paths to a good solution, chances are much better that the model will get there.
- Having a single attention layer (just one multi-head attention block and one feed forward block) only allows for one path to a good set of transformer parameters. Every element of every matrix needs to find its way to the right value to make things work well. It is fragile and brittle, likely to get stuck in a far-from-ideal solution unless the initial guesses for the parameters are very very lucky.
- The way transformers sidestep this problem is by having multiple attention layers, each using the output of the previous one as its input. The use of skip connections make the overal pipeline robust to individual attention blocks failing or giving wonky results. Having multiples means that there are others waiting to take up the slack. If one should go off the rails, or in any way fail to live up to its potential, there will be another downstream that has another chance to close the gap or fix the error. The paper showed that more layers resulted in better performance, although the improvement became marginal after 6.
- Another way to think about multiple layers is as a conveyor belt assembly line. Each attention block and feedforward block has the chance to pull inputs off the line, calculate useful attention matrices and make next word predictions. Whatever results they produce, useful or not, get added back onto the conveyer, and passed to the next layer. The following diagram shows the transformer redrawn as a conveyor belt:

Back to Top



- This is in contrast to the traditional description of many-layered neural networks as “deep”. Thanks to skip connections, successive layers don’t provide increasingly sophisticated abstraction as much as they provide redundancy. Whatever opportunities for focusing attention and creating useful features and making accurate predictions were missed in one layer can always be caught by the next. Layers become workers on the assembly line, where each does what it can, but doesn’t worry about catching every piece, because the next worker will catch the ones they miss.

Transformer Encoder and Decoder

- The Transformer model has two parts: encoder and decoder. Both encoder and decoder are mostly identical (with a few differences) and are comprised of a stack of transformer blocks. Each block is comprised of a combination of multi-head attention blocks, positional feedforward layers, residual connections and layer normalization blocks.
- The attention layers from the encoder and decoder have the following differences:
  - The encoder only has self-attention blocks while the decoder has a cross-attention encoder-decoder layer sandwiched between the self-attention layer and the feedforward neural network.
  - Also, the self-attention blocks are masked to ensure causal predictions (i.e., the prediction of token $\mathbf{N}$ only depends on the previous $\mathbf{N}-1$ tokens, and not on the future ones).
- Each of the encoding/decoding blocks contains many stacked encoders/decoder transformer blocks. The Transformer encoder is a stack of six encoders, while the decoder is a stack of six decoders. The i layers capture more basic patterns (broadly speaking, basic syntactic patterns), whereas the last layers

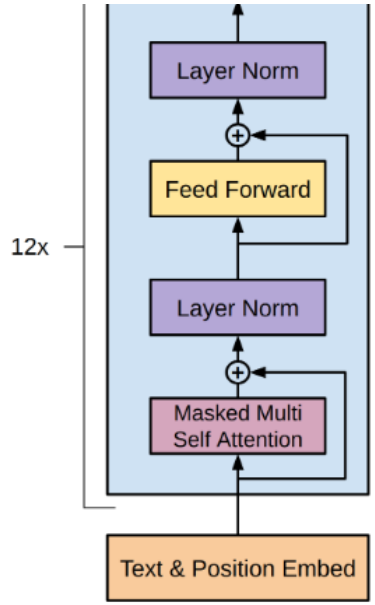
- The six encoders and decoders are identical in structure but do not share weights. Check weights shared by different parts of a transformer model for a detailed discourse on weight sharing opportunities within the Transformer layers.
- For more on the pros and cons of the encoder and decoder stack, refer Autoregressive vs. Autoencoder Models.

Decoder Stack

> *The decoder, which follows the auto-regressive property, i.e., consumes the tokens generated so far to generate the next one, is used standalone for generation tasks, such as tasks in the domain of natural language generation (NLG), for e.g., such as summarization, translation, or abstractive question answering. Decoder models are typically trained with an objective of predicting the next token, i.e., "autoregressive blank infilling".*

- As we laid out in the section on Sampling a Sequence of Output Words, the decoder can complete partial sequences and extend them as far as you want. OpenAI created the generative pre-training (GPT) family of models to do just this, by training on a predicting-the-next-token objective. The architecture they describe in this report should look familiar. It is a transformer with the encoder stack and all its connections surgically removed. What remains is a 12 layer decoder stack. The following diagram from the GPT-1 paper Improving Language Understanding by Generative Pre-Training shows the architecture of the GPT family of models:

Back to Top



- Any time you come across a generative/auto-regressive model, such as GPT-X, LLaMA, Copilot, etc., you're probably seeing the decoder half of a transformer in action.

Encoder Stack

> *The encoder, is typically used standalone for content understanding tasks, such as tasks in the domain of natural language understanding (NLU) that involve classification, for e.g., sentiment analysis, or extractive question answering. Encoder models are typically trained with a "fill in the blanks"/"blank infilling" objective – reconstructing the original data from masked/corrupted input (i.e., by randomly sampling tokens from the input and replacing them with `[MASK]` elements, or shuffling sentences in random order if it's the next sente*

Back to Top

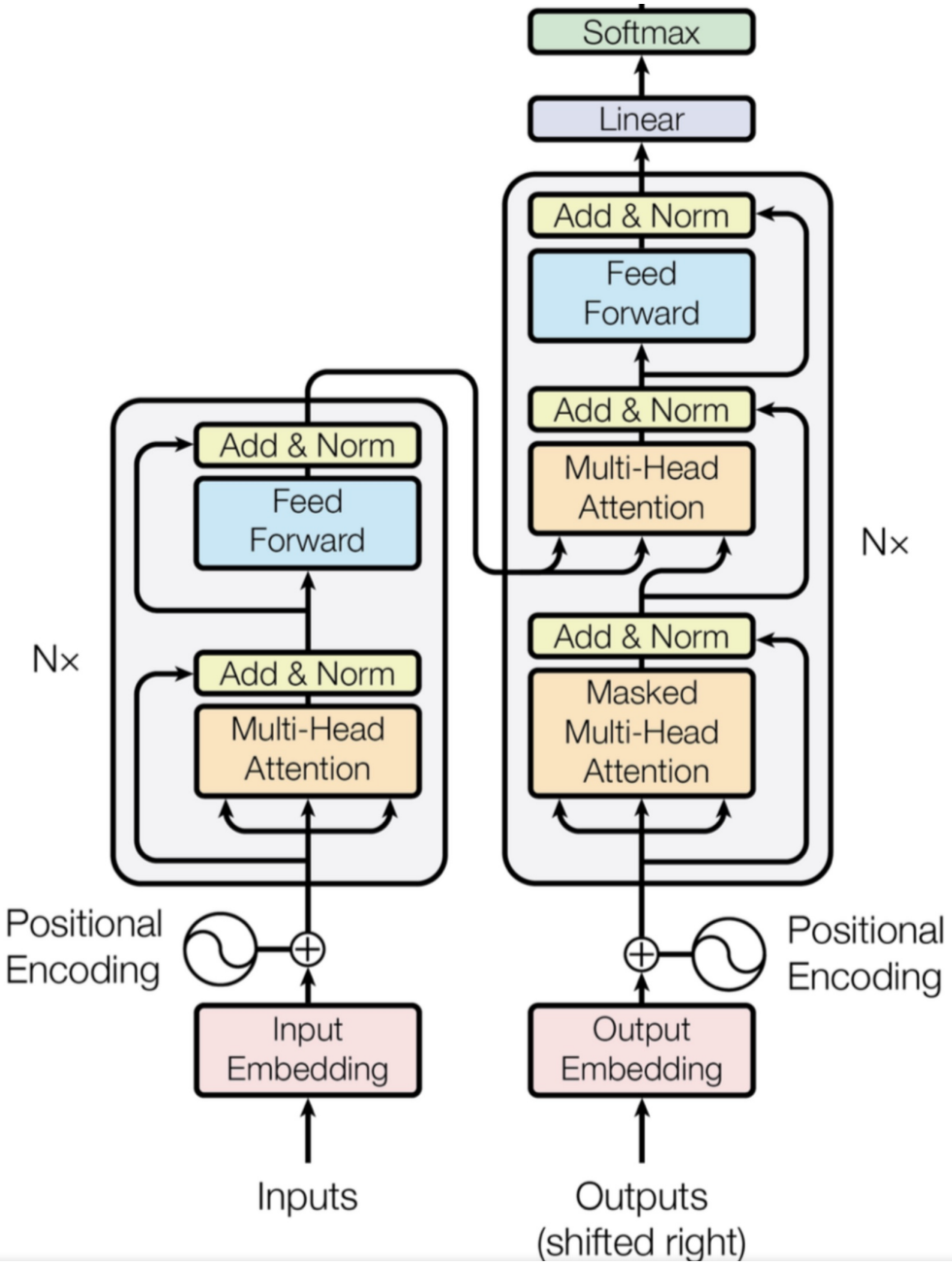
> *is that there's no explicit predictions being made at the end that we can use to judge the rightness or wrongness of its performance. Instead, the end product of an encoder stack is an abstract representation in the form of a sequence of vectors in an embedded space. It has been described as a pure semantic representation of the sequence, divorced from any particular language or vocabulary, but this feels overly romantic to me. What we know for sure is that it is a useful signal for communicating intent and meaning to the decoder stack.*

- Having an encoder stack opens up the full potential of transformers instead of just generating sequences, they can now translate (or transform) the sequence from one language to another. Training on a translation task is different than training on a sequence completion task. The training data requires both a sequence in the language of origin, and a matching sequence in the target language. The full language of origin is run through the encoder (no masking this time, since we assume that we get to see the whole sentence before creating a translation) and the result, the output of the final encoder layer is provided as an input to each of the decoder layers. Then sequence generation in the decoder proceeds as before, but this time with no prompt to kick it off.
- Any time you come across an encoder model that generates semantic embeddings, such as BERT, ELMo, etc., you're likely seeing the encoder half of a transformer in action.

Putting It All Together: the Transformer Architecture

- The Transformer architecture combines the individual encoder/decoder models. The encoder takes the input and encodes it into fixed-length query, key, and vector tensors (analogous to the fixed-length context vector in the original paper by Bahdanau et al. (2015)) that introduced attention. These tensors are passed onto the decoder which decodes it into the output sequence.
- The encoder (left) and decoder (right) of the transformer is shown below:

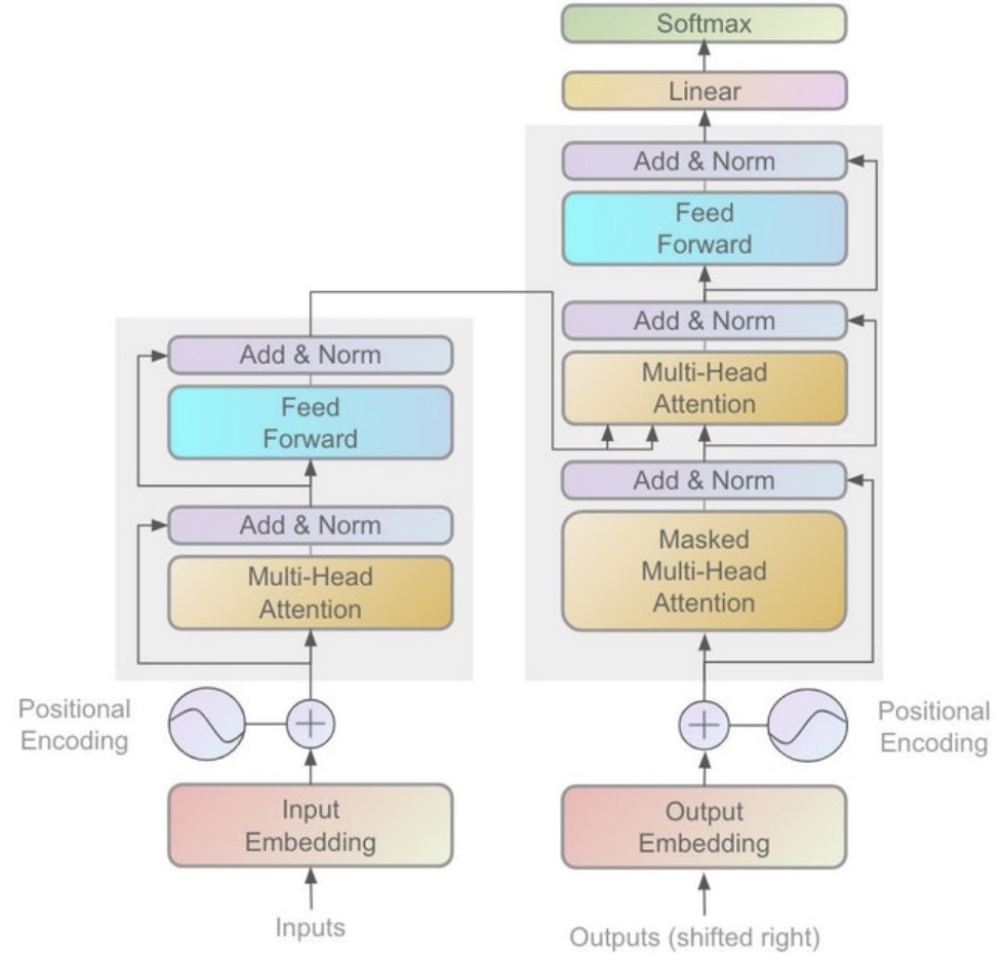
Back to Top


Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

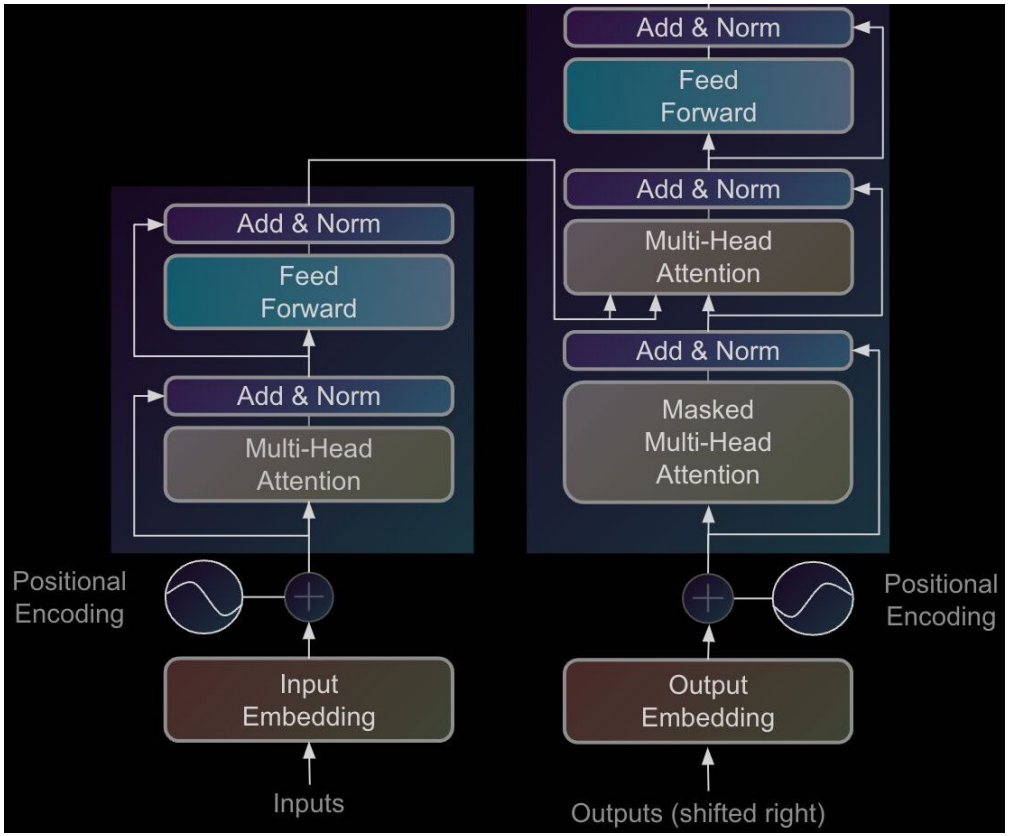
Back to Top

dot-product multi-head **cross** attention.

- Re-drawn vectorized versions from DAIR.AI are as follows:

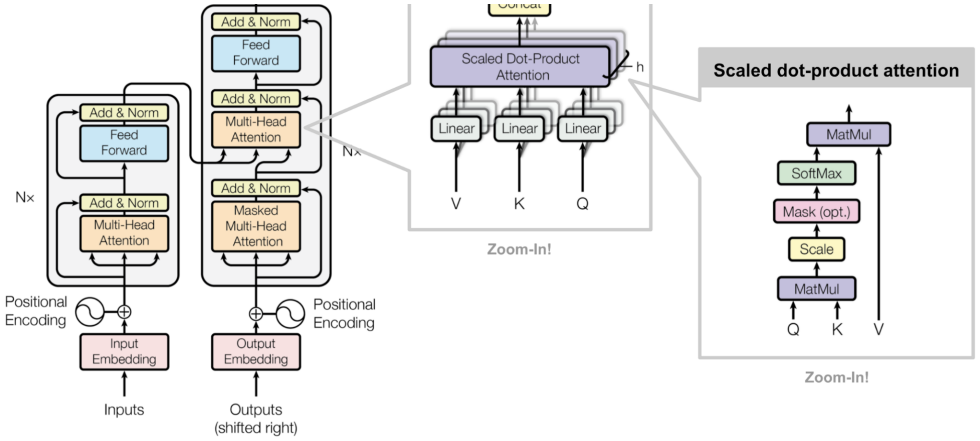


Back to Top

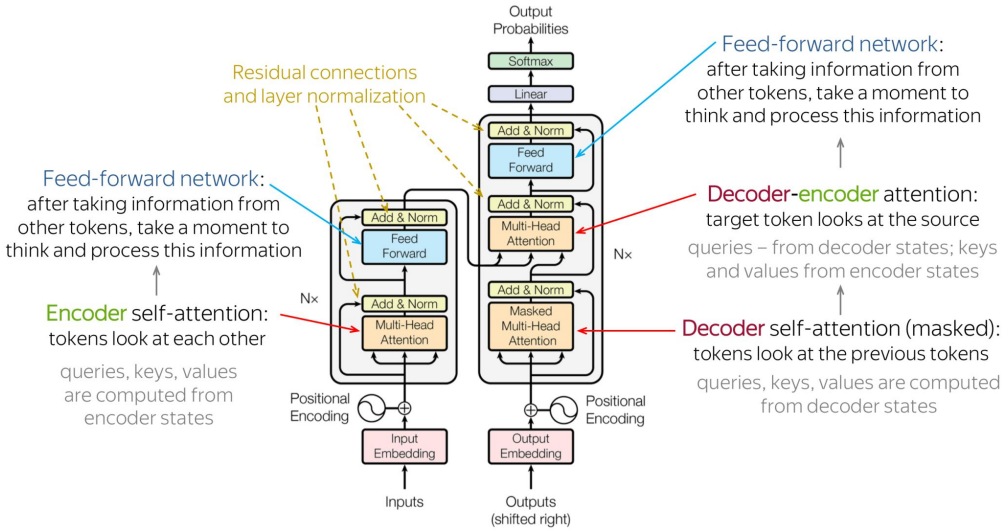


- The full model architecture of the transformer – from fig. 1 and 2 in Vaswani et al. (2017) – is as follows:

Back to Top



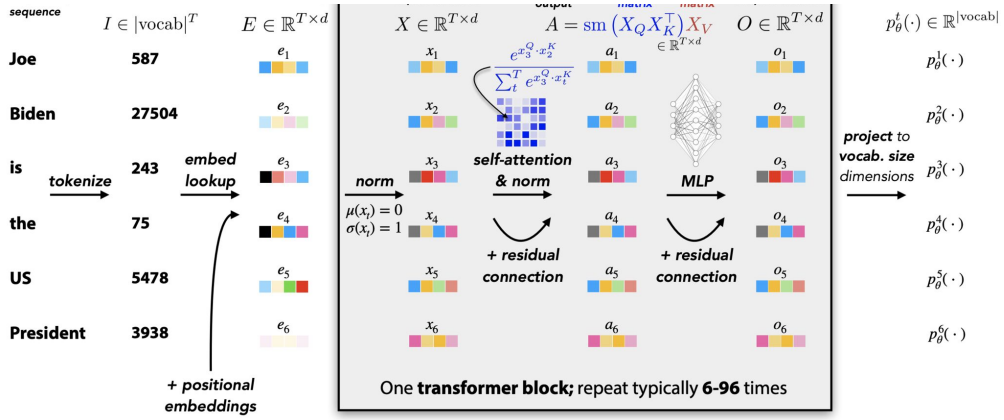
- Here is an illustrated version of the overall Transformer architecture from Abdullah Al Imran:



Back to Top

2. Tokenization: $\mathbf{I} \in |\text{vocab}|^{T}$.
3. Input embeddings lookup: $\mathbf{E} \in \mathbb{R}^{T \times d}$.
4. Inputs to Transformer block: $\mathbf{X} \in \mathbb{R}^{T \times d}$.
5. Obtaining three separate linear projections of input $\mathbf{X}$ (queries, keys, and values): $\mathbf{X_Q} = \mathbf{X}\mathbf{W_Q}, \quad \mathbf{X_K} = \mathbf{X}\mathbf{W_K}, \quad \mathbf{X_V} = \mathbf{X}\mathbf{W_V}$.
6. Calculating self-attention: $\mathbf{A} = \text{sm}\left(\mathbf{X_Q}\mathbf{X_K^T}\right)\mathbf{X_V}$ (the scaling part is missing in the figure below – you can reference the section on Types of Attention: Additive, Multiplicative (Dot-product), and Scaled for more).
    - This is followed by a residual connection and LayerNorm.
7. Feed-forward (MLP) layers which perform two linear transformations/projections of the input with a ReLU activation in between: $\text{FFN}(\mathbf{x}) = \max\left(0, \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1\right)\mathbf{W}_2 + \mathbf{b}_2$
    - This is followed by a residual connection and LayerNorm.
8. Output of the Transformer block: $\mathbf{O} \in \mathbb{R}^{T \times d}$.
9. Project to vocabulary size at time t: $\mathbf{p}_\theta^t(\cdot) \in \mathbb{R}^{|\text{vocab}|}$.

Back to Top



Loss Function

- The encoder and decoder are jointly trained ("end-to-end") to minimize the cross-entropy loss between the predicted probability matrix of shape `output sequence length` $\times$ `vocab` (right before taking the argmax on the output of the softmax to ascertain the next token to output), and the `output sequence length`-sized output vector of token IDs as the true label.
- Effectively, the cross-entropy loss "pulls" the predicted probability of the correct class towards 1 during training. This is accomplished by calculating gradients of the loss function w.r.t. the model's weights; with the model's sigmoid/softmax output (in case of binary/multiclass classification) serving as the prediction (i.e., the pre-argmax output is utilized since argmax is not differentiable).

Implementation Details

Tokenizing

Back to Top

Annotated Transformer will help you fill in these gaps.

- In the section on One-hot encoding, we discussed that a vocabulary could be represented by a high dimensional one-hot vector, with one element associated with each word. In order to do this, we need to know exactly how many words we are going to be representing and what they are.
- A naïve approach is to make a list of all possible words, like we might find in Webster's Dictionary. For the English language this will give us several tens of thousands, the exact number depending on what we choose to include or exclude. But this is an oversimplification. Most words have several forms, including plurals, possessives, and conjugations. Words can have alternative spellings. And unless your data has been very carefully cleaned, it will contain typographical errors of all sorts. This doesn't even touch on the possibilities opened up by freeform text, neologisms, slang, jargon, and the vast universe of Unicode. An exhaustive list of all possible words would be infeasibly long.
- A reasonable fallback position would be to have individual characters serve as the building blocks, rather than words. An exhaustive list of characters is well within the capacity we have to compute. However there are a couple of problems with this. After we transform data into an embedding space, we assume the distance in that space has a semantic interpretation, that is, we assume that points that fall close together have similar meanings, and points that are far away mean something very different. That allows us to implicitly extend what we learn about one word to its immediate neighbors, an assumption we rely on for computational efficiency and from which the transformer draws some ability to generalize.
- At the individual character level, there is very little semantic content. There are a few one character words in the English language for example, but not many. Emoji are the exception to this, but they are not the primary content of most of the data sets we are looking at. That leaves us in the unfortunate position of having an unhelpful embedding space.
- It might still be possible to work around this theoretically, if we could look at rich enough combinations of characters to build up semantically useful sequences like words, words stems, or word pairs. Unfortunately, the features that transformers create internally behave more like a collection of input pairs than an ordered set of inputs. That means that the representation of a word would be a collectio

Back to Top

Byte Pair Encoding (BPE)

- Fortunately, there is an elegant solution to this called byte pair encoding, which is a simple form of data compression in which the most common pair of consecutive bytes of data is replaced with a byte that does not occur within that data. A table of the replacements is required to rebuild the original data.
- Starting with the character level representation, each character is assigned a code, its own unique byte. Then after scanning some representative data, the most common pair of bytes is grouped together and assigned a new byte, a new code. This new code is substituted back into the data, and the process is repeated.

Example

- As an example (credit: Wikipedia: Byte pair encoding), suppose the data to be encoded is:

```
aaabdaaabac
```

- The byte pair “aa” occurs most often, so it will be replaced by a byte that is not used in the data, “Z”. Now there is the following data and replacement table:

```
ZabdZabac
Z=aa
```

- Then the process is repeated with byte pair “ab”, replacing it with Y:

Back to Top

- The only literal byte pair left occurs only once, and the encoding might stop here. Or the process could continue with recursive byte pair encoding, replacing “ZY” with “X”:

```
XdXac
X=ZY
Y=ab
Z=aa
```

- This data cannot be compressed further by byte pair encoding because there are no pairs of bytes that occur more than once.
- To decompress the data, simply perform the replacements in the reverse order.

Applying BPE to Learn New, Rare, and Misspelled Words

- Codes representing pairs of characters can be combined with codes representing other characters or pairs of characters to get new codes representing longer sequences of characters. There’s no limit to the length of character sequence a code can represent. They will grow as long as they need to in order to represent commonly repeated sequences. The cool part of byte pair encoding is that in infers which long sequences of characters to learn from the data, as opposed to dumbly representing all possible sequences. it learns to represent long words like transformer with a single byte code, but would not waste a code on an arbitrary string of similar length, such as `ksowjmckder`. And because it retains all the byte codes for its single character building blocks, it can still represent weird misspellings, new words, and even foreign languages.
- When you use byte pair encoding, you get to assign it a vocabulary size, ad it will keep building new codes until reaches that size. The vocabulary size needs to be big enough, that the character strings

Back to Top

which are hopefully recognizable words) and provides a concise code for each one. This is the process called tokenization.

Teacher Forcing

- Similar to recurrent neural networks, the teacher forcing strategy is used for training Transformers, which uses ground truth as input, instead of model output from a prior time step as an input. For more, check out What is Teacher Forcing for Recurrent Neural Networks? and What is Teacher Forcing?
    - **Pros:** Training with teacher forcing converges faster. At the early stages of training, the predictions of the model are very bad. If we do not use teacher forcing, the hidden states of the model will be updated by a sequence of wrong predictions, errors will accumulate, and it is difficult for the model to learn from that.
    - **Cons:** During inference, since there is usually no ground truth available, the RNN model will need to feed its own previous prediction back to itself for the next prediction. Therefore there is a discrepancy between training and inference, and this might lead to poor model performance and instability. This is known as Exposure Bias in literature.

Label Smoothing As a Regularizer

- During training, they employ label smoothing which penalizes the model if it gets overconfident about a particular choice. This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score.
- They implement label smoothing using the KL div loss. Instead of using a one-hot target distribution, we create a distribution that has a reasonably high confidence of the correct word and the rest of the smoothing mass distributed throughout the vocabulary.

Back to Top

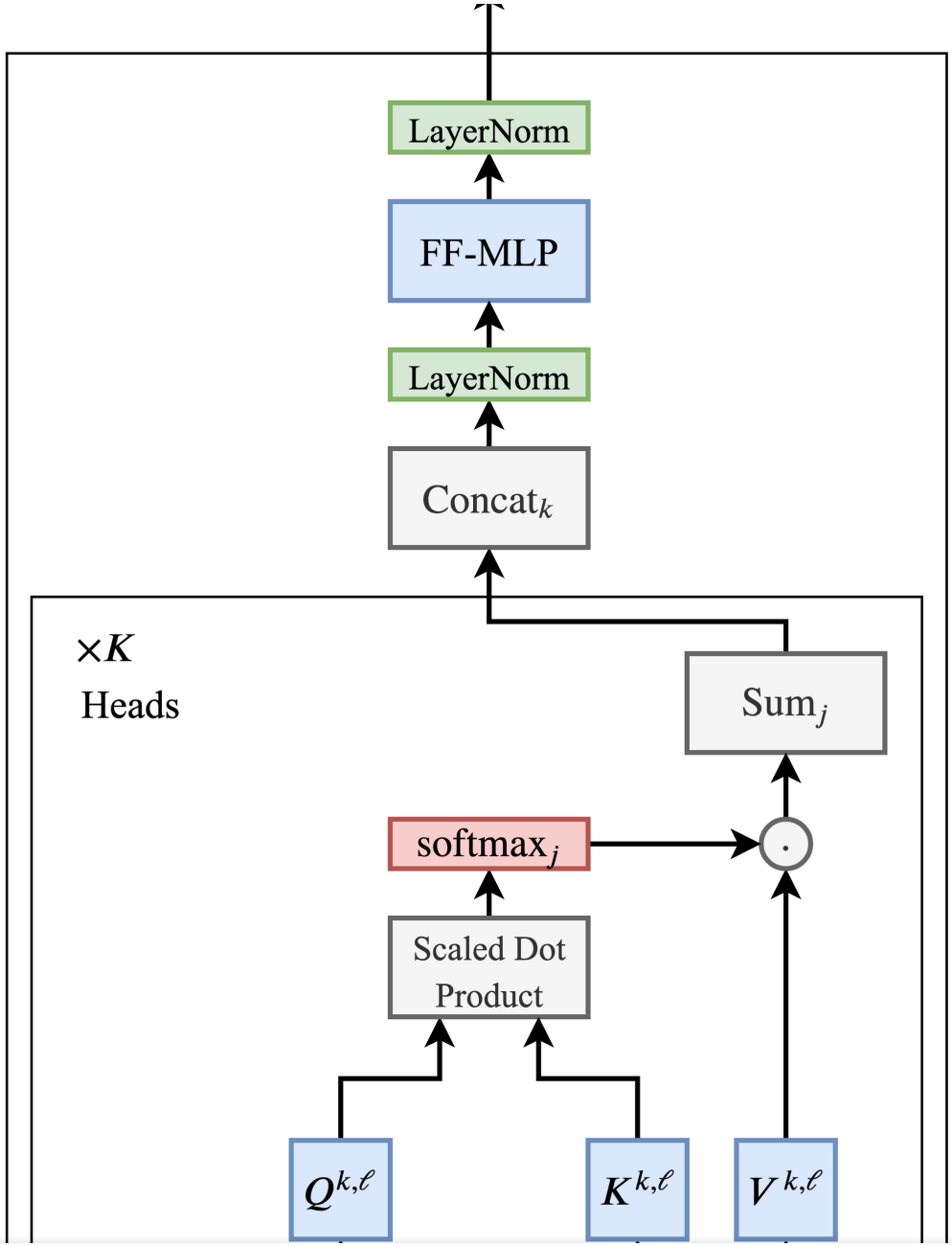
sharp or very distributed attention weights w_{ij} when summing over the features of the other words. **Scaling the dot-product** attention by the square-root of the feature dimension helps counteract this issue.

- Additionally, at the individual feature/vector entries level, concatenating across multiple attention heads-each of which might output values at different scales-can lead to the entries of the final vector $h_i^{\ell+1}$ having a wide range of values. Following conventional ML wisdom, it seems reasonable to add a normalization layer into the pipeline. As such, Transformers overcome this issue with **LayerNorm**, which normalizes and learns an affine transformation at the feature level.
- Finally, the authors propose another 'trick' to control the scale issue: a **position-wise 2-layer MLP** with a special structure. After the multi-head attention, they project $h_i^{\ell+1}$ to a (absurdly) higher dimension by a learnable weight, where it undergoes the ReLU non-linearity, and is then projected back to its original dimension followed by another normalization:

$$h_i^{\ell+1} = \text{LN}\left(\text{MLP}\left(\text{LN}\left(h_i^{\ell+1}\right)\right)\right)$$

- Since LayerNorm and scaled dot-products (supposedly) didn't completely solve the highlighted scaling issues, the over-parameterized feed-forward sub-layer was utilized. In other words, the big MLP is a sort of hack to re-scale the feature vectors independently of each other. According to Jannes Muenchmeyer, the feed-forward sub-layer ensures that the Transformer is a universal approximator. Thus, projecting to a very high dimensional space, applying a non-linearity, and re-projecting to the original dimension allows the model to represent more functions than maintaining the same dimension across the hidden layer would. The final picture of a Transformer layer looks like this:

Back to Top


LayerNorm
FF-MLP
LayerNorm
Concat$_k$
$\times K$
Heads
Sum$_j$
softmax$_j$
Scaled Dot Product
$Q^{k,\ell}$
$K^{k,\ell}$
$V^{k,\ell}$

h_i^{ℓ} $\{h_j^{\ell} \ \forall j \in S\}$

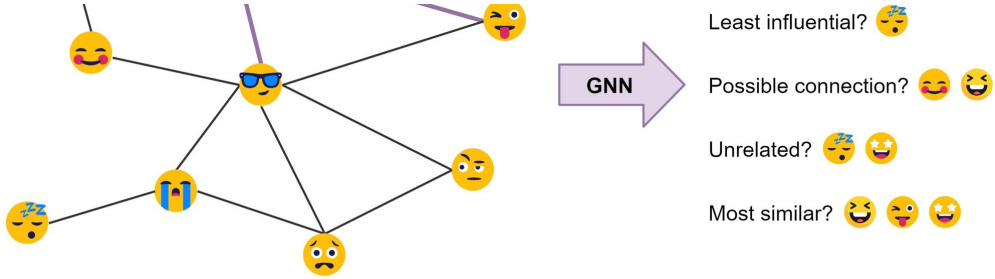
- The Transformer architecture is also extremely amenable to very deep networks, enabling the NLP community to scale up in terms of both model parameters and, by extension, data. **Residual connections** between the inputs and outputs of each multi-head attention sub-layer and the feed-forward sub-layer are key for stacking Transformer layers (but omitted from the diagram for clarity).

The Relation Between Transformers and Graph Neural Networks

GNNs Build Representations of Graphs

- Let's take a step away from NLP for a moment.
- Graph Neural Networks (GNNs) or Graph Convolutional Networks (GCNs) build representations of nodes and edges in graph data. They do so through neighbourhood aggregation (or message passing), where each node gathers features from its neighbours to update its representation of the local graph structure around it. Stacking several GNN layers enables the model to propagate each node's features over the entire graph—from its neighbours to the neighbours' neighbours, and so on.
- Take the example of this emoji social network below (source): The node features produced by the GNN can be used for predictive tasks such as identifying the most influential members or proposing potential connections.

Back to Top

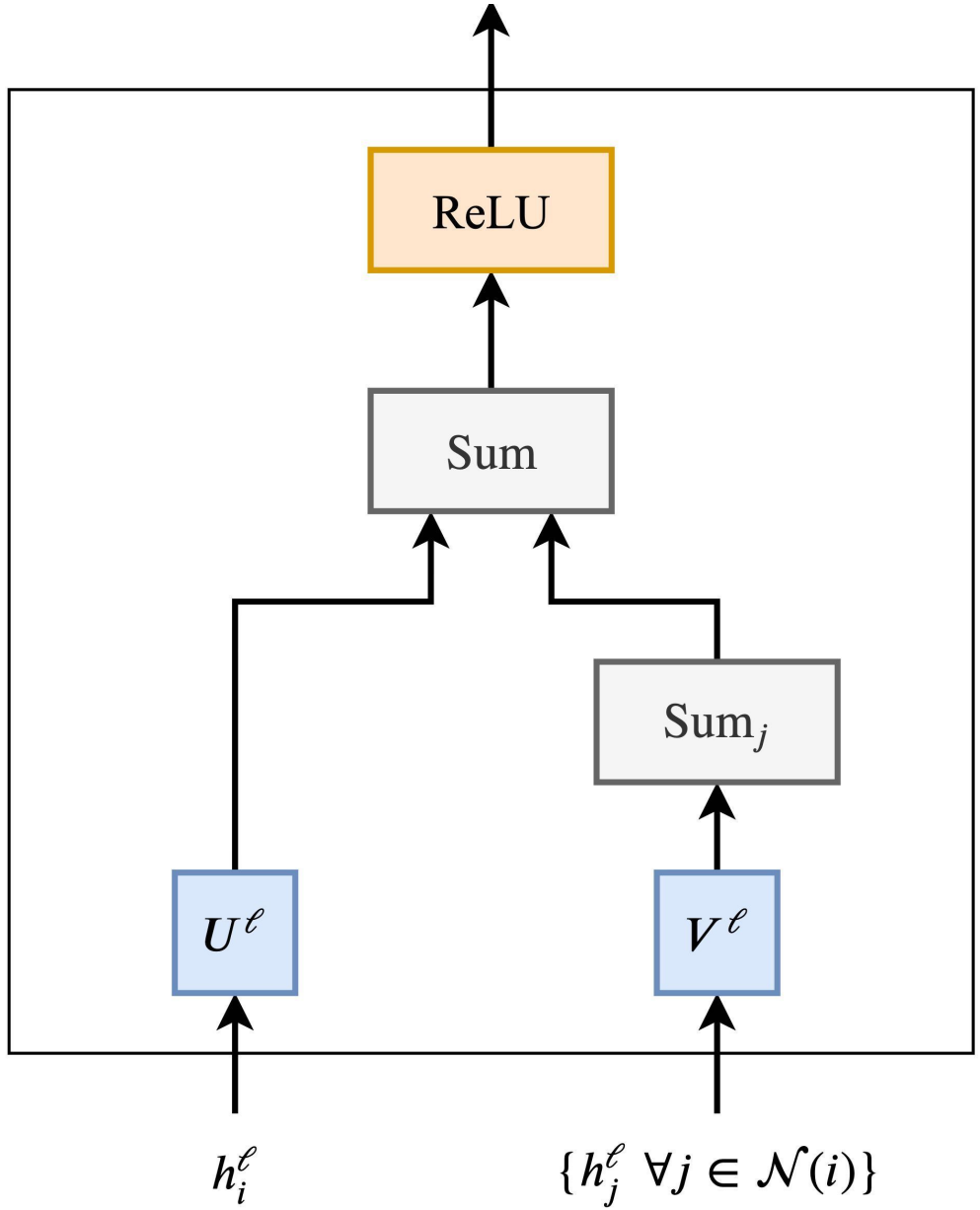


- In their most basic form, GNNs update the hidden features $\mathbf{h}$ of node $\mathbf{i}$ (for example, 😆) at layer ℓ via a non-linear transformation of the node's own features $\mathbf{h}_i^{\ell}$ added to the aggregation of features $\mathbf{h}_j^{\ell}$ from each neighbouring node $\mathbf{j} \in \square(\mathbf{i})$:

$$\mathbf{h}_i^{\ell+1} = \sigma\left(\mathbf{U}^{\ell}\mathbf{h}_i^{\ell} + \sum_{\mathbf{j}\in\square(\mathbf{i})}\left(\mathbf{V}^{\ell}\mathbf{h}_j^{\ell}\right)\right)$$

  - where $\mathbf{U}^{\ell}, \mathbf{V}^{\ell}$ are learnable weight matrices of the GNN layer and σ is a non-linear function such as ReLU. In the example, (😆) {😘 , 😎 , 😜 , 🤩 }.
- The summation over the neighbourhood nodes $\mathbf{j} \in \square(\mathbf{i})$ can be replaced by other input sizeinvariant aggregation functions such as simple mean/max or something more powerful, such as a weighted sum via an attention mechanism.
- Does that sound familiar? Maybe a pipeline will help make the connection (figure source):

Back to Top

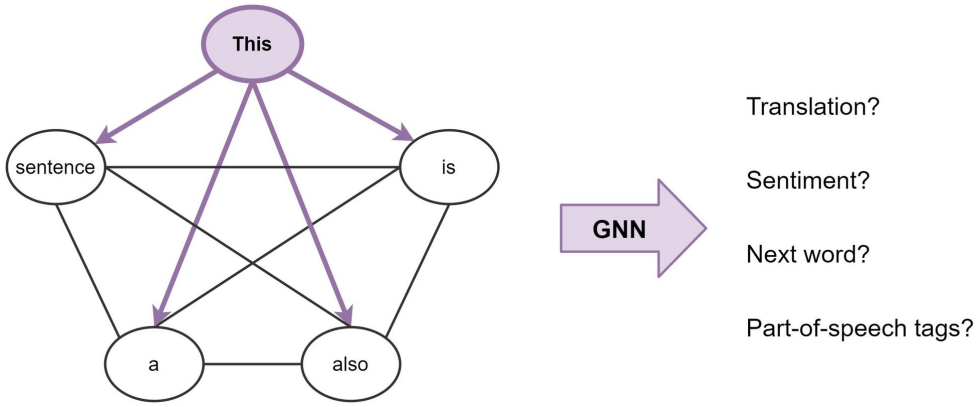

ReLU
Sum
Sum_j
U^ℓ
V^ℓ
h_i^ℓ
$\{h_j^\ell \ \forall j \in \mathcal{N}(i)\}$

Back to Top

Transformers are thus **a special case of GNNs** – they are just GNNs with multi-head attention.

Sentences are Fully-connected Word Graphs

- To make the connection more explicit, consider a sentence as a fully-connected graph, where each word is connected to every other word. Now, we can use a GNN to build features for each node (word) in the graph (sentence), which we can then perform NLP tasks with as shown in the figure (source) below.



- Broadly, this is what Transformers are doing: they are GNNs with multi-head attention as the neighbourhood aggregation function. Whereas standard GNNs aggregate features from their local neighbourhood nodes $j \in \square(i)$, Transformers for NLP treat the entire sentence $\square$ as the local neighbourhood, aggregating features from each word $j \in \square$ at each layer.
- Importantly, various problem-specific tricks—such as position encodings, causal/masked aggregation, learning rate schedules and extensive pre-training—are essential for the success of Transformers but seldom seem in the GNN community. At the same time, looking at Transformers from a GNN perspe could inspire us to get rid of a lot of the bells and whistles in the architecture.

learning, and graph networks by Battaglia et al. (2018) from DeepMind/Google, MIT and the University of Edinburgh offers a great overview of the relational inductive biases of various neural net architectures, summarized in the table below from the paper. Each neural net architecture exhibits varying degrees of relational inductive biases. Transformers fall somewhere between RNNs and GNNs in the table below (source).

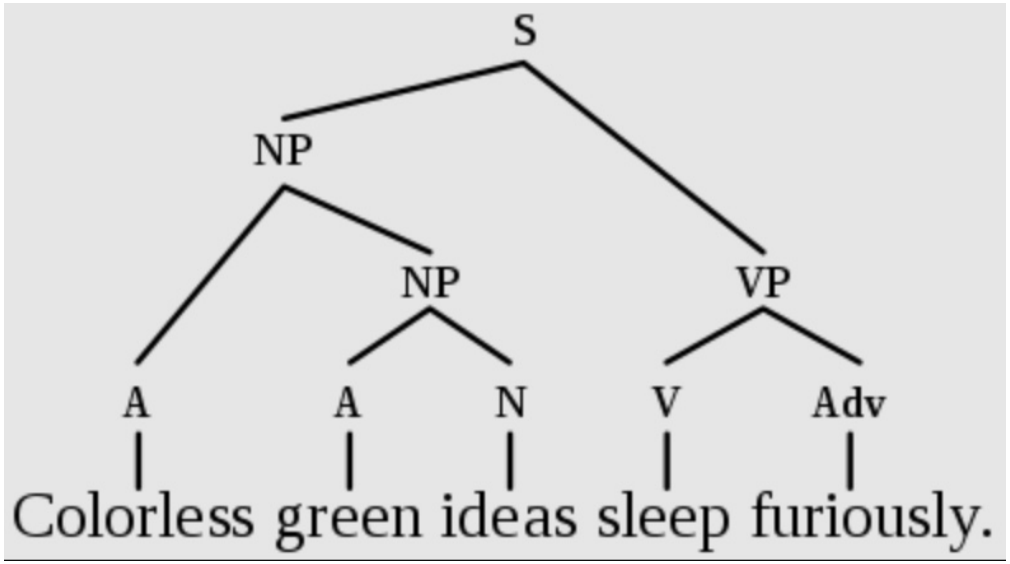
Component	Entities	Relations	Rel. inductive bias	Invariance
Fully connected	Units	All-to-all	Weak	-
Convolutional	Grid elements	Local	Locality	Spatial translation
Recurrent	Timesteps	Sequential	Sequentiality	Time translation
Graph network	Nodes	Edges	Arbitrary	Node, edge permutations

- YouTube Video from UofT CSC2547: Relational inductive biases, deep learning, and graph networks; Slides by KAIST on inductive biases, graph neural networks, attention and relational inference

Lessons Learned

Transformers: Merging the Worlds of Linguistic Theory and Statistical NLP Using Fully Connected Graphs

- Now that we've established a connection between Transformers and GNNs, let's throw some ideas around. For one, are fully-connected graphs the best input format for NLP?
- Before statistical NLP and ML, linguists like Noam Chomsky focused on developing formal theories of linguistic structure, such as syntax trees/graphs. Tree LSTMs already tried this, but ma

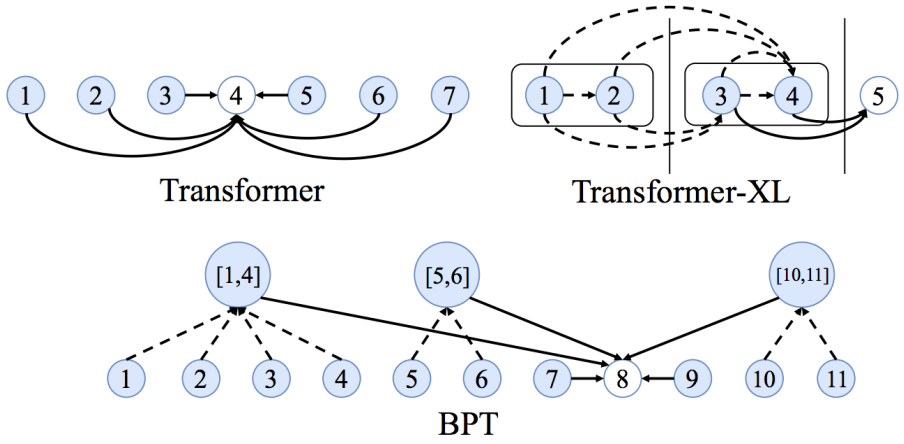


Long Term Dependencies

- Another issue with fully-connected graphs is that they make learning very long-term dependencies between words difficult. This is simply due to how the number of edges in the graph scales quadratically with the number of nodes, i.e., in an $\mathbf{n}$ word sentence, a Transformer/GNN would be doing computations over $\mathbf{n}^2$ pairs of words. Things get out of hand for very large $\mathbf{n}$.
- The NLP community's perspective on the long sequences and dependencies problem is interesting: making the attention mechanism sparse or adaptive in terms of input size, adding recurrence or compression into each layer, and using Locality Sensitive Hashing for efficient attention are all promising new ideas for better transformers. See Maddison May's excellent survey on long-term contex Transformers for more details.

Back to Top

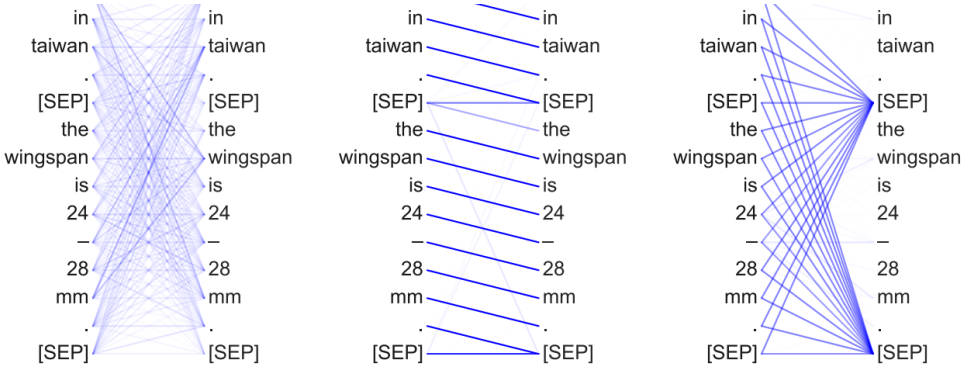
The following figure from Ye et al. (2019) shows binary partitioning for sentence graph sparsification.



Are Transformers Learning Neural Syntax?

- There have been several interesting papers from the NLP community on what Transformers might be learning. The basic premise is that performing attention on all word pairs in a sentence – with the purpose of identifying which pairs are the most interesting – enables Transformers to learn something like a **task-specific syntax**.
- Different heads in the multi-head attention might also be 'looking' at different syntactic properties, as shown in the figure (source) below.

Back to Top



Why Multiple Heads of Attention? Why Attention?

- The optimization view of multiple attention heads is that they **improve learning** and help overcome **bad random initializations**. For instance, Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned and it's accompanying post by Viota (2019) and Are Sixteen Heads Really Better than One? by Michel et al. showed that Transformer heads can be 'pruned' or removed after training without significant performance impact.

Benefits of Transformers Compared to RNNs/GRUs/LSTMs

- The Transformer can learn longer-range dependencies than RNNs and its variants such as GRUs and LSTMs.
- The biggest benefit, however, comes from how the Transformer lends itself to parallelization. Unlike an RNN which processes a word at each time step, a key property of the Transformer is that the word at each position flows through its own path in the encoder. There are dependencies between these paths in self-attention layer (since the self-attention layer computes how important each other word in the input

however, not of great importance for the decoder since it generates one word at a time and thus does not utilize parallel word paths.

What Would We Like to Fix about the Transformer? / Drawbacks of Transformers

- The runtime of Transformer architecture is quadratic in the length of the input sequence, which means it can be slow when processing long documents or taking characters as inputs. In other words, computing all pairs of interactions during self-attention means our computation grows quadratically with the sequence length, i.e., $O(T^2d)$, where T is the sequence length, and d is the dimensionality. Note that for recurrent models, it only grew linearly!
  - Say, $d = 1000$. So, for a single (shortish) sentence, $T \leq 30 \Rightarrow T^2 \leq 900 \Rightarrow T^2d \approx 900K$. Note that in practice, we set a bound such as $T = 512$. Imagine working on long documents with $T \geq 10,000$!?
- Wouldn't it be nice for Transformers if we didn't have to compute pair-wise interactions between each word pair in the sentence? Recent studies such as:
  - Synthesizer: Rethinking Self-Attention in Transformer Models
  - Linformer: Self-Attention with Linear Complexity
  - Rethinking Attention with Performers
  - Big Bird: Transformers for Longer Sequences
  - … show that decent performance levels can be achieved without computing interactions between all word-pairs (such as by approximating pair-wise attention).
- Compared to CNNs, the sample complexity (i.e., data appetite) of transformers is obscenely high. CNNs are still sample efficient, which makes them great candidates for low-resource tasks. This is especially true for image/video generation tasks where an exceptionally large amount of data is needed, even for (architectures (and thus implies that Transformer architectures would have a ridiculously high data

have their usecases.

- The runtime of the Transformer architecture is quadratic in the length of the input sequence. Computing attention over all word-pairs requires the number of edges in the graph to scale quadratically with the number of nodes, i.e., in an $\mathbf{n}$ word sentence, a Transformer would be doing computations over $\mathbf{n}^2$ pairs of words. This implies a large parameter count (implying high memory footprint) and thereby high computational complexity.
- High compute requirements has a negative impact on power and battery life requirements, especially for portable device targets.
- Overall, a transformer requires higher computational power, more data, power/battery life, and memory footprint, for it to offer better performance (in terms of say, accuracy) compared to its conventional competitors.

Why is Training Transformers so Hard?

- Reading new Transformer papers makes me feel that training these models requires something akin to black magic when determining the best learning rate schedule, warmup strategy and decay settings. This could simply be because the models are so huge and the NLP tasks studied are so challenging.
- But recent results suggest that it could also be due to the specific permutation of normalization and residual connections within the architecture.

The Road Ahead for Transformers

- In the field of NLP, Transformers have already established themselves as the numero uno architectural choice or the de facto standard for a plethora of NLP tasks.

Back to Top

say convolutions) will be the leading architectures of choice in the near future, especially if functional metrics (such as accuracy) are the sole optimization metrics. However, along other axes such as data, computational complexity, power/battery life, and memory footprint, transformers are currently not the best choice – which the above section on What Would We Like to Fix about the Transformer? / Drawbacks of Transformers expands on.

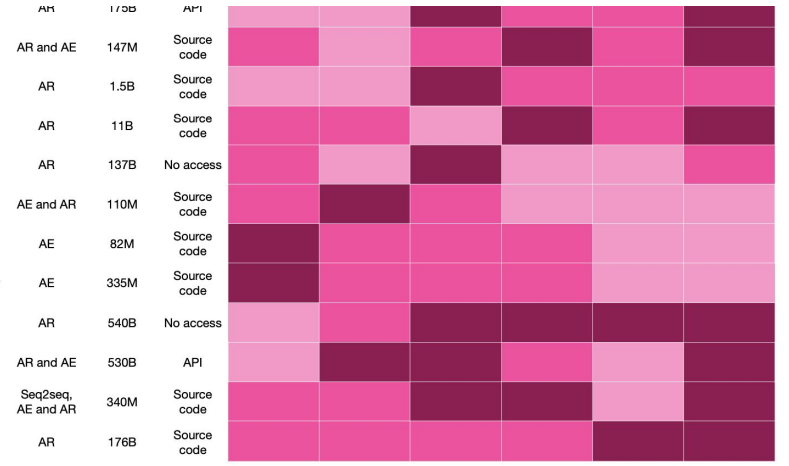
- Could Transformers benefit from ditching attention, altogether? Yann Dauphin and collaborators' recent work suggests an alternative ConvNet architecture. Transformers, too, might ultimately be doing something similar to ConvNets!

Choosing the Right Language Model for Your NLP Use-case: Key Takeaways

- Some key takeaways for LLM selection and deployment:
    1. When evaluating potential models, be clear about where you are in your AI journey:
        - In the beginning, it might be a good idea to experiment with LLMs deployed via cloud APIs.
        - Once you have found product-market fit, consider hosting and maintaining your model on your side to have more control and further sharpen model performance to your application.
    2. To align with your downstream task, your AI team should create a short list of models based on the following criteria:
        - Benchmarking results in the academic literature, with a focus on your downstream task.
        - Alignment between the pre-training objective and downstream task: consider auto-encoding for NLU and autoregression for NLG. The figure below shows the best LLMs depending on the NLP use-case (image source):

Back to Top

GPT-3	Parameter size	AR	175B	API
BART	Novel combination of pre-training objectives	AR and AE	147M	Source code
GPT-2	Parameter size	AR	1.5B	Source code
T5	Multi-task transfer learning	AR	11B	Source code
LaMDA	Dialogue; safety and factual grounding	AR	137B	No access
XLNet	Joint AE and AR	AE and AR	110M	Source code
DistilBERT	Reduced model size via knowledge distillation	AE	82M	Source code
ELECTRA	Computational efficiency	AE	335M	Source code
PaLM	Training infrastructure	AR	540B	No access
MT-NLG	Training infrastructure	AR and AE	530B	API
UniLM	Optimised both for NLU and NLG	Seq2seq, AE and AR	340M	Source code
BLOOM	Multilingual (46 languages)	AR	176B	Source code



3. The short-listed models should be then tested against your real-world task and dataset to get a first feeling for the performance.
4. In most cases, you are likely to achieve better quality with dedicated fine-tuning. However, consider few/zero-shot learning if you don't have the internal tech skills or budget for fine-tuning, or if you need to cover a large number of tasks.
5. LLM innovations and trends are short-lived. When using language models, keep an eye on their lifecycle and the overall activity in the LLM landscape and watch out for opportunities to step up your game.

Transformers Learning Recipe

- Transformers have accelerated the development of new techniques and models for natural langu processing (NLP) tasks. While it has mostly been used for NLP tasks, it is now seeing heavy adoption in

Back to Top

in learning about the world of Transformers.
- To dive deep into the Transformer architecture from an NLP perspective, here's a few links to better understand and implement transformer models from scratch.

HuggingFace Encoder-Decoder Models

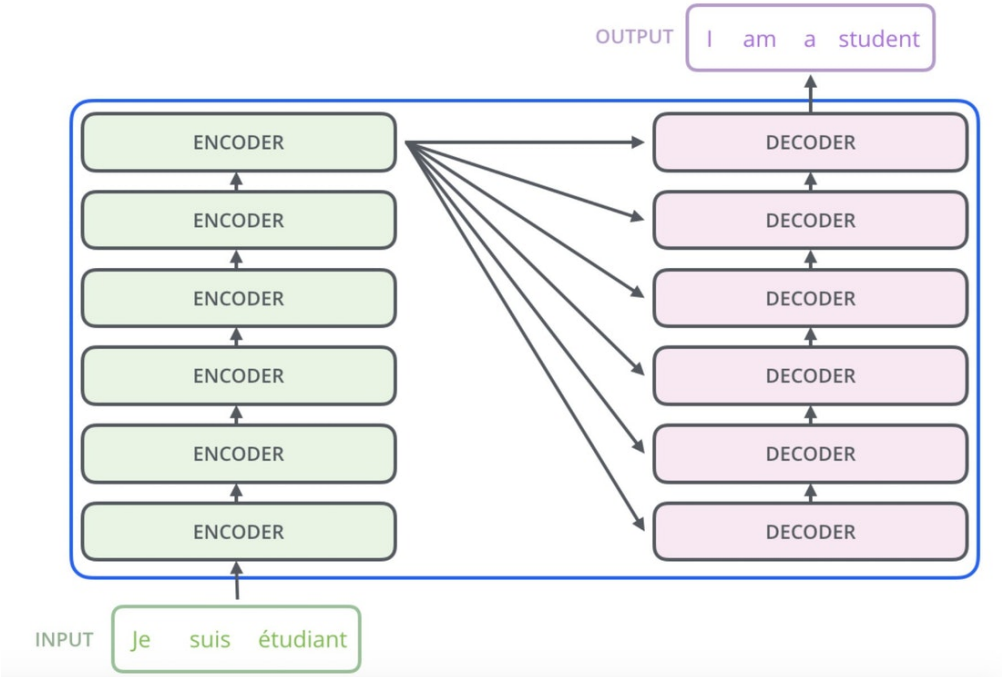
- With the HuggingFace Encoder-Decoder class, you no longer need to stick to pre-built encoder-decoder models like BART or T5, but can instead build your own Encoder-Decoder architecture by doing a mix-and-match with the encoder and decoder model of your choice (similar to stacking legos!), say BERT-GPT2. This is called "warm-starting" encoder-decoder models. Read more here: HuggingFace: Leveraging Pre-trained Language Model Checkpoints for Encoder-Decoder Models.
- You could build your own multimodal encoder-decoder architectures by mixing and matching encoders and decoders. For example:
  - Image captioning: ViT/DEiT/BEiT + GPTx
  - OCR: ViT/DEiT/BEiT + xBERT
  - Image-to-Text (CLIP): ViT/DEiT/BEiT + xBERT
  - Speech-to-Text: Wav2Vec2 Encoder + GPTx
  - Text-to-Image (DALL-E): xBERT + DALL-E
  - Text-to-Speech: xBERT + speech decoder
  - Text-to-Image: xBERT + image decoder
- As an example, refer TrOCR: Transformer-based Optical Character Recognition with Pre-trained Models and Leveraging Pre-trained Checkpoints for Sequence Generation Tasks.

High-level Introduction

Back to Top

The Illustrated Transformer

- Jay Alammar's illustrated explanations are exceptional. Once you get that high-level understanding of transformers, going through The Illustrated Transformer is recommend for its detailed and illustrated explanation of transformers:



Technical Summary

Back to Top

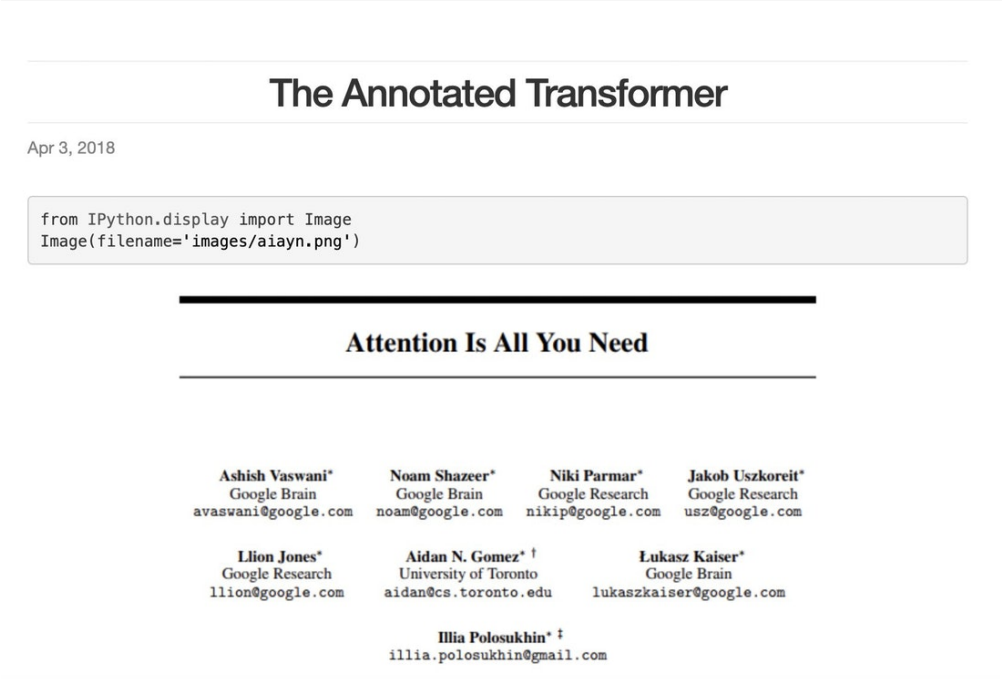
Notations

Symbol	Meaning
d	The model size / hidden state dimension / positional encoding size.
h	The number of heads in multi-head attention layer.
L	The segment length of input sequence.
$\mathbf{X} \in \mathbb{R}^{L \times d}$	The input sequence where each element has been mapped into an embedding vector of shape d, same as the model size.
$\mathbf{W}^k \in \mathbb{R}^{d \times d_k}$	The key weight matrix.
$\mathbf{W}^q \in \mathbb{R}^{d \times d_k}$	The query weight matrix.
$\mathbf{W}^v \in \mathbb{R}^{d \times d_v}$	The value weight matrix. Often we have $d_k = d_v = d$.
$\mathbf{W}_i^k, \mathbf{W}_i^q \in \mathbb{R}^{d \times d_k/h}; \mathbf{W}_i^v \in \mathbb{R}^{d \times d_v/h}$	The weight matrices per head.
$\mathbf{W}^o \in \mathbb{R}^{d_v \times d}$	The output weight matrix.
$\mathbf{Q} = \mathbf{X}\mathbf{W}^q \in \mathbb{R}^{L \times d_k}$	The query embedding inputs.
$\mathbf{K} = \mathbf{X}\mathbf{W}^k \in \mathbb{R}^{L \times d_k}$	The key embedding inputs.
$\mathbf{V} = \mathbf{X}\mathbf{W}^v \in \mathbb{R}^{L \times d_v}$	The value embedding inputs.
S_i	A collection of key positions for the i-th query $\mathbf{q}_i$ to attend to.
$\mathbf{A} \in \mathbb{R}^{L \times L}$	The self-attention matrix between a input sequence of lenght L and itself. $\mathbf{A} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top / \sqrt{d_k})$.
$a_{ij} \in \mathbf{A}$	The scalar attention score between query $\mathbf{q}_i$ and key $\mathbf{k}_j$.
$\mathbf{P} \in \mathbb{R}^{L \times d}$	position encoding matrix, where the i-th row $\mathbf{p}_i$ is the positional encoding for input $\mathbf{x}_i$.

Implementation

- Once you've absorbed the theory, implementing algorithms from scratch is a great way to test your knowledge and understanding of the subject matter.
- For implementing transformers in PyTorch, The Annotated Transformer offers a great tutorial.
- For implementing transformers in TensorFlow, Transformer model for language understanding offe great tutorial.

Back to Top



The Annotated Transformer

Apr 3, 2018

```
from IPython.display import Image
Image(filename='images/aiayn.png')
```

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Attention is All You Need

- This paper by Vaswani et al. introduced the Transformer architecture. Read it after you have a high-level understanding and want to get into the details. Pay attention to other references in the paper for diving deep.

Back to Top

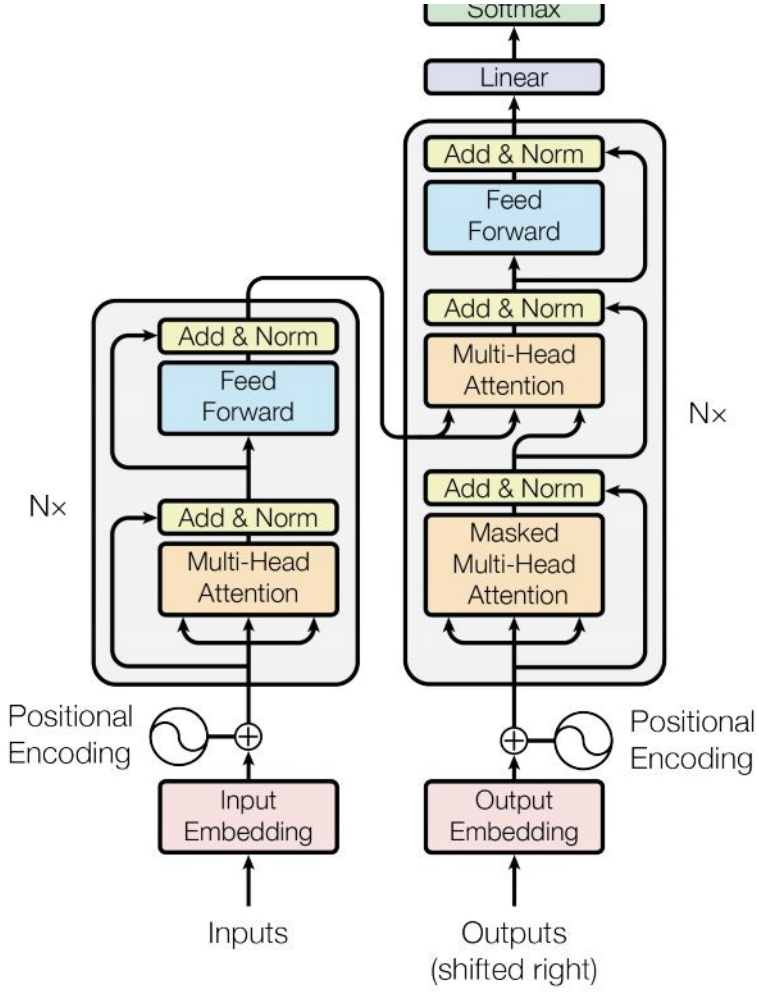
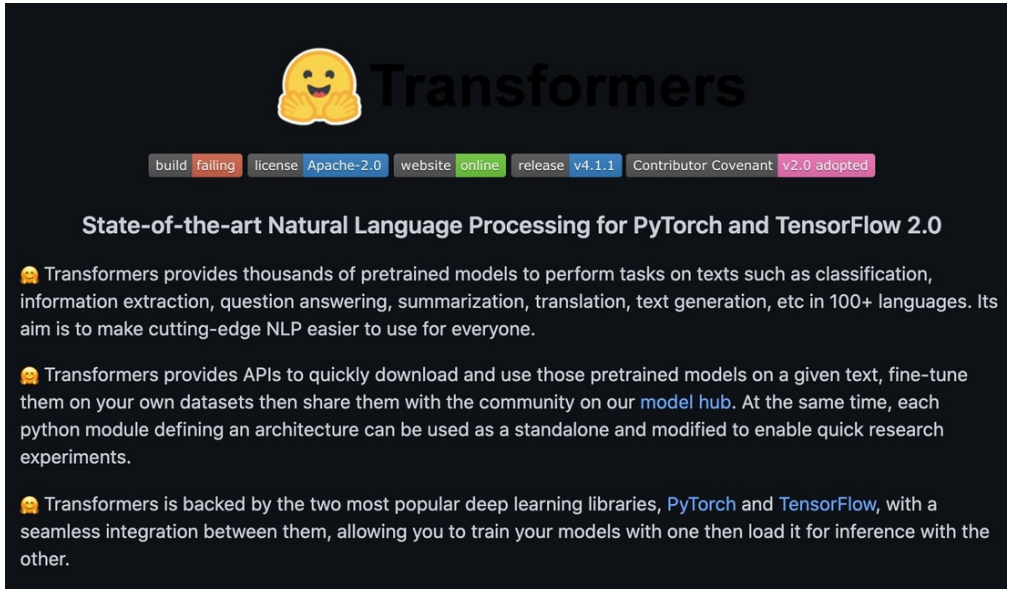


Figure 1: The Transformer - model architecture.

Applying Transformers

Back to Top

check that out as well.



Further Reading

- The Illustrated Transformer
- The Annotated Transformer
- Transformer models: an introduction and catalog
- Deep Learning for NLP Best Practices
- What is Teacher Forcing for Recurrent Neural Networks?
- What is Teacher Forcing?
- The Transformer Family

Back to Top

- Transformers from Scratch by Peter Bloem
- Positional encoding tutorial by Amirhossein Kazemnejad
- What is One Hot Encoding? Why and When Do You Have to Use it?
- Wikipedia: Dot product
- Wikipedia: Byte pair encoding
- Will Transformers Take Over Artificial Intelligence?
- Transformer Recipe
- The decoder part in a transformer model
- Why encoder input doesn't have a start token?
- What is the cost function of a transformer?
- Transformer models: an introduction and catalog
- Why does a transformer not use an activation function following the multi-head attention layer?
- CNN Explainer: Learn Convolutional Neural Network (CNN) in your browser!

Citation

If you found our work useful, please cite it as:

```
@article{Chadha2020DistilledTransformers,
  title   = {Transformers},
  author  = {Chadha, Aman},
  journal = {Distilled AI},
  year    = {2020},
  note    = {\url{https://aman.ai}}
}
```

Back to Top

www.amanchadha.com