Explanatory Notes for 6.3900

Shaunticlair Ruiz

Fall 2024

Contents

CHAPTER 8

Transformers

In this chapter, we want to focus on processing language. In particular:

Definition 1

Natural Language Processing (NLP) is a field of machine learning all about processing, understanding, and using **human language**.

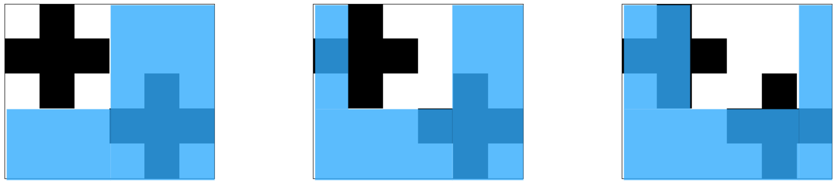
- **Example:** Chatbots, language translation, etc.

We'll start by considering a few candidate models for NLP, before moving to the state-of-the-art: **transformers**.

8.0.1 CNNs

In the previous chapter, we introduced the notion of a CNN:

- **Convolutional Neural Networks (CNNs)** view small regions of data, searching for patterns across the image.



In this example, we focus on a 3x3 segment of our data.

This kind of structure is useful for image processing: nearby pixels tend to be related to each other.

They might form a single line, or a corner, for example.

- By prioritizing "nearby" information, we can create models that easily find those **localized** patterns.
- We called this property **spatial locality**.

Concept 2

CNNs are designed to represent **locality**:

- In a CNN, **nearby** data is used to search for patterns.

This allows us to use smaller, **simpler** models:

- Rather than thinking about every possible connection between data, we only connect "nearby" data. Thus, we need **fewer** parameters.

8.0.2 The problem with locality

This presents one simple weakness, that we've ignored so far:

- If we focus on information that is **nearby**, we're missing out on information that's **far away**.
- We need a way to encode "distance" of information, that doesn't ignore the "distant" info.

Concept 3

If information is spread over **long distances**, our CNN model won't capture it.

- If a pattern is **too big** for our CNN filter, we'll have more trouble finding it.

This can become especially problematic for **language** processing.

Example: Consider the following sentence:

- **The sweater** that I found in the back of my old closet, which I hadn't opened since we moved into the house several years ago, **still fits me perfectly.**

Note that the beginning and the end of this sentence are linked as a single idea: "The sweater still fits me perfectly".

- But there's a **huge gap** between these phrases: it might be difficult to connect information over such a wide gap, while **ignoring** what's in-between.
- This also comes up in longer passages: in a paragraph, the first sentence might create **context** for the last sentence.

Concept 4

In language, words can be **far apart**, while still providing important **context** for the meaning of the text.

- Thus, language processing is difficult for models which focus too much on **locality**.

8.0.3 RNNs

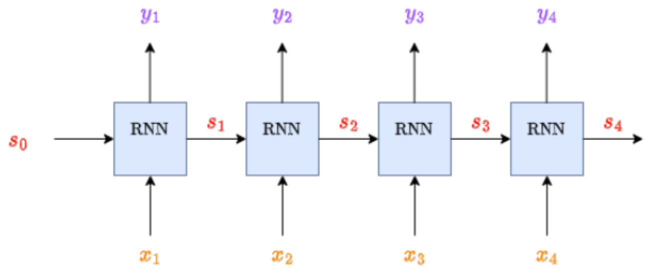
One useful observation might be that language tends to be *sequential*: words come in a very particular order.

Concept 5

In **image processing**, we see many pixels at the same time: the whole image is processed in **parallel**.

In **language processing**, we hear/read words one-by-one, in order: the data has a **sequential** structure.

Recurrent Neural Networks (RNNs) are, thus, a **sequential** model, designed for processing language.



Each x_t is one word in our sentence: we process the text, one word at a time. After every word, we update our memory ("state" s_t). y_t is our output at time t.

By storing information about previous words (using a **state**), our model can "read" each word **in order**, while still remembering earlier parts of the text.

- While a CNN can only observe k consecutive pixels/words in a row, our RNN might be able to contain *some* information about words that are much further back in time.

How well does this work? RNNs have seen success in the past, but it struggles with **forgetting**: our RNN can only store so much information about words it's seen before.

- As a passage gets longer, our RNN is only paying attention to words it's seen **recently**.

Moreover, our RNN doesn't have any way to **choose** which words to **prioritize**: each new word will have to replace some information about older words.

- So, our RNN naturally prioritizes the most **recent words**.
- But the most recent word isn't always the most important one, as we saw above (in the sweater example)!

The more recent words haven't been replaced yet.

Concept 6

RNNs (Recurrent Neural Networks) tend to struggle with longer bodies of text:

- The **longer** we run our RNN, the less it usually remembers about the **distant** past.

Moreover, it prioritizes recent words, even when more distant words may be **more important**.

In the end, RNNs have, in most language applications, been replaced by transformers: a different model for language processing.

However, some transformer models have begun using the concepts of LSTMs, an RNN variant. We won't cover this topic here.

8.0.4 Transformers

One clever way to think about this problem is to recognize that our goal is to decide which words are **related** to each other, whether they're nearby or far apart.

- In other words, which words should we pay **attention** to, in order to understand the text we're reading?

This is exactly the problem that **transformer models** solve, using the appropriately named **attention mechanism**.

Clarification 7

In this chapter, we'll use **transformers** to **process language**, using the mechanism of **attention**.

- But the same tools can be applied to **many other problems**: image and audio processing, robotics, etc.

We'll develop this model in several steps:

- First (11.1), we'll convert words into vectors. One-hot encoding is too simple, so we'll use a different approach: **vector embeddings**.
- Next (11.2), we'll figure out which words in a passage are **relevant**(or connected) to each other, using a clever system called **attention**.
- Finally (11.3), we'll put together these ideas to create a complete model, known as a **transformer**.

8.1 Vector embeddings and tokens

8.1.1 One-hot encoding isn't enough

First, we want to turn words into something computable, like a **vector**.

It's difficult to try to do math on the word "cheddar". It's not numerical.

The simplest approach would be **one-hot encoding**.

- **Example:** Suppose that we want to classify **furniture** as table, bed, couch, or chair.

$$\begin{bmatrix} \text{table} \\ \text{bed} \\ \text{couch} \\ \text{chair} \end{bmatrix} \tag{8.1}$$

- For each class:

$$v_{\mathtt{chair}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad v_{\mathtt{table}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad v_{\mathtt{couch}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad v_{\mathtt{bed}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{8.2}$$

This approach is simple, but often, it's *too* simple.

Concept 8

One-hot encoding loses a lot of information about the objects it's representing.

- It's hard to say which words are "**similar**" to each other, for example.

Example: You probably associate the word "sugar" with "sweet", and "salt" with "savory".

- But, if you use one-hot encoding, all of these words are "equally different".

You could **shuffle** the rows of one-hot vectors, and represent the same information.

So, we can't use the order of 1's and 0's to determine "closeness": the order can be **freely changed**.

$$v_{\mathtt{salt}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad v_{\mathtt{savory}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad v_{\mathtt{sugar}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad v_{\mathtt{sweet}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{8.3}$$

In order to incorporate this information, we'll need a better way to represent words as vectors.

8.1.2 Word Embeddings: Similarity between words

Our new approach will convert each word w into a **vector** v_w of **length** d.

Unlike one-hot encoding, we don't require that d equals the size of our vocabulary.

$$w \longrightarrow v_w \qquad\qquad v_w \in \mathbb{R}^d \tag{8.4}$$

How do we want to convert words into vectors? Above, we mentioned that one-hot doesn't tell us how **similar** two words are.

Clarification 9

There are many ways for words to be **similar**: similar word length, similar choice of letters, etc.

But in our case, we're interested in **semantics**: the **meanings** of the words. We want to know which words have similar meanings.

- **Example:** We don't consider "sugar" and "sweet" to be similar because they both start with "s".
  - They're similar because of **meaning**: sugar tastes sweet. Sweet strawberries contain sugar.

Concept 10

We often want our **word embeddings** v_w to tell us which words are **semantically similar** to each other: which words have similar **meanings**.

$$v_a \text{ and } v_b \text{ are } \textbf{similar vectors} \quad \Longleftrightarrow \quad a \text{ and } b \text{ are } \textbf{semantically similar words}$$

Our goal is to make this statement true. But we have a problem: these are *concepts*, rather than computable *numbers*.

- So, we'll have to turn each side into something computable.

8.1.3 Vector Similarity: Dot Products

First, we'll handle the left side: how do we know if vectors are **similar**?

- We've come across this problem multiple times, and we'll solve it the same way as always: using the **dot product**.

Concept 11

Review from the Classification chapter

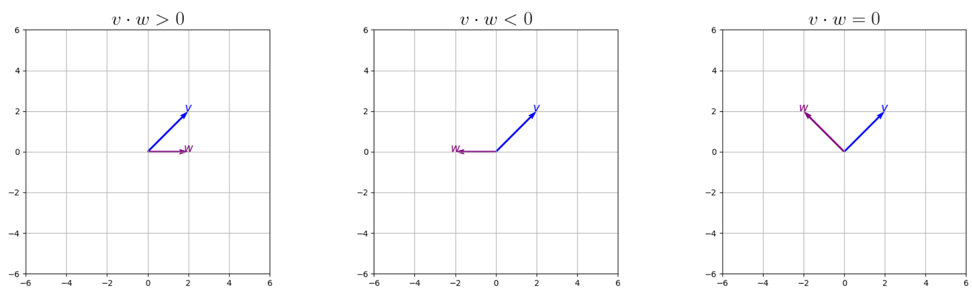
You can use the **dot product** between vectors u and v, **normalized by their magnitudes**, to measure their "**cosine similarity**".

$$S_C(u, v) = \frac{u \cdot v}{|u| \cdot |v|}$$

If two vectors are more **similar**, they have a **larger** normalized dot product.

- This function ranges from -1 (opposite vectors) to +1 (identical vectors). Perpendicular vectors receive a 0.

We call it "cosine similarity", because this is equal to the cosine of the angle α between u and v.



We can see here what we mean by "similar" or "dissimilar".

Clarification 12

You can use $S_C(u, v)$ to measure the **similarity** between two vectors, ignoring magnitude.

But for simplicity, we'll skip the **normalizing** step, and just take the **dot product**:

$$S_D(u, v) = u \cdot v = u^\top v$$

We're getting closer to a computable form:

$$\overbrace{(v_a \cdot v_b) \text{ is } \textbf{large}}^{\text{Similar vectors}} \iff a \text{ and } b \text{ are } \textbf{semantically similar words} \tag{8.5}$$

8.1.4 Word2vec

Next, we should get into the math of how to determine which words are likely to be **similar**.

- But this is a bit cumbersome, and isn't really necessary for understanding transformers.

The short version: we expect words which frequently appear in the same contexts, to be similar.

So, we relegate this mathematical labor to Appendix D, where we'll get into the details of **skipgram** and **word2vec**.

Definition 13

We can think of **word2vec** as a system for **word embeddings** where words which have **similar meanings**, have **similar vector embeddings**.

- Most commonly, we measure "vector similarity" with the **dot product**.

Instead, we'll skip a couple steps, and look at things from a high level.

8.1.5 Probability

Our goal is to be able to numerically talk about the "similarity" or "relatedness" of words.

Above, we represent this with a **dot product**: this gives us a **real number** $u \cdot v \in \mathbb{R}$.

- This number isn't very **meaningful**, though. For example, what does a "similarity of 37" even mean? Is that high? Is that low?

Generally, our best bet for understanding a number like this is to **compare** it to other numbers.

You know that someone who is 6'5" is really tall, because you know how tall other people tend to be.

So, let's think about the **relative** similarity of words: if we have two words, w_1 and w_2, which one is v **more related** to?

We'll focus on one simple tool for comparison: **probability**.

- One way to think about it is, "how likely is w_i to be the **most relevant** word to v, in any given context?"
- Alternatively, "how **confident** are we that these words are actually closely related, compared to others?"

In skipgram, our probability comes from asking, "how likely is w_i to show up in the same context?"

The **higher** the probability of word w_1, the **lower** the probability of word w_2, and vice versa.

We'll represent this "relatedness of word w_i to word v" as probability $P(w_i \mid v)$.

Concept 14

One way to describe the **relatedness** of different words w_i is with a **probability** $P(w_i \mid v)$.

This has a few advantages:

- A probability is easier to **interpret** than a real number.
- We can directly **compare** different words.
- We can systematically **convert** our dot product to a probability.

This "probability" interpretation is a bit better justified if you read the skipgram section.

How do we turn a **real number** $v_a \cdot v_b$ into a **probability** $P(b \mid a)$?

- For a probability, we need to compare b to every other word: this is a **multi-class problem**, using the **softmax function**.

$$\text{Softmax}(z_k) = \frac{e^{z_k}}{\sum_i e^{z_i}} \tag{8.6}$$

Let's review the concept behind "softmax":

Definition 15

Suppose that we have n **possible words** (n "classes"), and we want to figure out which one is **correct**.

The k^{th} **class** has a score, z_k, used to compute probability.

- The bigger z_k is, the **more likely** k is to be the **correct class**.

To keep it **positive**, z_k is converted to e^{z_k}: each e^{z_i} competes to see which class is more likely.

- To create a probability, we **compare** the score of class k to all of our other classes, using **softmax**.

$$\overbrace{e^{z_k}}^{\text{Class } k} \quad \text{vs} \quad \overbrace{\sum_i e^{z_i}}^{\text{All classes}} \qquad \Longrightarrow \qquad \text{Softmax}(z_k) = \frac{e^{z_k}}{\sum_i e^{z_i}}$$

We repeat this process for every possible word i, to get all of our predictions.

What is our "score" z_k? We could use $(v_a \cdot v_b)$:

- The higher $(v_a \cdot v_b)$ is, the more **similar/related** we expect a and b to be.
- The same is true for z_k: if z_k is larger, then our **probability** goes up.

So, we can use our dot product as a "score" z_k:

$$z_b = v_a \cdot v_b \tag{8.7}$$

We'll plug this into our probability equation:

Key Equation 16

The **more similar** (bigger dot product) a and b are, the **more likely** we predict to find them together.

- We use a **softmax** to compute this probability for each possible word b.

$$\mathbf{P}\left\{b \;\middle|\; a\right\} = \frac{e^{v_a \cdot v_b}}{\sum_i e^{v_a \cdot v_i}}$$

Or, in alternate notation:

$$\mathbf{P}\left\{b \;\middle|\; a\right\} = \frac{\exp\left(v_a \cdot v_b\right)}{\sum_i \exp\left(v_a \cdot v_i\right)}$$

This kind of interpretation makes our word embeddings a bit more **useful**.

- Later, we'll find that it's the most important part of making **transformers** work!

8.1.6 "Adding" words together

Our word2vec system works under the hope that these vector embeddings can accurately represent the **meanings** of words.

- In practice, this assumption works **surprisingly well**, for being so simple.

One example is the idea of "**adding**" words together. Normally, it's hard to say how to "add words" together, but we *do* know how to add **vectors**.

Consider the following example:

$$v_{\text{king}} - v_{\text{man}} + v_{\text{woman}} \quad \approx \quad v_{\text{queen}} \tag{8.8}$$

This sort of reasoning makes sense to most english speakers:

$$\overbrace{v_{\text{king}} - v_{\text{man}}}^{\text{ruler}} \; + \; v_{\text{woman}} \quad \approx \quad \overbrace{v_{\text{queen}}}^{\text{female ruler}} \tag{8.9}$$

We can repeat this process for other words:

Paris is the capital of France, and Rome is the capital of Italy.

$$v_{\text{paris}} - v_{\text{france}} + v_{\text{italy}} \quad \approx \quad v_{\text{rome}} \tag{8.10}$$

Concept 17

Transforming a word into a **vector** allows you to use vector operations, like **addition** and **subtraction**.

- The result can be surprisingly **meaningful**, for some word combinations.

This approach doesn't always work, but the fact that it works *sometimes* suggests that our vectors might capture real information about the "**meanings**" of words.

That said, this approach is often an over-simplification:

Concept 18

Reducing a word to a **single vector** can cause problems, because the same word might change its meaning, based on **context**.

- **Example:** For example, the word "bank" has a very different meaning when you compare "bank account" to "river bank".

This idea of "context" is what we hope to solve next.

8.1.7 Tokenization

One clarification, before we move on: so far, we've talked about predicting whole **words**, because it's easy to work with.

- But often, for language analysis, we break up words into parts, called **tokens**.
- These are the objects we study/predict, rather than whole words.

Definition 19

Rather than using/predicting entire **words**, we use **small parts of words**, called **tokens**.

- A "token" is the **smallest unit** in our language model.

- **Example:** You might break up the word "eating" into "eat" and "ing": both are meaningful, by themselves.
- This process of turning words into tokens is called **tokenization**.

While "tokens" are used more often than "words", words often make for better examples, so we'll keep using them through the rest of this chapter.

Clarification 20

We'll continue using words (instead of tokens) for examples, when it's convenient.

8.2 Attention

Our **word embedding** technique has given us a basic way to talk about which words are "**related**".

- We can even use this to learn some about the "**meanings**" of words.

But there's some work to be done:

> **Concept 21**
> Our **word embedding** technique has two major problems, for representing the **meanings** of words:
>
> - There's a lot of information we're **missing**: **similarity** to other words isn't enough. We'll need a vector to represent that information.
> - The meaning of a word is **contextual**: the sentence you put a word in, will affect its meaning.

It may not look like it, but our **word embedding** technique has already given us the basic tools we need to solve these problems.

Here's the basic idea, for how we handle each problem:

> **Concept 22**
> We'll create a system that solves both of these problems, using **3 word embeddings**: v, k, and q.
>
> - We'll **embed information** about each word in a **value vector** v.
> - When finding the **meaning** of a word, we'll calculate **context** from nearby words.
>   - We'll use **word similarity** to figure out which parts of the context are **most important**.
>   - For this purpose, word will need **two embeddings**: a **key vector** k, and a **query vector** q.
>
> The result is a powerful model called the **attention mechanism**.

This description is over-simplified, which is why we'll need to go into detail below.

8.2.1 The Attention Mechanism: queries, keys

Let's consider an **example**, to get used to the idea of k, v, and q.

Suppose we want a general idea of what "mexican" food is like. We'll need to consider lots of foods, and take an **average** of those we consider to be "mexican".

Admittedly, we're turning "mexican-ness" into a number, which can be a bit strange.

It may help to think this way: "if someone is talking about mexican food, how often are they talking about this food?"

This problem comes in three parts: let's consider the first two, "query" and "key".

- **Query** q: we're searching for "mexican" food. The word "mexican" is represented by a **query vector** q.
  - This is like our previous **word2vec** embedding: if two vectors are similar, then we expect them to have similar/related **meanings**.
  - So, we'll **compare** q to each food, to see which foods are 'close' to mexican.

Definition 23

The **query vector** q represents a word, that we're **comparing** to **several other words** ("keys").

- It answers the question, "what kinds of words are we **searching** for?"

Using word embeddings, we design q to be "meaningful": similar words, should have similar vectors.

- And we expect **similar words** to be **more relevant** to our query.

- **Key** k: Each food (apple, burrito, sushi...) has a **key vector** k, representing it.
  - A **word2vec**-style embedding, just like the **query**.
  - Combining k and q will tell us which foods are **'more' mexican**.

Definition 24

The **key vector** k represents a word, that we want to **compare** to **the query** q.

- It answers the question, "what kinds of searches does this word **match**"?

Because it's a word embedding, which encodes meaning, we expect that, if **k and q are similar**, then our key word is **more relevant** to our query.

Each embedding has a role: a **query** is used to search for relevant words, and a **key** is responding to that search, on behalf of one word.

Reminder that when we say "word", we're simplifying: we could talk about any kind of token.

Concept 25

Another way we could view keys vs. queries:

- **Query vector** q: asks, "how relevant are these words/tokens **to me**?"
- **Key vector** k: asks, "how relevant is my word **to the query**?"

Notice that we've made a perspective shift, in how we view word embeddings:

Concept 26

When we were developing word2vec, we wanted **similar vectors** to represent **semantically similar** words.

- But, in this case, we're less focused on "similarity", than **relevance**.

We look for **keys** that are the **most relevant** to our **query**.

These two ideas don't necessarily conflict, but they have somewhat different goals.

8.2.2 The Attention Mechanism: attention weights

How do we compute how similar k and q are? The same way as we did for word2vec: we use a **dot product**.

Key Equation 27

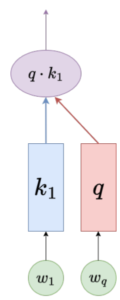
We can get a score for how **relevant** the word b is to word a, by taking the **dot product** between **b's key**, and **a's query**.

$$q_a \cdot k_b$$

We can also write this as matrix multiplication:

$$q \cdot k = q^T k$$

This gives us a "**score**": the higher $k \cdot q$ is, the more similar they are.



We convert w_1 and w_q into a key and query, respectively, before taking the dot product.

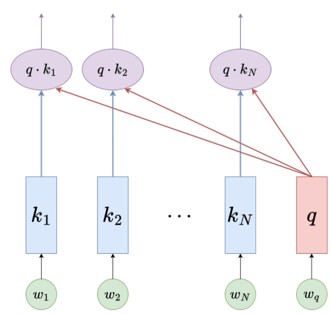
Notation 28

Note that k and q have to have the **same length**: they're both $(d_k \times 1)$ column vectors.

If their lengths don't mach, we can't take the dot product.

But we're not just considering one key word: we're considering **all of of them**.

- In our "mexican food" example, we need to check every food, to see which ones best fit the category.



We re-use our query q for every single dot product.

Notation 29

We have N distinct keys.

In the official notes, we use n instead of N. This doesn't affect any of our math.

How do we compare each of these keys?

- Once again, we'll reuse a tool from word2vec: **softmax**.

Key Equation 30

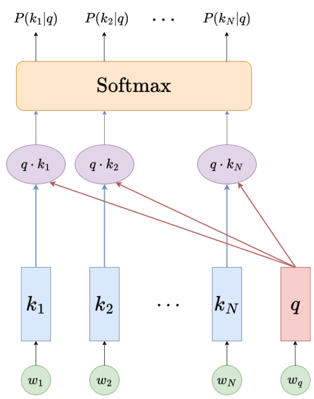
We can compute the relative **relevance** of a key k_j, by:

- **Comparing** each key k_i to q $(q \cdot k_i)$
- Use **softmax** to compute $p(k_j|q)$: given query q, how important is k_j?

$$\mathbf{P}\left\{k_j \;\middle|\; q\right\} \quad = \quad \frac{e^{q \cdot k_j}}{\sum_i e^{q \cdot k_i}}$$

$\mathbf{P}\left\{k_j \;\middle|\; q\right\}$ tells you, "how much **attention** should q pay to k_j"?

- Thus, we call $\mathbf{P}\left\{k_j \;\middle|\; q\right\}$ an **attention weight**.



Finally, we've converted each word into their "probability" of being relevant.

One notational thing: we can write this a bit more densely.

- So far, we've been computing $q^{\top}k_i$ for each k_i term **separately**.
- But, one benefit of matrix multiplication, is that we can **combine multiple operations** into one.

First, we'll **combine** all of our key vectors into a matrix K:

This matrix has shape $(N \times d_k)$: the transpose of what you might expect.

$$K = \begin{bmatrix} | & | & & | \\ k_1 & k_2 & \dots & k_N \\ | & | & & | \end{bmatrix}^{\top} \tag{8.11}$$

With that, we can compute all of our dot products **at the same time**:

This product has shape $(1 \times N)$.

$$q^\top K^\top = \begin{bmatrix} q \cdot k_1 \\ q \cdot k_2 \\ \vdots \\ q \cdot k_N \end{bmatrix}^\top \tag{8.12}$$

And we can combine all of these together into a softmax.

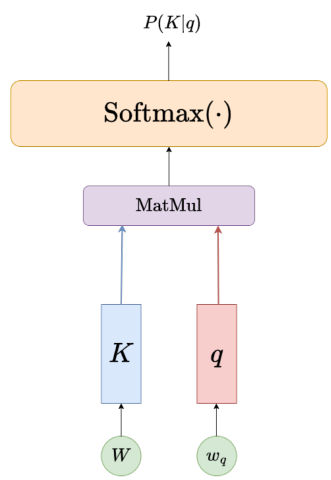
Key Equation 31

By combining all of our keys into a matrix K, we can compute all of our **attention weights at the same time**.

$$\mathbf{P}\left\{K \;\middle|\; q\right\} = \begin{bmatrix} \mathbf{P}(k_1 \mid q) \\ \mathbf{P}(k_2 \mid q) \\ \vdots \\ \mathbf{P}(k_N \mid q) \end{bmatrix}^\top = \text{softmax}\left(q^\top K^\top\right)$$

It has shape $(1 \times N)$.

Note that here, softmax creates a row vector.



Now, our diagram is visually simpler, though it reflects the same information. "MatMul" means "Matrix Multiplication".

8.2.3 Scaling factor for softmax

One pragmatic detail. First, let's quickly define:

Notation 32

Reminder that keys and queries are both vectors of length $(d_k \times 1)$.

We have one problem: the larger d_k is, the more terms in our dot product: our dot product can grow unreasonably **large**.

- This can cause computational issues.

So, we **normalize** our dot product by a factor of $\sqrt{d_k}$.

Key Equation 33

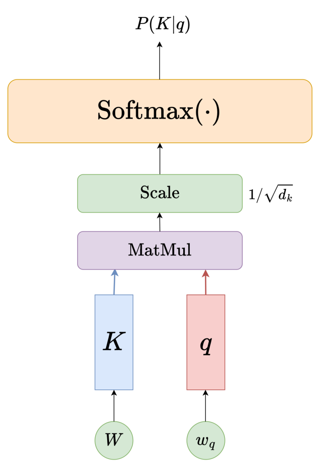
When computing **attention weights**, we **normalize** our dot product $q^T k$ by a factor $\sqrt{d_k}$.

- This compensates for the fact that longer vectors will create larger dot products.

So, when computing our attention weights α, we use the formula:

$$\alpha(q, K) = \text{softmax}\left(\frac{q^\top K^\top}{\sqrt{d_k}}\right)$$

It still has shape $(1 \times N)$.



We scale down our MatMul by the appropriate factor.

8.2.4 The Attention Mechanism: values, attention

Now, we have a collection of **attention weights**: each one tells us relevant each word is to q.

- Now, we want to make them useful. Our original goal was to get an **average** sense of what "mexican" food is like.

To make this concrete, we'll introduce our third embedding: the **value vector**.

- **Value** v: Each food has a **value vector**, directly storing information about a word.
  - Unlike the key/query vectors, this embedding isn't based on **similarity** to other words.
  - Instead, it usually contains more direct **information** about our word: in this example, maybe it contains the price, calories, ingredients, etc.

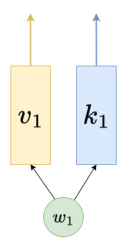
Note that, in a real model, value vectors are often "learned" during training. So, they won't always contain such simple, easily explained data.

Definition 34

The **value vector** v represents a word, and stores useful **information** that it can contribute to the **query**.

- It answers the question, "what useful data could this word contribute to the query?"

By **adding together** the value vectors from each word **relevant** to the query, we can get an overall "**averaged value**" for q.



Each word has both a value and a key attached to it.

For our example, let's suppose that the value vector contains price, calories, and salt.

$$v_i = \begin{bmatrix} \text{price}_i \\ \text{cal}_i \\ \text{salt}_i \end{bmatrix} \tag{8.13}$$

We want to get an "average" calorie count for mexican food.

- Some foods are **common** for mexican food, and some are more rare.
- So, to get an average, we'll need to **emphasize** more "common" mexican food.

How do we do that? Using our **attention weights**: the larger the attention weight, the more "**relevant**" a food is to our mexican food calculation.

If we use q to represent mexican food, and k_i is the **key** for the i^{th} food, we get:

$$\text{cal}_q = \overbrace{\sum_i P(k_i|q)\ \text{cal}_i}^{\text{Weighted average}} \tag{8.14}$$

Rather than repeating this process for each row of v, we can just do a **weighted average** of the whole vector, at the same time:

$$v_q = \sum_i P(k_i|q)\, v_i \tag{8.15}$$

Key Equation 35

Each word i has a **value vector** v_i, which represents all of the useful **information** it can provide to the **query**.

- We can use a **weighted average** to combine all of these value vectors together: this provides the "**overall context**" for the query.
- Each value is weighted based on its **attention weight** $P(k_i|q)$: how likely it is to be relevant.

$$v_q = \sum_i P(k_i|q)\, v_i$$

This is the calculation for **attention**.

Just like we did for the $k_i \cdot q$ operation, we can re-write this in terms of matrix multiplication.

- We'll change from $P(k_i|q)$ to $P(K|q)$.

$$\mathbf{P}\left\{K \;\middle|\; q\right\} = \begin{bmatrix} \mathbf{P}(k_1 \mid q) \\ \mathbf{P}(k_2 \mid q) \\ \vdots \\ \mathbf{P}(k_N \mid q) \end{bmatrix}^\top = \text{softmax}(q^\top K^\top)$$

- We'll stack all of our value functions v_i into a matrix V.

$$V = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \dots & v_N \\ | & | & & | \end{bmatrix}^\top \tag{8.16}$$

Notation 36

We'll assume that we have N value vectors of length d_k.

- v_i has shape $(d_k \times 1)$, V has shape $(N \times d_k)$.

Now, we can compute with every value vector at once:

If you study the classic "Attention is all you need" paper, you'll find that their version of k and q are transposed compared to ours.

Key Equation 37

We can compute **attention** using matrix multiplication:

$$\text{Attention}(q, K, V) \quad = \quad \text{softmax}\left(\frac{q^\top K^\top}{\sqrt{d_k}}\right)V$$

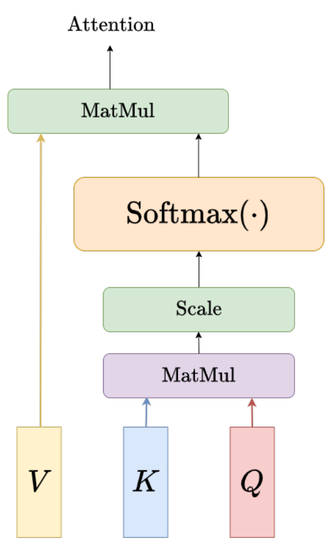
Where $\text{softmax}\left(\frac{q^\top K^\top}{\sqrt{d_k}}\right)$ computes our **attention weights**.

- Under this definition, attention has shape $(1 \times d_k)$.

Definition 38

$\text{Attention}(q, K, V)$ is the **weighted average** of all of our value vectors (transposed).

- Attention is the **result** of **aggregating** information from N different words: each word is represented by a key k_i, and a value vector v_i.



We now have a completed representation of attention.

In the "Attention is all you need" paper, this diagram is analogous to Figure 2 (left).

Here, we omit the "Mask" layer (discussed later).

With this, we can summarize the basic idea of attention:

Concept 39

Attention is a mechanism that allows you to **combine** information from multiple tokens, weighting each token by how **relevant** it is.

This mechanism is broken into three parts:

- **Value vector** v: **what information** are we trying to combine?
- **Query vector** q: what kinds of words are **relevant** to this search?
- **Key vector** k: what kinds of searches is this word **relevant** for?

Each token has a **value vector** (information from that token), and a **key vector** (used to compare this token to the query).

Note that this isn't the **only** way to do attention:

Clarification 40

There are multiple ways we can implement attention.

- For example, we use $q \cdot k$ to measure similarity, but we could **replace** it with a different metric.

Reminder: a "metric" is just "a way of measuring something. The dot product is a **similarity metric** for vectors.

So far, we've mostly focused on the mathy details of **how** attention works: an abstract idea of "relevance" between words, "combining" the value ("meaning") of different words, etc.

- Here, we'll try something different: we'll focus more on **why** we use attention, and how it applies to a real, concrete situation.

8.2.5 Why we need context

Attention is designed to integrate information from other, **nearby** words. But why do we need to do this?

- Because language is heavily dependent on context.

Consider the task of **language translation**: we have a sentence in one language, and we want to convert it into another language, while **preserving the meaning**.

Let's translate the sentence:

I miss her warm smile.

We'll focus on the word "warm".

- Most commonly, "warm" means "higher-than-average temperature". For example, being under a blanket is warm.
- But most humans would say that, in this situation, the word "warm" means 'friendly' or 'kind'.

We know this because of the *context*: a "warm smile" usually means a "kind smile". The word 'smile' has **changed the meaning** of the word 'warm'.

Concept 41

The meaning of a word can change based on the other words which are **nearby**.

- This is why we need to integrate **context** for language processing.

If our machine blindly translated "warm", without context, we could've ended up with the wrong meaning in another language.

8.2.6 Why we need *attentive* context

So, we need to use context. But what makes attention special?

- It allows us to figure out **which words** are most important to us!

In the above sentence, the word "smile" changed the meaning of "warm".

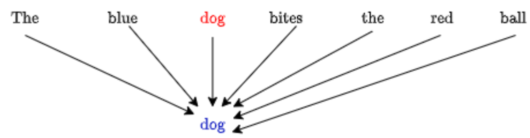
- How do we know that "smile" is the important context word? "her" is equally far from "warm".
- Attention handles this for us: we "**pay more attention**" to the word 'smile' than the word 'her', when we're trying to understand "warm".

Concept 42

Attention allows us to determine which parts of the **context** are **most important** to a particular word.

8.2.7 Self-attention

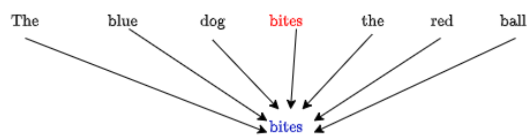
Attention has given us a tool for comparing one word to every other word in a sentence.



"dog" is represented by a query q_{dog}, compared to the key for every other word in the sentence. This gives us our attention weights.
Next, we combine these weights with the value vector for each word. This gives us our **attention**: the "contextual meaning" of the word **dog**.

This has a limitation: we're only focusing on a single word, "dog".

- But we need to get the meaning of **every word** in the sentence, based on the context from other words.



This time, we use the query q_{bites} for the word "bites". However, the key and value vectors are still the same for each word.
We need to repeat this attention process once for each word.

This is interesting: we're seeing how much each word affects each other word in the sentence. We're seeing how the sentence provides context for **itself**.

- This is why we call this **self-attention**.

Definition 43

Self-attention is the process of using attention on every word in a passage.

- For the i^{th} word, we compare it to **every other word** in the passage.

This allows us to interpret each word, based on the **context** provided by the rest of the sentence.

Technically, we also compare each word to itself.

8.2.8 Self-attention in matrix form

How do we handle this, mathematically?

- When we are getting the attention for word w_i, we use its query q_i to compare it to other words in the sentence.

We've gone from having a single query q to having many q_i: one for each word in the sentence.

$$Q = \begin{bmatrix} | & | & & | \\ q_1 & q_2 & \dots & q_N \\ | & | & & | \end{bmatrix}^T \tag{8.17}$$

Let's note some conventions:

Notation 44

A few useful **dimensions**: in an attention problem, we have...

- n_k keys of length d_k.
- n_q queries of length d_q.
- n_v values of length d_v.

In **practice**, we usually take $d_k = d_q = d_v$, and simply refer to all three as d_k.

- Each column vector (k_i, v_i, q_i) has shape $(d_k \times 1)$.

In **self-attention**, we take $n_k = n_q = n_v$, and simply refer to all three as N.

- Matrices K, V, and Q all have shape $(N \times d_k)$.

Each of these queries will create a separate set of attention weights, $\text{softmax}(q_i^T K)$.

Key Equation 45

We define the **self-attention weight** matrix A, to represent all attention weights:

$$A = \begin{bmatrix} \text{softmax}\left(q_1^T K^T / \sqrt{d_k}\right) \\ \text{softmax}\left(q_2^T K^T / \sqrt{d_k}\right) \\ \vdots \\ \text{softmax}\left(q_N^T K^T / \sqrt{d_k}\right) \end{bmatrix} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

This is an $(N \times N)$ matrix.

- Row i tells us all of the attention weights applied to query q_i.
- Col j tells us the attention weights for key k_j.
- Element α_{ij} (row i, col j) tells us, "how **important** is word j (key) as context for word i (query)"?

We can use this to get the total attention:

Note that the elements in row i must add up to 1: we have softmax.

This is not true for column j: they're probabilities for different queries.

Key Equation 46

The **self-attention** equation is given as

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

It is a $(N \times d_k)$ matrix.

Row i gives the **averaged value vector** $y^{(i)}$ for the i^{th} word, based on all of the surrounding **context**.

We could view this as the "output" for the i^{th} word.

- We can write this in element-wise form:

$$y^{(i)} = \sum_{j=1}^{N} \alpha_{ij} v_j$$

One theme we'll run into, many times in this chapter, is that attention-based models benefit from being able to **parallelize**:

Concept 47

Transformer Parallelization I

Computing self-attention can be strongly parallelized:

- Each $q_j^T k_i$ term is **independent** of the others: we can compute all of the key-query dot products **at the same time**, rather than waiting for one to finish before starting the others.
- We can compute each softmax term at the same time, as well.

This remains true if we're using **cross-attention**, where the keys and queries come from different words.

8.2.9 Positional Encoding

First, a problem we need to address:

- Currently, our key k_i is determined by asking the **identity** of the word at index i.
- This key doesn't encode information about the **position** of this word in the sentence.

But clearly, the position of a word will determine its meaning.

- **Example:** "The cat lies on the green table" and "the green cat lies on the table" are **not the same**: moving the word "green" to a different index changes its meaning.

We fix this by adding information to keep track of this position.

Definition 48

We apply **positional encoding** to each word embedding: each embedding includes information about the **position** of a word in the text.

- This allows our attention mechanism to use this information when deciding the **relevance** of different words.

8.2.10 Masking

One common use for transformer models is **text prediction**: learning what word should come next, based on what it has seen so far.

Typically, we would give our model the text, and give it a chance to try to **predict** each index, **before** it can see it.

We need to prevent our model from being able to **cheat**:

- We don't want our model to be able to see the words it's supposed to be predicting.

The blue ~~dog bites the red ball~~

So, we'll hide those words, so our model can't see them. In this case, we want our model to predict the next word: "dog".

This is called **masking**.

Definition 49

Masking is a technique where we **hide** some information from our model, so it can't use that information.

- For example, if our model is being used to **predict text**, we hide the text that it's trying to predict.

However, the word "masking" can apply to **any** situation where we want to hide tokens from the model.

8.2.11 Attention Heads

We have a system for "attention": deciding which words provide the **most important** context/information, and paying more attention to those words.

But there's something we haven't considered: the "importance" of different words, depends on **what you're interested in**. Let's consider a couple examples:

- **Syntax**: which words are **subjects, objects, verbs, adjectives**?

  Example: "The boy kicks the red ball": our focus is on the word "ball".

  – "red" is important for color.

  – "kicks" is important for knowing what's happening to the ball.

  – "boy" is important for knowing who is acting on the ball.

- **Semantics**: which words change the **meaning** of our target word?

  Example: "I miss her warm smile": our focus is on the word "warm".

  – The word "smile" changes the meaning of warm from 'high temperature' to 'kind'.

- **Coreference**: which words are referring to the **same object**?

  Example: "John said that he isn't hungry": our focus is on the word "John".

  – "he" refers to the same object as "John": if we apply something to the word "he", it also applies to "John".

Concept 50

What is "**important**" in a sentence can **change**, based on what you're trying to study.

- And generally, these ideas of "important" won't agree with each other.

Above, we suggested several different perspectives on "what is important".

- Rather than having our attention mechanism try to handle all of these kinds of importance, we could create a **separate mechanism** for each one of them.

We'll do just that: each "perspective" will be represented by a different mechanism. We call each of these, **attention heads**.

Definition 51

A transformer model may use **multiple** attention mechanisms **at the same time**:

- Each attention mechanism is a different "**perspective**" on our data: it focuses on different aspects of the text (grammar, meaning, tone, etc.)
- To accomplish this, each one represents a word w with a different k, q, and v.

We call each mechanism one **attention head**.

If we have 3 different attention heads, each one may **encode** the word "silly" differently. We could have three different keys for this one word: k^1, k^2, and k^3.

- Each head will require a distinct word encoding: $K^{(h)}$, $Q^{(h)}$, and $V^{(h)}$.

Concept 52

Transformer Parallelization II

Each attention head uses calculations which are **independent** from the others: we can compute each attention head at the same time!

8.3 Transformers

Now that we've built up attention, we'll use it to build a **transformer**. We'll assume our transformer uses self-attention, though the math works out similarly even if it doesn't.

Definition 53

A **transformer block** is a collection of attention heads running in **parallel**, applied to the same text.

A **transformer** is composed of several transformer blocks in **series**: the output of one block is the input of another.

8.3.1 How to create embeddings

Something we've ignored for a while is, "how do we construct our **embeddings** K, Q, and V"?

- We aren't actually given them: we're given a sequence of **tokens**: each token is a vector x representing a word. So, our whole body of text is a matrix X.

Each vector is length d: this is different from the length of the embedding, d_k.

We'll compute each embeddings by using a **linear transformation**:

Key Equation 54

We use **projection matrices** W_k, W_q, and W_v to transform each **token** $x^{(i)}$ into embeddings k,q, and v.

$$\begin{aligned} k_i &= W_k^\top x^{(i)} \\ q_i &= W_q^\top x^{(i)} \\ v_i &= W_v^\top x^{(i)} \end{aligned}$$

All three projection matrices have shape $(d \times d_k)$.

All of our tokens are stored in matrix X:

Reminder that:

d is the original length of $x^{(i)}$

d_k is the length after embedding.

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(N)} \\ | & | & & | \end{bmatrix}^\top \tag{8.18}$$

Unlike our usual X, this is transposed: shape $(N \times d)$.

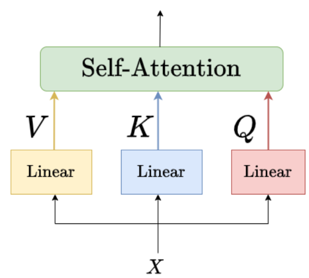
We can get the keys, queries, and values for all of our vectors in matrix form:

Key Equation 55

We can compute K, Q, and V:

$$K = XW_k$$
$$Q = XW_q$$
$$V = XW_v$$

Based on this linear transform, we modify our diagram:



We have to generate V, K, and Q before we can use them.

Concept 56

One benefit of computing keys, values, and queries based on **weight matrices** is that we can **train** these matrices:

- Rather than manually designing the **embeddings**, we can allow our model to learn whichever embedding is most useful.

8.3.2 Attention Heads

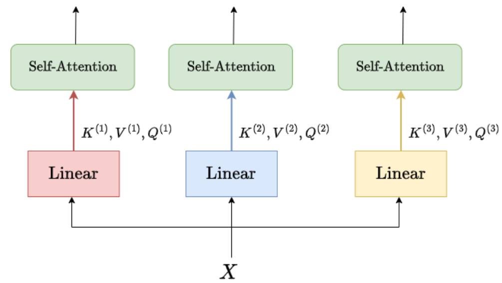
What if we have **multiple** attention heads?

Notation 57

If we have H attention heads in a transformer block we'll indicate the h^{th} head with:

$$K^{(h)} = XW_{h,k}$$
$$Q^{(h)} = XW_{h,q}$$
$$V^{(h)} = XW_{h,v}$$

Each attention head is applied in parallel:



Here's an example with $H = 3$ attention heads. Each uses a distinct set of keys, values, and queries.

To finish off our multi-headed attention unit, we do two more things:

- Transform each token back into the original dimensions: going from length-d_k to length-d.
- Combine the results from each attention head: we'll do a **weighted average**.

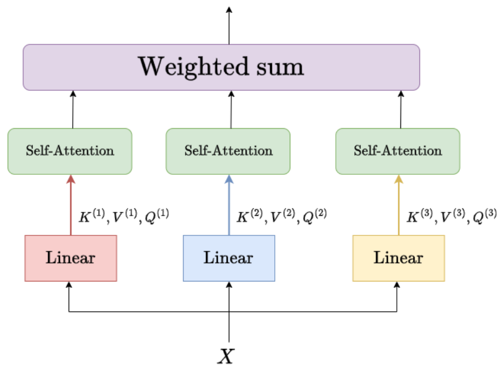
Key Equation 58

After computing attention for each head, we take a **weighted average** of our heads, combining them together:

- For each head, we use matrix $W_{h,c}$ to **scale** the weight of each head, and convert them back to their original **shape**.
  - $W_{h,c}$ has shape (d_k, d).
- We **add** together the results, gathering information from each head.

$$u = \sum_{h=1}^{H} \text{Attention}\left(Q^{(h)}, K^{(h)}, V^{(h)}\right) W_{h,c}$$

u, the final output of our **multi-headed attention**, has shape $(N \times d)$, where the j^{th} column represents the j^{th} token.



We have our completed multi-headed attention unit!

In the "Attention is all you need" paper, this diagram is analogous to Figure 2 (right).

Instead of directly doing a weighted sum, they concatenate each attention head, and then apply a linear weight W^o.

These are equivalent.

Note that this is the same shape as our original input, X:

$$U = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \dots & u^{(N)} \\ | & | & & | \end{bmatrix}^\top \tag{8.19}$$

In fact, we can compute this multi-headed attention, one $u^{(i)}$ at a time.

Reminder that α_{ij} is an attention weight from A, and $v^{(j)}$ is a value vector of V.

Key Equation 59

We can combine our multi-attention heads as

$$u^{(i)} = \overbrace{\sum_{h=1}^{H} W_{h,c}^\top}^{\text{Heads}} \overbrace{\left(\sum_{j=1}^{N} \alpha_{ij}^{(h)} v_j^{(h)} \right)}^{\text{Attention}}$$

This is a nested sum: $\sum_{h,j}(\cdot)$,

not a product of two sums,

$(\sum_h(\cdot)) \cdot (\sum_j(\cdot))$

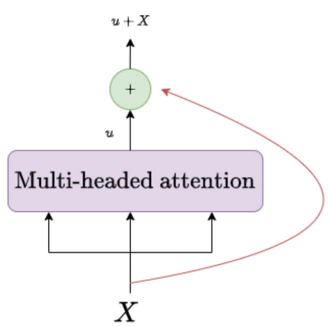
8.3.3 Residual Connections

Our next component will handle a problem with **deep** neural nets that we've addressed before: vanishing/exploding gradient.

> **Definition 60**
>
> *(Review from Neural Networks 2)*
>
> **Vanishing gradient** occurs when a deep neural network ends up with **very small gradients** in the **earlier** layers.
>
> This happens because a deeper neural network has a **longer chain rule**: if all of the terms are **less than one**, they'll multiply into a very small value, "**vanishing**".
>
> This means that our gradient descent will have **almost no effect** on these earlier weights, **slowing down** our algorithm considerably.

In short: the "further away" from our input layer, the messier our gradients get.

One simple solution is to include our original, **unmodified** input, deeper in the neural network: we just **add** it, so that our second layer gets to see the input data, too.



Our output contains direct information about the input. This hopefully improves training.

> **Definition 61**
>
> In a **residual block**, the **input** x is added to the **output** $F(x)$ of the block (in our case, multi-headed attention) .
>
> $$\text{output} = F(x) + x$$
>
> - This is designed to reduce the risk of **vanishing gradient**, by directly exposing deeper layers to the input
>   - The long chain rule is what causes vanishing gradient: we've created a shorter chain rule.

If you ever hear someone refer to a "ResNet" or "Residual Network", this is a CNN that uses the same technique!

8.3.4 Layer Normalization

Another topic from the NN chapter: **batch normalization**.

Definition 62

(Review from Neural Networks 2)

Batch Normalization is a process where we

- Standardize the pre-activation for each layer **across data points in the batch** using mean μ_i and standard deviation σ_i (for the i^{th} dimension).

$$\overline{Z}_{ij} = \frac{Z_{ij} - \mu_i}{\sigma_i}$$

- Choose the new mean and standard deviation for the pre-activation using $(n \times 1)$ vectors G and B

$$\widehat{Z}_{ik} = G_i * \overline{Z}_{ij} + B_i$$

In short: we set the (mean, sd) to (0,1) and then scale it back up to (G_i, B_i).

We would get the same kinds of benefits from **normalization** in transformers as we did before in NNs.

Stabilizing our training process, mostly.

But rather than normalizing across multiple **data points** (batch), we'll normalize across the **features** (layer) of a single token.

Key Equation 63

Suppose we have a $(d \times 1)$ data point $z = \begin{bmatrix} z_1 & z_2 & \cdots & z_d \end{bmatrix}^T$.

Layer normalization computes the mean μ_z and standard deviation σ_z across our **features** z_i

$$\mu_z = \frac{1}{d}\sum_i z_i \qquad \sigma_z = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(z_i - \mu_z)^2}$$

And then **normalizes** them.

$$z_{norm} = \frac{z - \mu_z}{\sigma_z}$$

Finally, we scale them back up, to have mean β and s.d. γ.

$$\text{LayerNorm}(z; \gamma, \beta) = \gamma\left(\frac{z - \mu_z}{\sigma_z}\right) + \beta$$

Now that we understand this process, we can apply this to our transformer model:

Layer normalization can be used on a single data point, while batch normalization requires many.

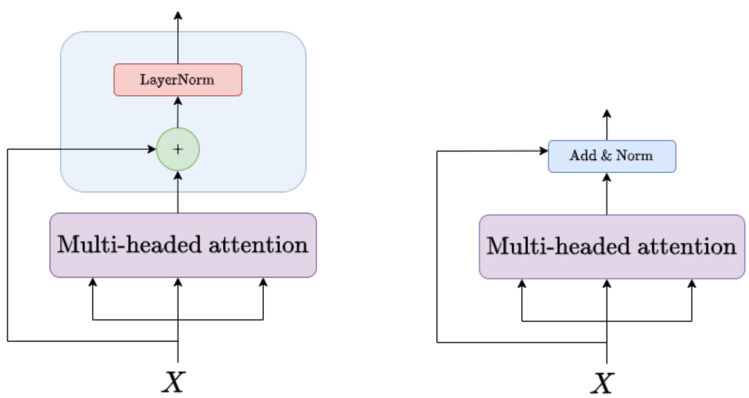
- After we get $u + x$ (creating the residual block), we use layernorm on each token separately:

Concept 64

At the end of our **residual block**, we apply LayerNorm to each of our tokens separately

- We take our $(N \times d)$ object $u + X$ and normalize the features of each of our N tokens (shape $(d \times 1)$) **separately**.

$$u_{norm}^{(i)} = \text{LayerNorm}(u^{(i)} + X^{(i)}, \gamma_1, \beta_1)$$



We append a LayerNorm layer. We'll follow the convention from the "Attention is all you need" paper and combine these into a single unit: "Add+Norm".

With this, our Residual Connection is complete.

8.3.5 Feed Forward

In our CNNs, after convolution, we would use a **fully-connected feed-forward network** to analyze the processed data.

- We'll follow the same sort of pattern here: the main difference being that we apply feed-forward after only **one layer** of **multi-headed attention**.

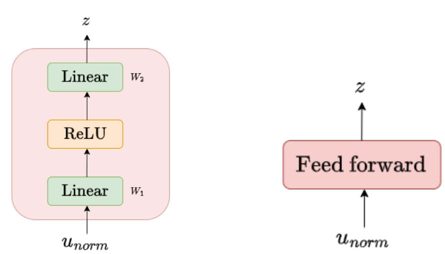
Key Equation 65

After we apply **Add & Norm** to our Multi-headed attention, we run the output through a **feed-forward** layer, processing the data it receives.

- We use a linear layer W_1, a ReLU layer, and another linear layer W_2.

$$z = W_2^T \text{ ReLU}\left(W_1^T u_{norm}\right)$$

We can think of this as apply a hidden FC layer to our network, followed by another linear transform.



Linear, ReLU, linear. Once again, following "Attention is all you need", we simply call this the "Feed forward" Layer.

We'll follow this up with another LayerNorm:

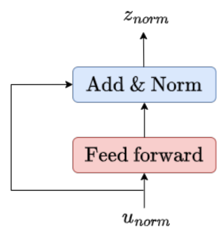
Meaning, we use another residual block.

Concept 66

After our **feed-forward layer**, we apply **Add & Norm** again.

$$z^{(i)}_{norm} = \text{LayerNorm}(z^{(i)} + u^{(i)}_{norm}, \gamma_2, \beta_2)$$

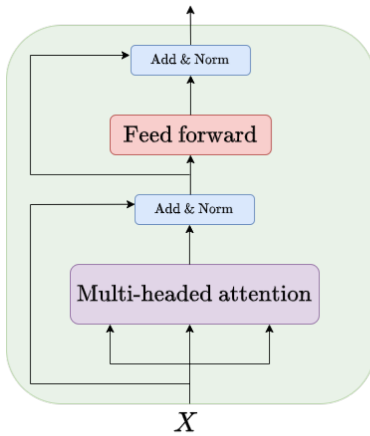
This is the final output of our **transformer block**.



z_{norm} is the final result of our transformer block.

8.3.6 Transformer Block

With this, we can assemble our transformer block, top to bottom:



We have a transformer block!

Definition 67

A **transformer block** is made up of several functions composed together:

- **Multi-headed attention**
  - Each head encodes the input text X as keys $K^{(h)}$, a value Qh, and vectors $V^{(h)}$: one for each token.
  - Based on these, we compute **attention**.
  - Finally, we linearly combine information from across all H heads.
- **Add & Norm**
- **Feed-forward**
  - We apply a fully-connected layer (linear+ ReLU), then another linear unit.
- **Add & Norm**

Both "Add & Norm" layers accomplish the same thing: they create a **residual connection**.

- We add the input to the output, and then layer normalize.

Concept 68

Each layer of our transformer block serves an important function:

- The **multi-headed attention** layer explores connections between tokens, and provides information about the internal structure of our data.
- The **feed-forward** layer processes our information nonlinearly (via ReLU).
- The **add & norm** layers create residual connections between the input/output of the preceding layer, improving our gradient-training process.

From here, we can design a transformer model by combining many of these transformer units in series.

8.3.7 Translation Task: training

We just have one more layer of complexity, before we finish. Let's consider a training example, for the task of translating from english to spanish.

$$\text{I'm not hungry yet} \implies \text{Todavía no tengo hambre}$$

Our transformer will start by predicting the **first word** in the sentence: presumably "todavía".

- But not necessarily: if our model isn't **well-trained** yet, it might predict some random word, like "espacio".

It's also possible for us to have multiple valid translations, but we'll ignore that for now.

Now, we want to predict the *second word* in our output. But we just brought up an important problem:

- The best "second word" in our translation is **dependent** on the first word. We should factor that into our model, when predicting the second word.
- If our first word was wrong, then we're **more likely** to use an incorrect second word!

The solution? Instead of using the first word we predicted, we use the **correct** first word.

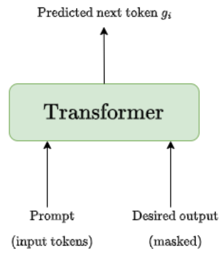
- Only one condition we need to remember: we need to **mask** the rest of the "correct" output sentence, so our model can't use it to cheat.

> **Concept 69**
> When **training** our model to complete a language task, our model predicts each word (token) one-by-one, based on two pieces of data:
>
> - The entire **input prompt**
> - The **desired output** sequence for every token **before** the one we want to predict.

Example: Suppose we're predicting the third word in our above sentence. We'll use the first two "correct" words as part of our model:

In this case, "tengo" is the word we want to predict.

$$\begin{bmatrix} \text{I'm not hungry yet} \\ \text{Todavía no} \end{bmatrix} \implies \text{tengo}$$



Our model is actually trained with *two* inputs.

Something to take note of: we predict the i^{th} token based on the input, and the first $i-1$ **desired inputs**.

- That means that, when predicting token g_i, we don't care what we predicted for the previous tokens!
- We don't need to finish predicting token i to predict token $i+1$: we can do them at the same time!

Concept 70
Transformer Parallelization III

Predicting token i is an independent calculation from predicting a second token j.

- That means we can predict every token in our sentence at the same time!

This is a *huge* advantage in training transformers: it can essentially think about the entire sentence at the same time, massively speeding up training.

Clarification 71
We can't parallelize **token generation** when we're using our model **after** training:

- We can parallelize during training because we're using the **desired output** for the previous $i-1$ tokens.
- When using our model for unseen data, we don't have "desired output": we have to use our **actual output** for the previous tokens.

We have to **wait** for our model to predict the first $i-1$ tokens, before it predicts the i^{th} token.

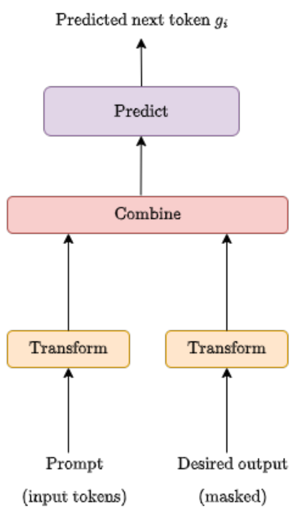
8.3.8 Encoder + Decoder Structure

Now, we have to structure our transformer model to be able to handle both the input and desired output.

Concept 72

There are three tasks we want our model to complete:

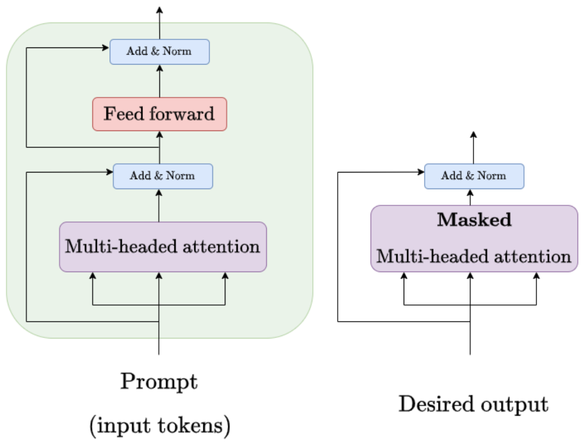
- **Process** our input sequence,
- **Process** our target sequence (desired output),
- **Combine** the two sequences of information
- **Predict** the next character based on this data.



We'll choose functions to handle each of these tasks.

- We'll process our prompt using a complete transformer block. This unit is our **encoder**: we encode our prompt in a form that is more **meaningful** to our computer.
- However, for our desired output, we'll only use **attention**, learning about the internal structure of the output.
  - We'll also use this unit to **mask** our output (so our transformer can't "look ahead" at future tokens).

Why not add the feed-forward layer? We'll add it later: there's another component we want to add first (see below).



We'll add the second feed-forward unit later. **First**, we want to **combine** information from our input, with the earlier tokens of our output.

We accomplish this with another attention unit: this time, we'll use **cross-attention**.

Definition 73

In **cross-attention**, our **queries** come from one sequence of text, while our **keys/values** come from a **different** sequence of text.

- As opposed to **self-attention**, where our keys/values/queries all come from the same sequence.

Our goal is to use the **earlier** part of our **output sentence** to determine which parts of our **input** we should **pay attention to** in our input sentence, when choosing the next token.

For example: if our output sentence already includes a word, it might be less likely we'll need to use that word again.

- Our **keys/values** represent the words we might want to **pay attention to**.
- Our **queries** help us decide **what** to pay attention to.

We can use "attend" as a verb meaning "pay attention to": this is common when talking about transformers.

For example, in this case, we're deciding "which input tokens to attend to".

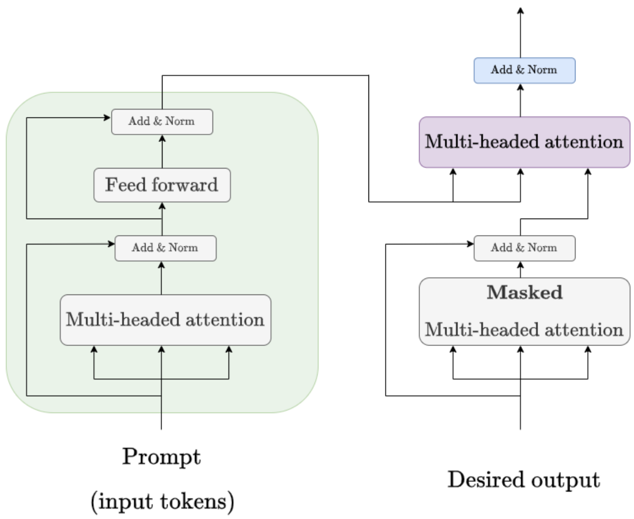
Thus, we'll use keys/values from our encoded input, and queries from our previous output tokens.

Concept 74

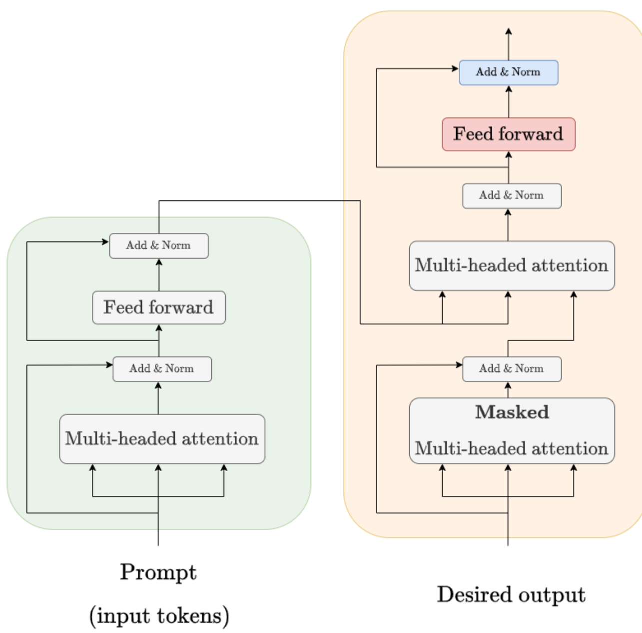
We integrate our **input tokens** and our previous **output tokens** using **cross-attention**: we apply attention, using

- Our encoded input as our **keys and values**.
- Our attended output as our **queries**.

This is our **encoder-decoder layer**.



Now that we've integrated information from both our input and output, we finally include our **feed-forward** unit: we'll process our integrated information.



This unit on the right is called our **decoder**.

We've got a complete encoder/decoder setup:

Concept 75

We break our transformer into an "**encoder**" and "**decoder**" unit:

- The encoder transforms our **input** into a representation that contains more useful information: connections between tokens in the prompt, etc.
- The decoder transforms that encoding into a **output**/response: this decoder takes the information we've gathered, and applies it to our problem.

Consider the translation example:

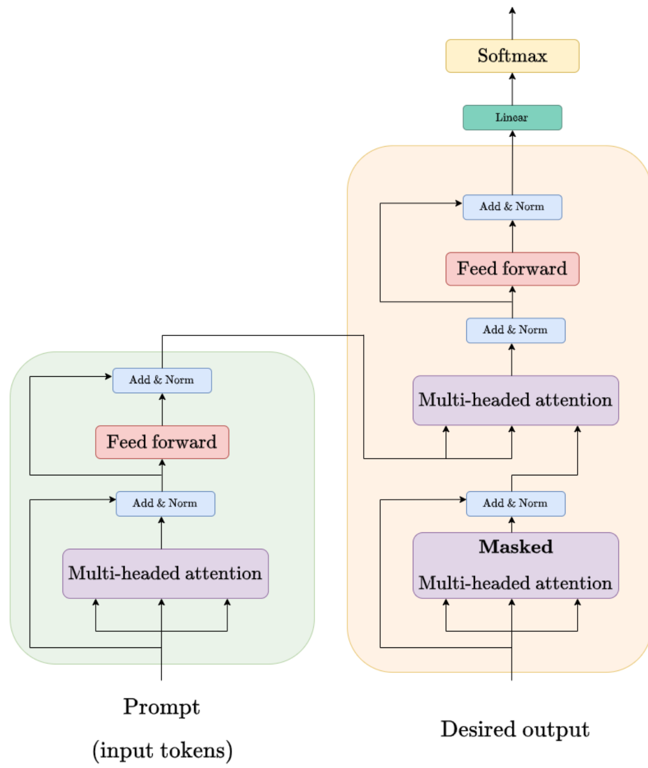
- The encoder stores our English text in a form that hopefully represents the **meaning**.
- The decoder "decodes" that representation into a form we can read, but in a **different language**: Spanish, in our example.

In this analogy, we've created a special "code" that we write in English, and read in Spanish.

8.3.9 Predicting a token

Only one step left: using this decoded information, we need to **choose** our token. This is the multi-class classification problem: we use the same protocol as we always do.

- We linearly transform our data: each token gets a "**score**", based on how likely we think it is to be the correct one.
- We apply **softmax**, to turn these scores into probabilities. We get a probability for every possible token.



This is essentially a completed transformer.

This is the (now-famous) diagram from the "Attention is all you need" paper!

We've excluded the initial embedding (turning words into vectors) and positional embedding (adding information about the position of each word in the sentence).

We could include those for completeness, but that would just take up more space.

Only one detail still missing:

- Our decoder/encoder typically has **several copies** of the same unit in a row: for example, we might have 3 transformer blocks in a row for our encoder.

Notably, this is only one kind of transformer model: which architecture we use depends on the problem, cost constraints, etc.

8.3.10 Training Process

We typically train transformer models in two stages: pre-training and fine-tuning.

Definition 76

In **pre-training**, we expose our model to a very large dataset of human language, so it can learn **patterns** in that language.

- We can use **unlabelled** data in this stage: thus, we have an unsupervised/self-supervised problem.

This stage of training is typically expensive.

Definition 77

In **fine-tuning**, we take our pre-trained model, and train it for a **specific task**.

- We use **labelled** data in this stage.

It tends to be much faster and less expensive than pre-training.

8.3.11 Variations

We could make variations on this network:

- Use more/fewer decoder/encoder units.
- Use a different style of attention (rather than the dot product, we use some other similarity metric).
- Move LayerNorm to different parts of the network.

8.4 Terms

- Natural Language Processing (NLP)
- *(Review)* Convolutional Neural Networks (CNNs)
- Locality
- Recurrent Neural Networks
- *(Review)* Word Embedding
- Co-occurrence
- Context window
- Skipgram
- Word2vec
- Token
- Key Vector
- Query Vector
- Value Vector
- Attention Weights
- d_k
- Attention
- Self-attention
- Positional Encoding
- Masking
- Attention Head
- Projection Matrix
- Multi-headed attention
- Residual Block
- Residual Connection
- Layer Normalization
- Add & Norm
- Feed-forward layer (transformers)

- Transformer Block
- Cross-attention
- Encoder (Transformers)
- Decoder (Transformers)
- Encoder-Decoder Layer
- Pre-training
- Fine-tuning