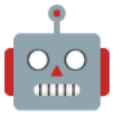


Transformers Memo

Prelude

Required Background

This document assumes that you have some familiarity with neural networks and training them (and all the machine learning, linear algebra and vector calculus that entails). Terms like “gradient”, “parameters”, “softmax”, and “regularization” will be tossed around without

accompanying explanation. Other than that, I am basically assuming that you haven't heard of a Transformer, and am building intuition from the ground up, starting with self-attention. If you don't have this background, I'd suggest getting up to speed on that first before reading this.

Word Embeddings

Implicit in all following discussion of machine translation and language modeling is that you can feed "words" to a neural network. Of course, neural networks speak in numbers, not words, so words have to be represented as a *word embedding* vector in $\mathbb{R}^d$. There is a fixed vocabulary, and each word in the vocabulary corresponds to a *d*-dimensional vector in a lookup table. In the past, these might have been learned separately (via a procedure like GloVe or Word2Vec), and then taken as fixed inputs to the model. In more recent research, including the Transformer, embeddings are initialized randomly, and learned during training to minimize loss.

For open-vocabulary tasks (where we want the model to be able to understand words it may not have seen during training), sub-word units—chunks of words, characters, or Unicode code-points—are used instead of words. Whether using words or sub-words, an important preprocessing step is to break pieces of text into *tokens* (words or sub-words), then map these tokens to integer indices into the vocabulary. It is these integer indices, rather than passages of text, that are fed directly to the neural network.

Transformer Architecture & Attention

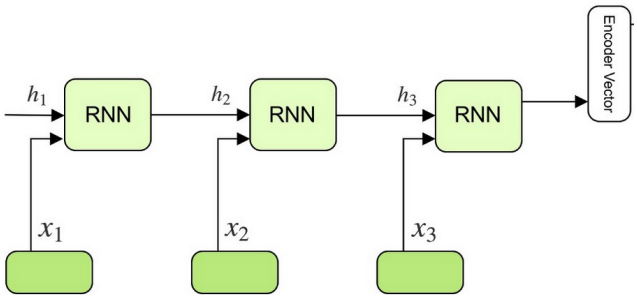
The Transformer model originates from the 2017 paper "Attention is All You Need" (Vaswani et al.). The Transformer was conceived as a method for sequence-to-sequence mapping for machine translation, but the attention-based encoder and decoder introduced in the paper have spread like wildfire, and have been used to great effect for all sorts of language-modeling tasks. Pretty much all recent state-of-the-art work in large language models (BERT, GPT-3, etc.) is derived from this attention-based architecture. It has also caught on in computer vision, and more recently, even reinforcement learning! In this section, I'll give a high-level overview of what attention is, and how transformers use it to model sequences in the natural language setting.

The Idea of Attention

N.B. — for a more detailed/visual explanation, see this article or this video.

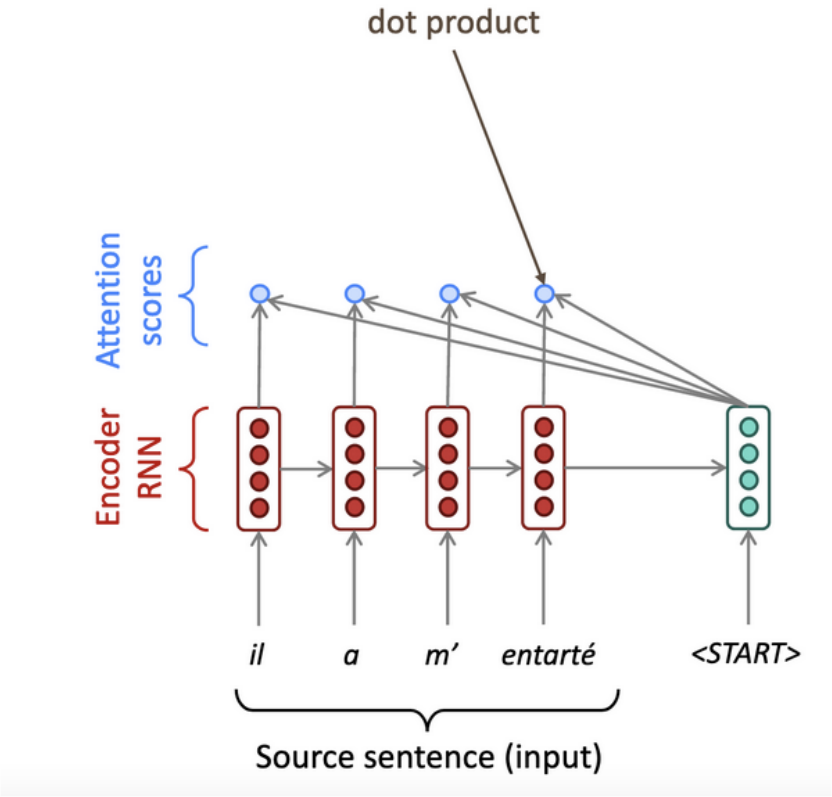
Attention in Recurrent Neural Networks

Before the advent of transformers, neural machine translation (translating from language to another using neural networks) was based on *recurrent* models, which processed the input sentence one word at a time using shared weights, generating a new hidden state h_t for each position t in the input sentence (based on h_{t-1} and the t-th word in the sentence). At the end of the sequence, the final hidden state would be used as the representation of the whole sentence. This was limiting, because all the information from the whole sentence had to be compressed into that one state. It rendered models unable to deal effectively with longer sentences.



Recurrent neural network encoding a sequence.

Attention was initially popularized as a solution to this problem: instead of just using the final hidden state to represent the sentence, Bahdanau et al. proposed using *all* of the hidden states (one for each input word) to represent the sentence. At each step of decoding (where the representation of the input sentence is "decoded" into the target language one word at a time), the hidden states are averaged, weighted by their relevance to the current step of decoding. In this way, the decoder can assign more weight ("pay attention") to hidden states that are most relevant to the current decoding step. The weights, i.e. the relevance of one vector to another, can be determined most simply by a dot product. This (roughly) works because vectors that point in similar directions tend to have more positive dot products, and vectors that are opposite or orthogonal have smaller or negative dot products. The raw scores of the dot product are then normalized by a softmax function.



Dot-product attention weights. Source: Chris Manning's CS224N slides.

Self-Attention Without RNNs

Attention caught on as a way to improve recurrent models for machine translation, as described in the previous section. But it really took off with the seminal paper, "Attention is All You Need," by Vaswani et al. This paper dispenses with the recurrent, "one-word-at-a-time" architecture in favor of what it calls the Transformer, which processes an entire sequence in parallel. In order to capture dependencies between words (or tokens) in a sequence, the authors use *self-attention*, which is like the attention mechanism discussed before, but applied *reflexively* to compute the relevance of words in a sequence S *with respect to each other* (rather than with respect to a separate decoder state).

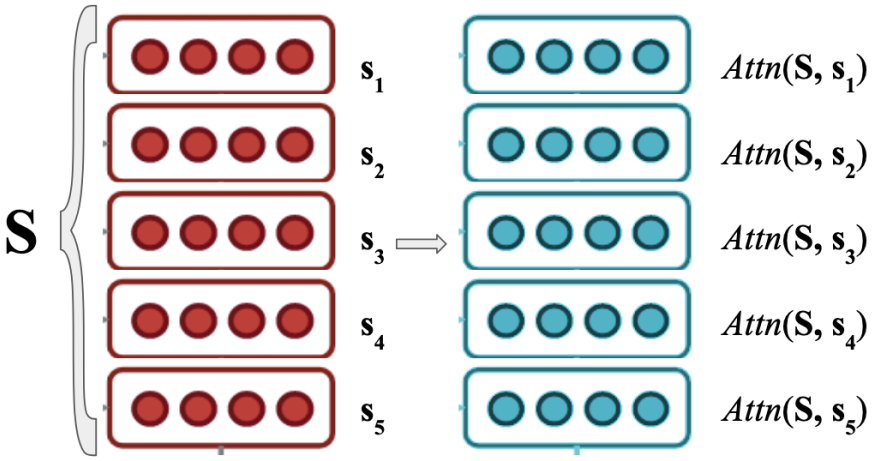
This can be hard to grasp at first, but it helps to think about attention as a *function*. This function takes a sequence S and a query x, and outputs a *weighted average* of the elements of S, where the weights for each s_i are determined by their relevance to x (for now, measured by a dot product).

$$Attn(S, x) = \sum_{i=1}^{|S|} w_i s_i$$

$$\text{...where } \vec{w} = Softmax\left(Sx\right)$$

Note: In these equations, we treat S as a $n \times d$ matrix, where n is the sequence length, and d is the embedding dimension. (Visually, the sequence looks like a stack of vectors.)

As before, weights $w_1, w_2, \ldots, w_{|S|}$ are computed first by taking the dot product of x with each element of S, and then the elements of S are averaged using these weights. Self-attention is just a specific case where x itself is an element of S. To compute self-attention for an entire sequence, we just compute $Attn(S, s_i)$ for all $s_i \in S$.



Reflexively applying the attention function to a sequence S.

To "vectorize" this computation, we can compute the weights ($w_{ij} = s_i \cdot s_j$) for all pairs of elements in S at once with one matrix multiplication, and normalize row-wise with a softmax:

$$W_{Attn} = Softmax(SS^\top)$$

The ij-th entry of the $n \times n$ matrix W_{Attn} represents the attention weight between s_i and s_j. For example, for the toy sentence "Hello I love you," we would compute the weight (relevance) of each word to each other word, giving the weight matrix below.

	Hello	I	love	you
Hello	0.8	0.1	0.05	0.05
I	0.1	0.6	0.2	0.1
love	0.05	0.2	0.65	0.1
you	0.2	0.1	0.1	0.6

Simple self-attention matrix.

Then, these weights are used in a weighted average to transform an input sequence $(s_1, s_2, \ldots, s_n)$ into an equally-sized sequence $(z_1, z_2, \ldots, z_n)$, where each output z_i is an attention-weighted average of the elements of S, based on their relevance to s_i. (In our running example, notice that each row of the matrix sums to 1; each *row* is a set of weights used to average the sequence.)

Vectorized, this looks like multiplying the matrices W_{Attn} and S so that each output element z_i is equal to a weighted average of s_i's, weighted by the relevant row of W_{Attn}.

$$Out = W_{Attn}S$$

What is this doing, exactly? Intuitively, by applying attention to a sentence, the embedding (”meaning”) of a word or sequence element “absorbs” some information from its neighbors, creating a richer representation that is influenced by the context.

Fancier Self-Attention: Queries, Keys, and Values

One thing you might think is missing from the previous exposition is the opportunity for *learning*. As laid out above, self-attention is a deterministic (though complicated!) function of the input sequence S, with no trainable parameters. Sure, the gradients can propagate through one or more attention layers back to the original (trainable) word embeddings, but the attention layers themselves are not parameterized, so they can’t specialize or learn anything to solve the specific task at hand. Using just a dot product to measure similarity is a crude approach as well—the network can’t *learn* which things should be similar, it is hard coded into the embeddings.

The introduction of queries, keys, and values (introduced in *Attention is All You Need*) brings trainable parameters into the attention layer, allowing it to adapt as the neural network trains. To understand the motivation for queries, keys, and values, recognize that in a self-attention layer, each input element s_i plays several roles:

1. It acts as the **query** x that all elements in the sequence are compared to to assess their similarity, to compute attention weights in $Attn(S, x)$.
2. It acts as the **key**, as an element of S, when its similarity is compared to the query x, also to compute the attention weight between itself and x.
3. It acts as the **value** for averaging, i.e. the s_i in $\sum_i w_i s_i$.

It makes sense (in the *self-attention* setting) to allow these three to be different vectors, and to be learned rather than fixed. This allows the model to flexibly "learn" how to copy information from one position in the sequence to another. To do this, we modify self-attention by using three different linear projections of s_i for these three different roles. The weights (W_Q, W_K, W_V) of these projections are learned as the model is trained.

$$
\begin{aligned}
q_i &= W_Q s_i \\
k_i &= W_K s_i \\
v_i &= W_V s_i
\end{aligned}
$$

The self-attention computation is the same as before, but with these values replacing s_i for each of its roles. Here is what the vectorized computation would look like for the whole sequence S:

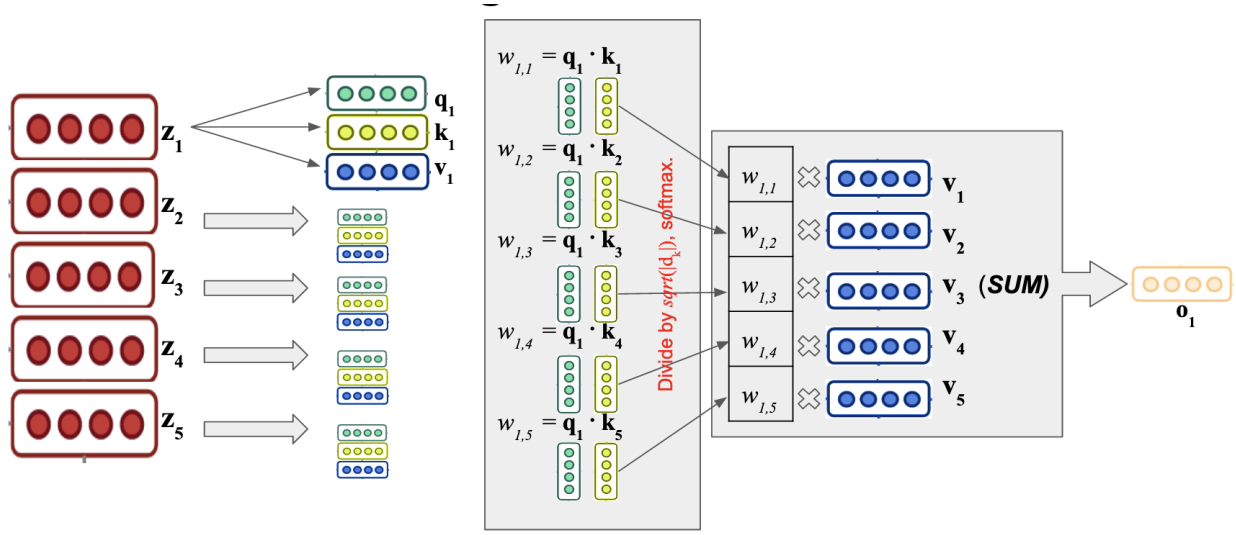
$$Q = SW_Q,\ K = SW_K,\ V = SW_V$$

$$W_{Attn} = Softmax\left(\frac{QK^\top}{\sqrt{d_k}}\right)$$

$$Attn(S, S) = W_{Attn} V$$

Note: Vaswani et al. also introduce the idea of dividing $QK^\top$ by a constant factor $\sqrt{d_k}$, the square root of the dimension of queries, keys, and values, in order to stop vanishing gradients when the embedding dimension is large.

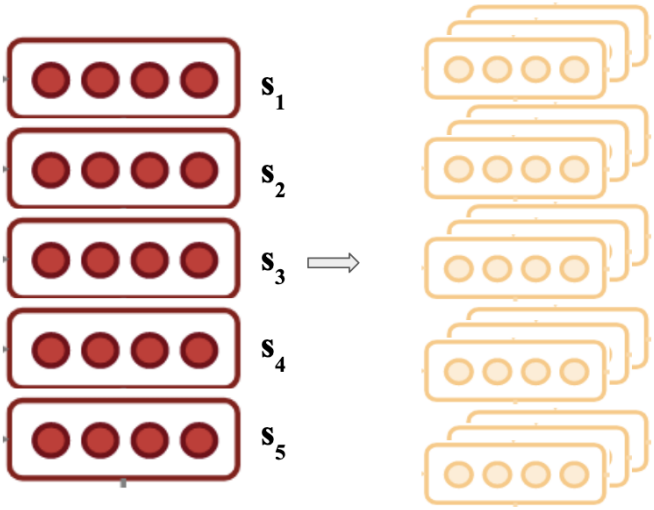
The whole process is summarized below, mostly zoomed in on the self-attention computation for the first element of the input sequence (here, z_1), mapping it all the way to the first element of the output sequence (here, o_1).



Multi-Head Attention

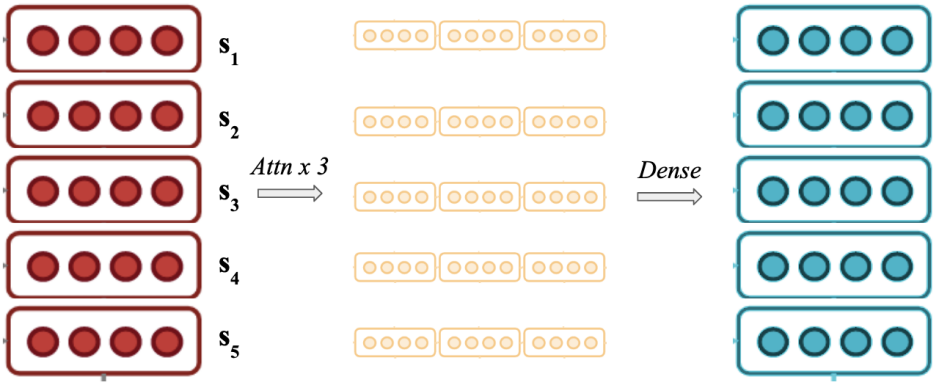
The fancier query-key-value attention outlined above is great! But, we might still have complaints—with only one set of (learned) weights for projecting all tokens into queries, keys, and values, we have a bit of a one-size-fits-all solution. It's also difficult for a query to attend to multiple positions at the same time, since with softmax, the largest value dominates (hence the "max"). To introduce *even more* flexibility, and allow more "representational subspaces" in the self-attention layer, we *repeat* the attention computation multiple times, with a different set of W_Q, W_K, W_V. (In practice, these multiple attention computations can all happen in parallel.)

Each iteration of the attention computation is called an *attention head*. This means instead of just one output, a self-attention layer with *h* heads produce a *stack* of *h* different outputs. Since each set of weights is different, each output in the stack is different, and can attend to a different set of interrelationships among the tokens in the sequence.



Attention computation with three heads.

Afterwards, the stack of outputs is concatenated and projected back to the embedding dimension d. (We often want to apply many self-attention layers one after the other, and don't want the output size to grow exponentially.) It is also common to have d_k (the dimension of projected keys, queries, and values) smaller than d, so that introducing multi-head attention does not make the model much larger and slower. Often, $d = d_k * h$ (where h is the number of attention heads) but this is not strictly necessary.



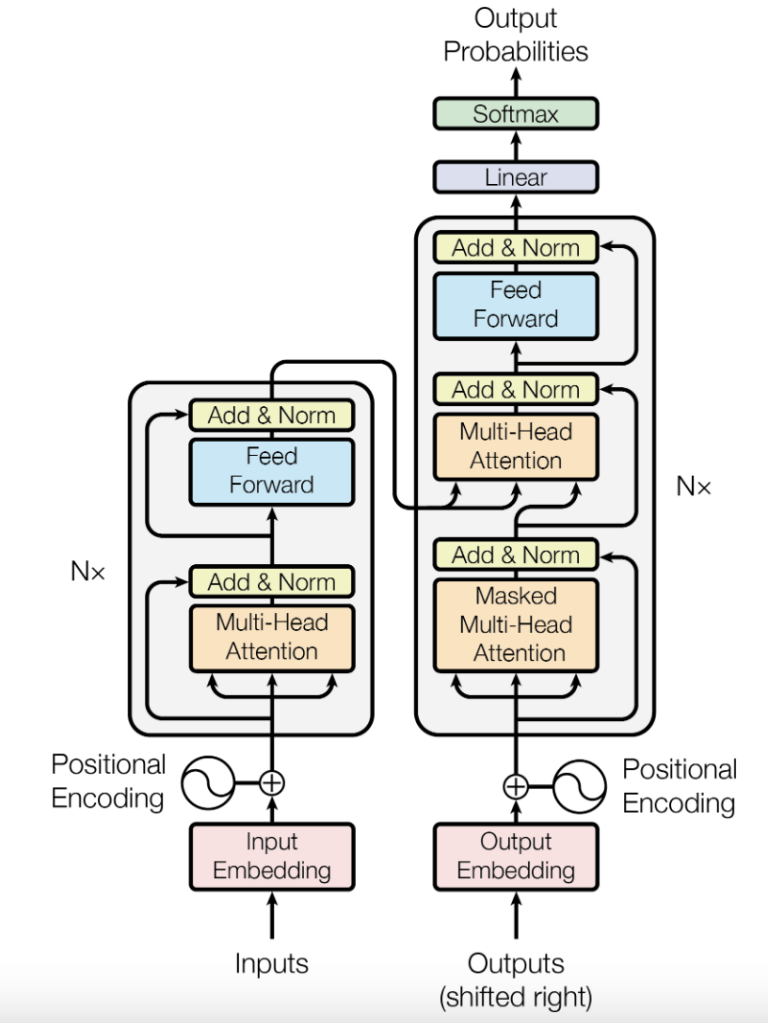
Concatenating and projecting the result to match the input dimension.

That's nearly all you need to know about attention to understand the most basic (encoder-only) Transformer. There are a few extra tricks used in the Transformer decoder, which we'll talk about later. But the backbone, and the source of the main benefits of the Transformer, is self-attention, which allows *all pairs of sequence items* to interact while processing inputs in parallel,

overcoming many weaknesses of the previous generation of models, and resulting in much better performance on GPUs, which are optimized for many parallel matrix multiplications.

The Transformer Encoder

The original Transformer was designed for neural machine translation, i.e. to translate a sentence from one language to another. Though attention is the backbone of the model, it doesn't quite get us there on its own. There are a few other building blocks that make up the Transformer (and are used, in various forms, in pretty much all Transformer-y models like BERT and GPT-3).



Transformer architecture. Source: Vaswani et al.

Positional Encodings

Unlike **recurrent neural networks** they've largely replaced, where tokens are fed to the model one at a time, transformers digest a sequence all at once. And unlike **convolutional neural networks**, which employ sliding windows to compute functions of "nearby" values, there is no notion of "local" relationships, as the Transformer computes self-attention of *all input vectors* with respect to *all input vectors* for the entire sequence, and then summarizes the result. This is why it is *permutation-invariant:* if you shuffle the input, you'd get back a shuffled output, but otherwise identical, output. This poses a problem, since the order of words in a sentence actually matters. "The dog bit the man" *means something different* than "the man bit the dog." Because the Transformer doesn't really have a notion of order, we have to *add information* to give the model a hint about the order of tokens in the input sequence: a **positional encoding**.

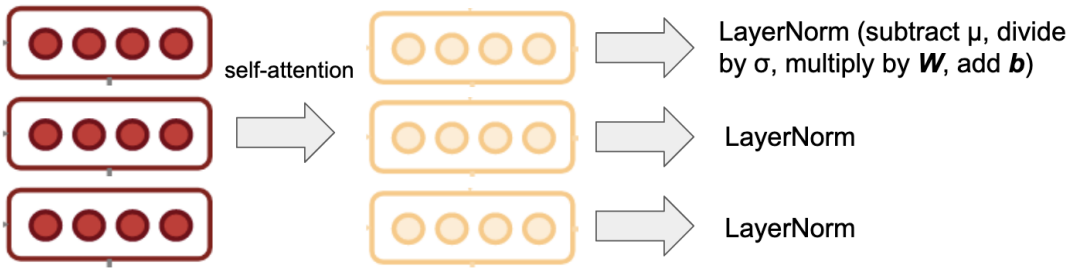
In practice, the way this is done in the original *Attention is All You Need* paper is with periodic functions of various wavelengths. For each position $(1, 2, \ldots, n)$ we compute a series of sine and cosine functions to get position vectors $(\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_n)$. Certain mathematical properties of these periodic functions make them useful for allowing the model to attend to relative positions. (If you're interested, you can read a much more detailed explanation here.) These vectors are then concatenated or added to the input sequence $(x_1, x_2, \ldots, x_n)$. Positional encoding only happens once, right before the input sequence (i.e. a sequence of word embedding vectors) is fed to the model.

An alternative approach is to use a learnable **positional embedding**, which is also added to the word embeddings before they are fed to the model. Since the original Transformer paper, many other ways of informing the model of absolute or relative positions of tokens have been proposed, but those details are outside the scope of this memo.

Layer Normalization

Various forms of normalization and standardization are common in machine learning in order to improve learning. Historically, inputs to machine learning models would be normalized, standardized, or whitened before being fed to the model. A more recent proposal is to *explicitly include normalization* at multiple stages of neural network architectures, rather than simply as a data preprocessing step. Because normalization (subtracting a mean, dividing by standard deviation) is a mathematical operation like any other, we can compute gradients and back-propagate through normalization layers. This helps training by reducing *internal covariate shift*, i.e. the shifting distributions of values at each layer as parameters update, which can cause problems with gradients and saturating/dying activation functions. Normalization helps networks train faster, and even provides some regularization.

BatchNorm proposes normalizing the *same neuron* across a batch of inputs. LayerNorm averages in the other direction, i.e. across all neurons in a layer for a single training example. This has some advantages over BatchNorm: it is agnostic to batch size, and it can be used in recurrent architectures. In the Transformer, it is only used to average across the embedding dimension (i.e. we normalize each element $s_1, s_2, \ldots, s_n$ in the input sequence S separately). After normalization, there is a linear layer (i.e. multiply each neuron by a learned weight **W** and add a bias **b**).



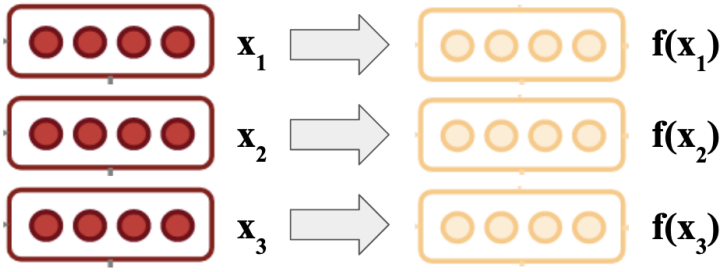
More recently, state-of-the-art architectures have omitted the bias **b**, and some have even argued for a lightweight variant of LayerNorm that normalizes without centering (RMSNorm).

Dropout

Dropout is a regularization technique for neural networks that zeroes out neurons with some small probability during training. This results in a model that is more robust—it can't rely too heavily on any particular neuron, and it prevents neurons from "co-adapting" and relying too heavily on each other. This prevents the model from overfitting, and basically gives rise to a neural network that, at test time, is equivalent the average of an "ensemble" of different neural networks. Dropout is applied at various points in the Transformer for regularization.

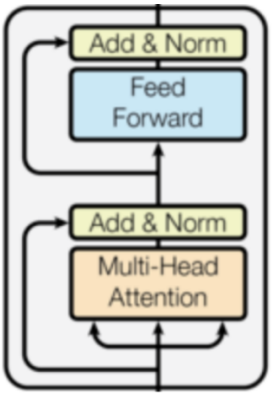
Position-Wise Feed Forward Network

Each Transformer self-attention layer is followed by a "position-wise" feed-forward network, which is a fancy way of saying the *same small neural network* is applied to each sequence element $s_1, s_2, \ldots, s_n$. Each layer of the Transformer has its own separate position-wise feed-forward network, but the same network is shared across sequence elements (i.e. the first word in a sentence has the same feed-forward network applied to it as the last word).



All Together: The Encoder Block & Transformer Encoder

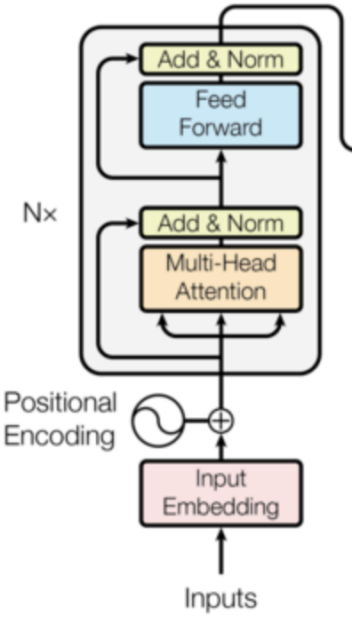
A Transformer encoder block is just a combination of two sublayers: multi-head self-attention, followed by the position-wise feed-forward network. After each sublayer, a *residual connection* is employed, which means that the input of the sublayer is added back to the output. This approach (originating from the ResNet paper) helps with the problem of vanishing gradients, and allows information to flow more easily between layers, allowing neural networks to get much deeper without becoming impossible to train. This is followed by LayerNorm (together, "Add & Norm"). Dropout is used before each Add & Norm step.



It has also become common to use dropout *within* self-attention layers (randomly drop some units after the softmax), but this is not mentioned in the original Transformer paper. Newer Transformer variants also often use a "pre-norm" (LayerNorm before the sublayer, rather than after), which tends to make training a bit less finicky. (Vaswani et al. have to use a whole bag of tricks like learning rate warmup to make their model improve stably.)

The Transformer encoder is one half of the Transformer model—the half which maps input sequences to an embedding, which can then be used to *output* a translated sequence by the decoder. The encoder comprises the embedding step and positional encoding for the input

sequence, followed by multiple Transformer encoder blocks (as laid out above). The original Transformer paper uses 6 blocks in the encoder.



The Transformer Decoder

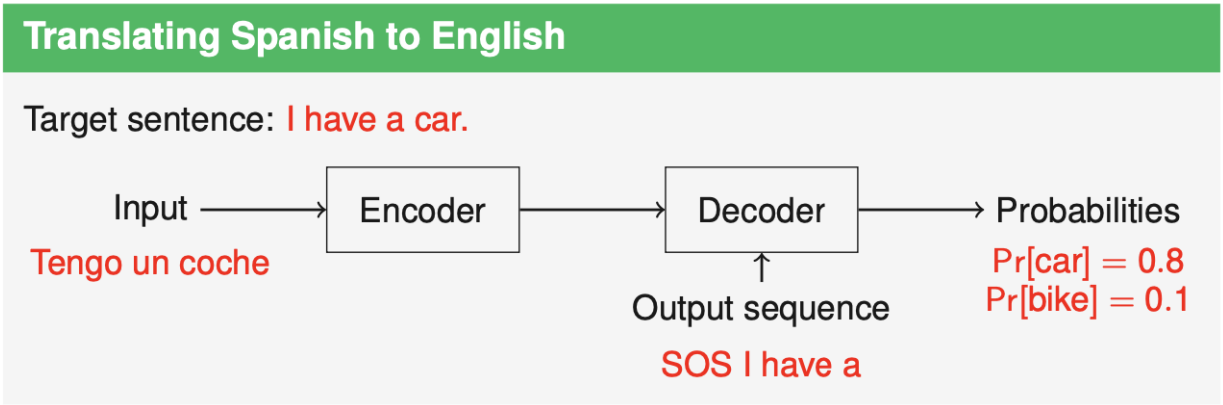
N.B.—This section is a bit complicated, and you can skip it and still come away with decent knowledge of self-attention, enough to understand how the Transformer is adapted for tabular data. However, the decoder is relevant if you want to fully understand how the original Transformer is used as a sequence-to-sequence model (i.e. for machine translation), or understand autoregressive large language models based on the Transformer decoder, such as OpenAI's GPT-3.

Decoding: Training vs. Inference

Recall that the goal of the Transformer is sequence-to-sequence translation. Once a sequence has passed through the Transformer encoder, we have an output that is a representation of the input sequence, $Z = (z_1, z_2, \ldots, z_n)$. That is the *encoding* of the sequence. To get a legible *output* sequence, we now have to *decode* this hidden representation.

During **inference or deployment**, we don't have a target sentence (we don't know the answer in advance), and generate the output one word at a time. At each decoding step, the decoder receives: (1) The encoding of the input sequence, and (2) The output sequence generated so far. The decoder outputs a probability distribution over the next word, and we either sample from it,

or greedily choose the next word, and add that to the output sequence so far. At the first step, we only pass a [start of sequence] token. *This means to generate a sequence of length k, we have to run the decoder k times!*



Source: Lennart Sverson's slides on Transformer models.

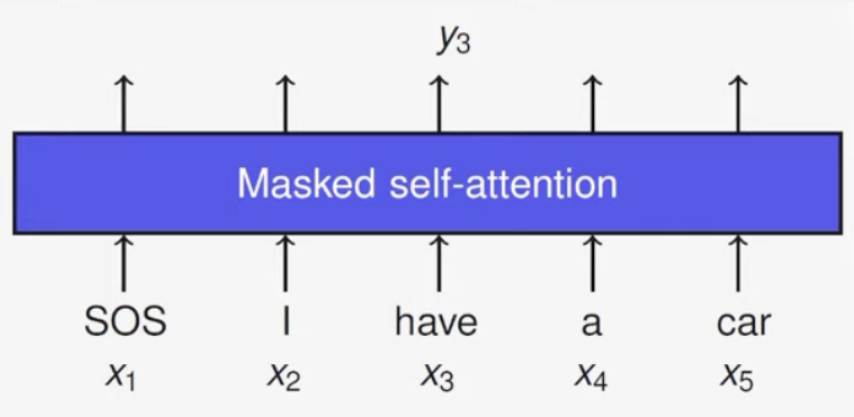
During **training**, the main difference is that we already know the target sentence, and we want to encourage the model to produce that sentence. We don't feed the decoder *its own* output to get the next word like during deployment; instead we feed it *the right answer so far*, and make it guess the next word (This is called **teacher forcing**). Suppose the correct translation is "I love you a lot", and we give the network the correct answer so far, which might be "I love." The network predicts the next word as "myself", incorrectly. Rather than feeding in "I love myself" to get the next word, we just feed in the correct answer, "I love you".

Doing this naïvely, training takes as many decoding steps as deployment: we have to run the decoder k times to train on a target sentence of length k, giving it more of the correct sentence with each iteration, and making it predict the next word. Meanwhile, the decoder architecture (which, like the encoder, takes a sequence $S = (s_1, s_2, \ldots, s_k)$ and maps it to an equal-length sequence) is producing output that is mostly discarded. What a waste! However, with the help of one simple trick (masked self-attention), the architecture of the Transformer allows all of these steps to happen in parallel—we can feed the entire target sequence to the decoder during training, and use a mask on the attention matrix to stop it from "cheating" and looking ahead. This mechanism, *masked multi-head self-attention* (or *causal attention*) is explained in the following section.

Masked Multi-Head Self-Attention

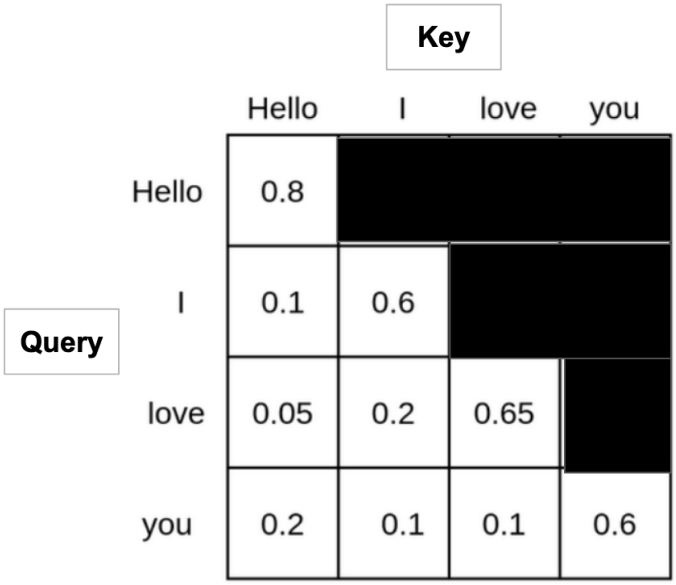
The decoder receives two inputs: (1) the target sequence, and (2) the encoded input sequence (output by the encoder). We want the output of the decoder $Z = (z_1, z_2, \ldots, z_k)$ to correspond to a valid set of k next-token predictions for the target sequence $T = (t_1, t_2, \ldots, t_k)$, so that $z_1 = t_2, z_2 = t_3$ and so on. If we just used normal self-attention, the decoder could just *copy* t_2 from the input and correctly output that $z_1 = t_2$, and do the same for all z_i, instead of learning to predict unseen words.

Masked multi-head self-attention insists that, during the attention computation (which creates a new representation for each t_i based on a weighted average of the whole sequence), only tokens *before* t_i can contribute information. All tokens *after* t_i must not contribute, which means they must have attention weight of 0. For instance, in the example below, the output y_3 (which aims to predict x_4) should only depend on $x_{1:3}$, and not on x_4 or x_5.



Source: Lennart Sverson's slides on Transformer models.

Once this objective is understood, masked self-attention is easy to implement: attention weights are computed as normal, but before the softmax, all attention weights that would let information flow from a later token to an earlier token are zeroed out (to be precise, set to $-\infty$). The result looks something like the simple attention matrix below: for a given query, it can only have a non-zero attention weight for keys that correspond to *itself*, and *earlier tokens*. This means that when this attention matrix is used to compute a weighted average of *values*, the representation for a given position (e.g. the word "love") is only an average of the values for earlier tokens ("Hello" and "I"), and itself.



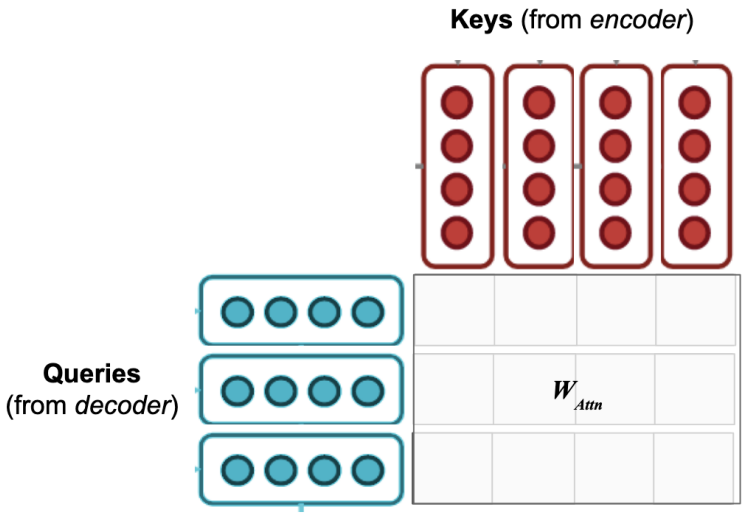
After zeroing out the values, the softmax will normalize the weights to ensure they add up to 1 (which they do not in the illustration above). This masked self-attention is all that is needed to allow parallelism during training. During testing/deployment, it also makes sense to use the mask, since any sequence elements *after* the one you are trying to predict haven't been output yet, and are unknown/undefined.

Encoder-Decoder Attention

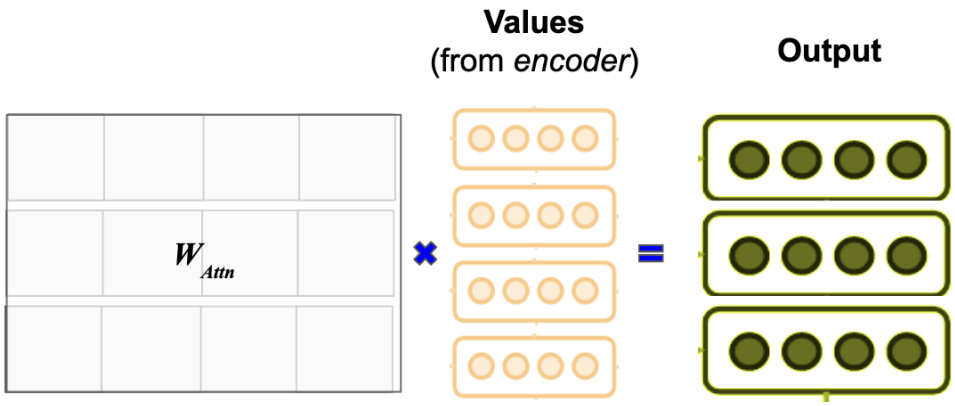
So far, we've only talked about the part of the decoder that processes the *target sequence* (during training), or the *generated sequence so far* (during deployment). The last obvious piece is that the decoder must digest the information about the *input sequence* from the encoder in order to translate it! This is also done using attention—not unlike how it was used in recurrent models by Bahdanau et al., where the representation of the input sequence is collapsed into a dynamic weighted average of encoder states, based on their relevance to the current decoding step.

This attention block takes in two inputs: (1) the output of the previous decoder layer; and (2) the final encoder output. (These sequences may have different lengths—for example, *te amo* in Spanish translates to *I love you* in English, so the encoder sequence would have length 2, and the target sequence would have length 3. However, both encoder and decoder work with sequences of tokens that have the same embedding dimension d_{model}.) The multi-head attention computation is much the same, but unlike the self-attention computations outlined above, the

queries come from the decoder sequence, and the *keys* and *values* come from the encoder. For each decoder token $t_1, t_2, \ldots, t_m$, we use its queries compute its attention weights relative to keys for each encoder token $e_1, e_2, \ldots, e_n$, giving a $m \times n$ attention-weight matrix, which captures the relevance of each encoder token to each decoder token.

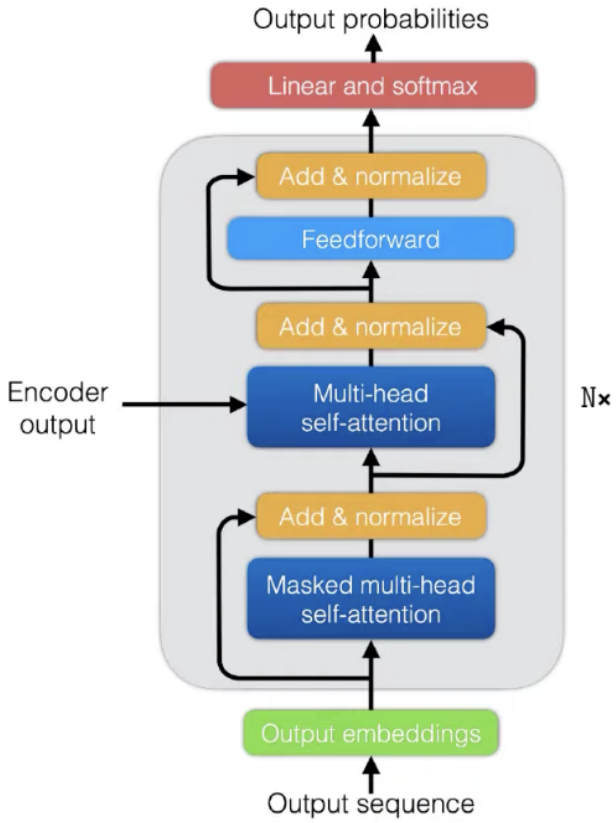


Using the attention matrix, we compute a weighted average of the encoder state *values* $v_1, v_2, \ldots, v_n$, for each decoder token, weighted by their relevance to that decoder token. The result is a new sequence of length m (matching the dimensions of the *decoder*), comprised of information from the *encoder* (the decoder state only affects the weights). When this is added and normed with the previous decoder state, information flows from the encoding of the input sequence into the decoder, with attention paid to the most relevant parts of the input sequence.



We don’t have to worry about masking here, because the decoder is supposed to know/understand the entire input sequence—there’s nothing to hide there. And no decoder states interact with one another during this attention computation—only with the encoder states.

Putting it All Together: The Decoder Block & Transformer Decoder



Source: Lennart Sverson's slides on Transformer models.

A Transformer decoder block is a combination of three sublayers: masked multi-head self-attention, encoder-decoder attention, and finally a position-wise feed-forward network (as in the encoder). Just like the encoder block, there is a residual connection, LayerNorm, and dropout after each sublayer. The decoder is just a stack of N decoder blocks. Each one takes two inputs: (1) the encoder output (which is the same for all N decoder blocks), and (2) the output of the previous decoder block (or the target sequence for the very first decoder block). Finally, at the end, a fully connected (linear) layer is used to project the decoder outputs so that they are the size of the vocabulary, and then a softmax is applied to get *probabilities* of the next token at each position.

Using Transformers for Self-Supervised Pretraining

The original Transformer was designed for a specific task (machine translation), and was trained on the appropriate data (pairs of sequences from different languages), to learn embeddings and weights to optimize performance on that task. However, later research leveraging transformers introduced the idea of *self-supervised pretraining*: learning from a large amount of unlabeled

data on a “pretext” task, and then fine-tuning the model on labeled data for a downstream task (such as sentiment classification).

Transfer learning had been popular in computer vision for years (many tasks were improved by transferring a model trained on ImageNet), and was used to a limited extent in natural language processing (mostly pretraining word embeddings). ULMFiT (2018) was the first model to successfully leverage self-supervised pretraining (with an LSTM model rather than a transformer), but it was the incredible results achieved with transformers (beginning with BERT and GPT) that made self-supervised pretraining ubiquitous in the language domain.

BERT (2018)

BERT, and later BERT-like architectures, use only the *encoder* part of the original Transformer model. This means there is no masked attention, so every word in the output can be influenced by every word in the input—there’s no *autoregressive* property. BERT is trained on a masked-language-modeling objective (predicting words that are masked out in the input sentence), and a next-sentence-prediction task (predicting if Sentence B followed Sentence A in the original document). These are both semi-supervised tasks that can easily be done on a large amount of free-text data, without any intensive manual labeling. As a result, BERT can be trained on gigabytes of data and learn very informative representations for words and sequences, that can then be fine-tuned on downstream tasks. Since BERT is designed to digest a whole sequence without masking (i.e. attention can flow in both directions), it is generally used tasks that involve understanding the sequence (i.e. classification, summarization) rather than predicting the next token (that’s where GPT excels).

GPT (2018)

GPT, and its newer (larger) cousins GPT-2 and GPT-3, use only the *decoder* part of the Transformer (omitting encoder-decoder attention, since there’s no encoder). As a consequence, a given word ingested by GPT can only attend to words that came before it. This is important, because the pretext task for GPT is *next-token prediction* (also called *causal language modeling*). The model is fed tons of unlabeled text, and its task is to predict the next unseen token. GPT is therefore a great model for generating text in response to some prompt, since all you have to do is have it repeatedly predict the next token. Newer versions of GPT have been shown to be effective at zero-shot and few-shot NLP tasks, simply by virtue of learning how to predict the next token.