TRANSFORMERS FROM SCRATCH

18 Aug 2019 · code on github · video lecture

Transformers are a very exciting family of machine learning architectures. Many good tutorials exist (e.g. [1, 2]) but in the last few years, transformers have mostly become simpler, so that it is now much more straightforward to explain how modern architectures work. This post is an attempt to explain directly how modern transformers work, and why, without some of the historical baggage.

I will assume a basic understanding of neural networks and backpropagation. If you'd like to brush up, this lecture will give you the basics of neural networks and this one will explain how these principles are applied in modern deep learning systems.

A working knowledge of Pytorch is required to understand the programming examples, but these can also be safely skipped.

Self-attention

The fundamental operation of any transformer architecture is the *self-attention operation*.

> *We'll explain where the name "self-attention" comes from later. For now, don't read too much in to it.*

Self-attention is a sequence-to-sequence operation: a sequence of vectors goes in, and a sequence of vectors comes out. Let's call the input vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$ and the corresponding output vectors $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_t$. The vectors all have dimension k.

To produce output vector $\mathbf{y}_i$, the self attention operation simply takes *a weighted average over all the input vectors*

$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{x}_j \ .$$

Where j indexes over the whole sequence and the weights sum to one over all j. The weight w_{ij} is not a parameter, as in a normal neural net, but it is *derived* from a function over $\mathbf{x}_i$ and $\mathbf{x}_j$. The simplest option for this function is the dot product:

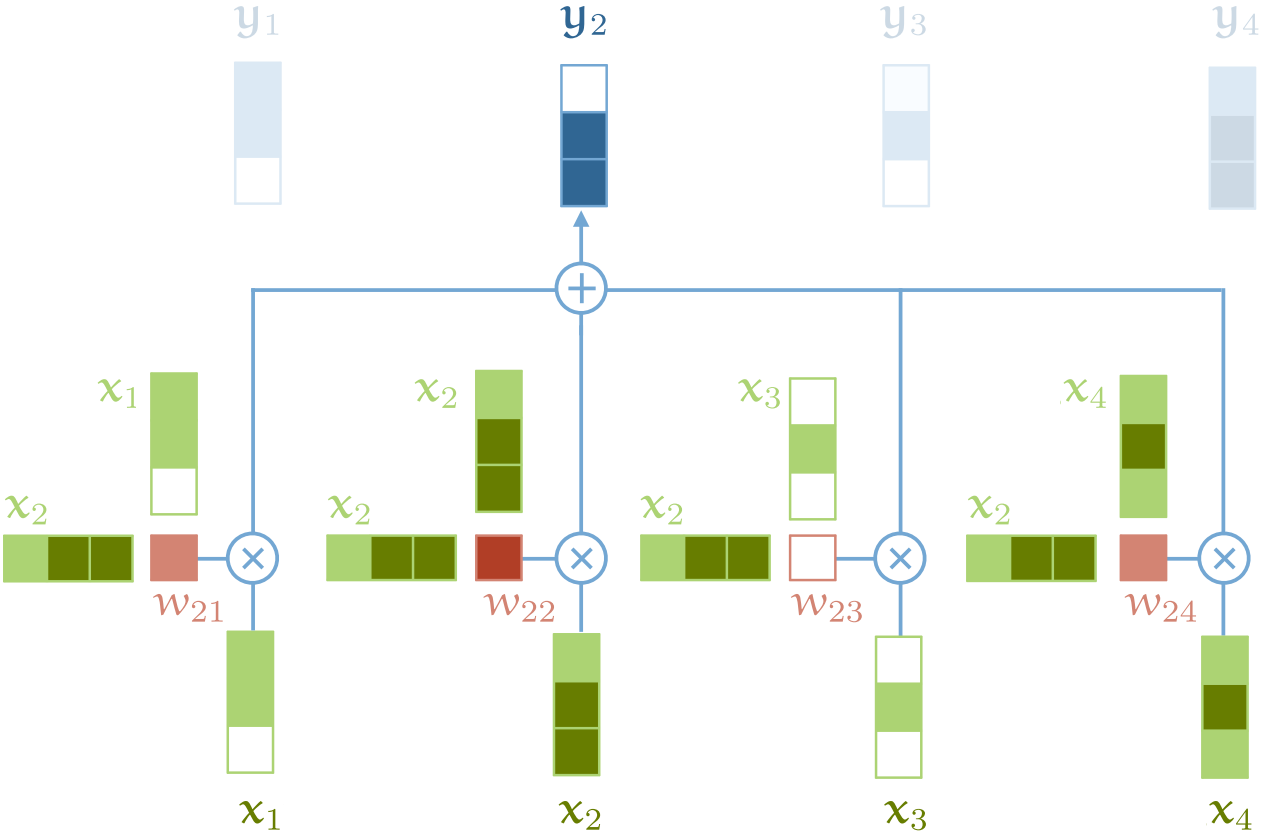
$$w'_{ij} = \mathbf{x}_i^T \mathbf{x}_j \ .$$

Note that $\mathbf{x}_i$ is the input vector at the same position as the current output vector $\mathbf{y}_i$. For the next output vector, we get an entirely new series of dot products, and a different weighted sum.

The dot product gives us a value anywhere between negative and positive infinity, so we apply a softmax to map the values to $[0, 1]$ and to ensure that they sum to 1 over the whole sequence:

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_j \exp w'_{ij}} .$$

And that's the basic operation of self attention.



A visual illustration of basic self-attention. Note that the softmax operation over the weights is not illustrated.

A few other ingredients are needed for a complete transformer, which we'll discuss later, but this is the fundamental operation. More importantly, this is the only operation in the whole architecture that propagates information *between* vectors. Every other operation in the transformer is applied to each vector in the input sequence without interactions between vectors.

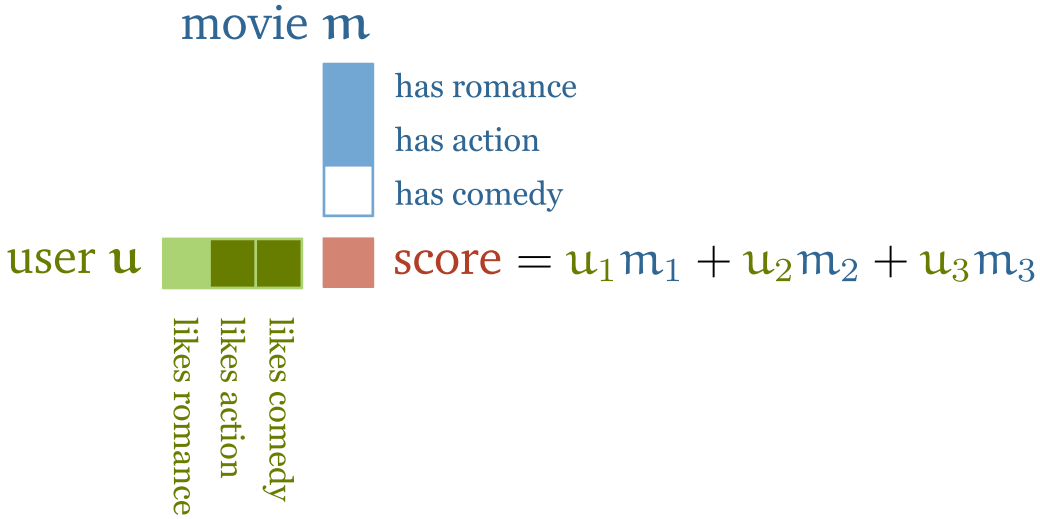
Understanding why self-attention works

Despite its simplicity, it's not immediately obvious why self-attention should work so well. To build up some intuition,

let's look first at the standard approach to *movie recommendation*.

Let's say you run a movie rental business and you have some movies, and some users, and you would like to recommend movies to your users that they are likely to enjoy.

One way to go about this, is to create manual features for your movies, such as how much romance there is in the movie, and how much action, and then to design corresponding features for your users: how much they enjoy romantic movies and how much they enjoy action-based movies. If you did this, the dot product between the two feature vectors would give you a score for how well the attributes of the movie match what the user enjoys.



If the signs of a feature match for the user and the movie—the movie is romantic and the user loves romance or the movie is *unromantic* and the user hates romance—then the

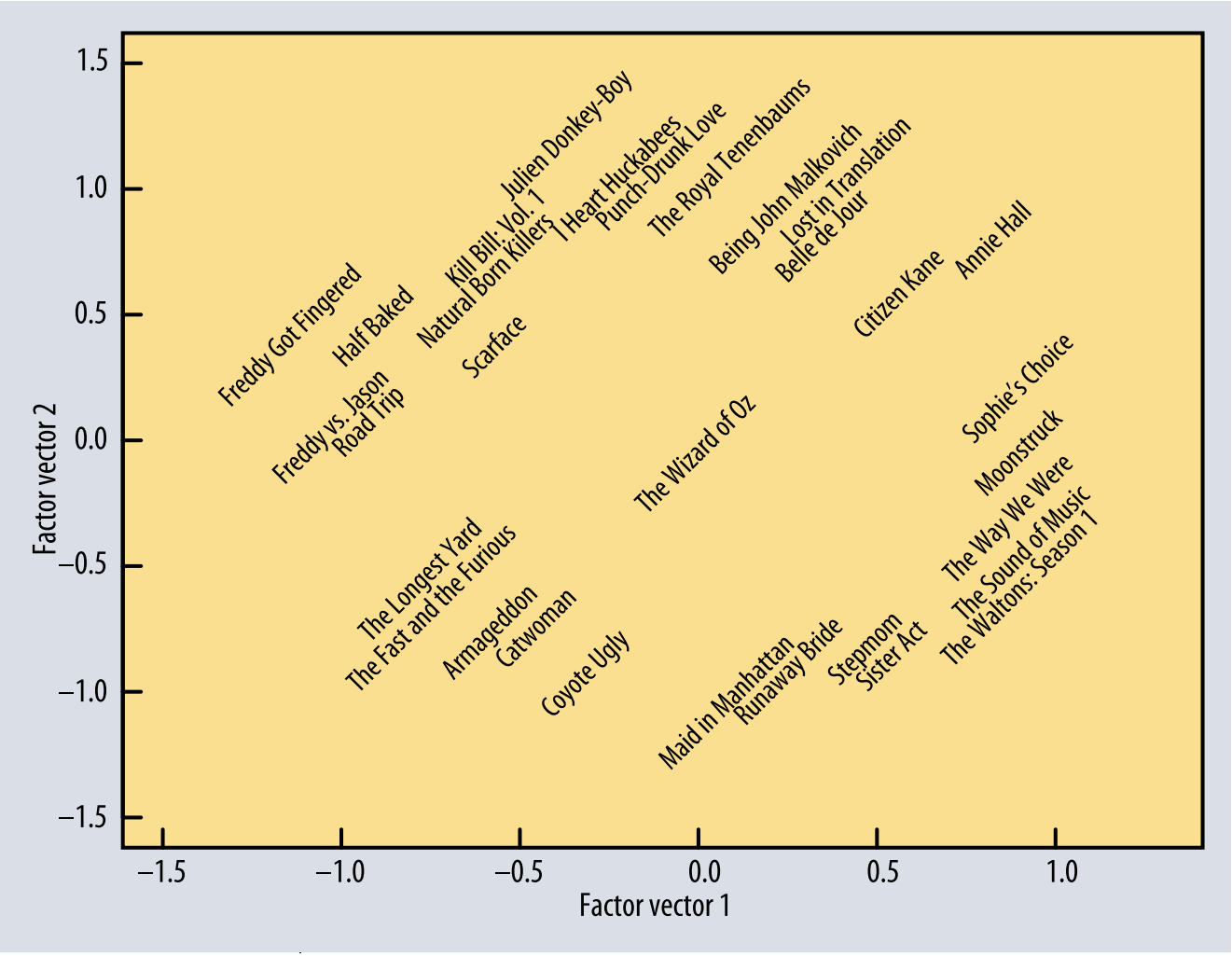
resulting dot product gets a positive term for that feature. If the signs don't match—the movie is romantic and the user hates romance or vice versa—the corresponding term is negative.

Furthermore, the *magnitudes* of the features indicate how much the feature should contribute to the total score: a movie may be a little romantic, but not in a noticeable way, or a user may simply prefer no romance, but be largely ambivalent.

Of course, gathering such features is not practical. Annotating a database of millions of movies is very costly, and annotating users with their likes and dislikes is pretty much impossible.

What happens instead is that we make the movie features and user features *parameters* of the model. We then ask users for a small number of movies that they like and we optimize the user features and movie features so that their dot product matches the known likes.

Even though we don't tell the model what any of the features should mean, in practice, it turns out that after training the features do actually reflect meaningful semantics about the movie content.



The first two learned features from a basic matrix factorization model. The model had no access to any information about the content of the movies, only which users liked them. Note that movies are arranged from low-brow to high-brow horizontally, and from mainstream to quirky vertically. From [4].

> *See this lecture for more details on recommender systems. For now, this suffices as an explanation of how the dot product helps us to represent objects and their relations.*

This is the basic principle at work in the self-attention. Let's say we are faced with a sequence of words. To apply self-attention, we simply assign each word t in our vocabulary an *embedding vector* $\boldsymbol{v}_t$ (the values of which we'll learn). This is what's known as an *embedding layer* in sequence modeling. It turns the word sequence the, cat, walks, on, the, street into the vector sequence

$$\boldsymbol{v}_{\text{the}}, \boldsymbol{v}_{\text{cat}}, \boldsymbol{v}_{\text{walks}}, \boldsymbol{v}_{\text{on}}, \boldsymbol{v}_{\text{the}}, \boldsymbol{v}_{\text{street}} .$$

If we feed this sequence into a self-attention layer, the output is another sequence of vectors

$$\boldsymbol{y}_{\text{the}}, \boldsymbol{y}_{\text{cat}}, \boldsymbol{y}_{\text{walks}}, \boldsymbol{y}_{\text{on}}, \boldsymbol{y}_{\text{the}}, \boldsymbol{y}_{\text{street}}$$

where $\boldsymbol{y}_{\text{cat}}$ is a weighted sum over all the embedding vectors in the first sequence, weighted by their (normalized) dot-product with $\boldsymbol{v}_{\text{cat}}$.

Since we are *learning* what the values in $\boldsymbol{v}_t$ should be, how "related" two words are is entirely determined by the task. In most cases, the definite article the is not very relevant to the interpretation of the other words in the sentence; therefore, we will likely end up with an embedding $\boldsymbol{v}_{\text{the}}$ that has a low or negative dot product with all other words. On the other hand, to interpret what walks means in this sentence, it's very helpful to work out *who* is doing the walking. This is

likely expressed by a noun, so for nouns like cat and verbs like walks, we will likely learn embeddings $\boldsymbol{v}_{\text{cat}}$ and $\boldsymbol{v}_{\text{walks}}$ that have a high, positive dot product together.

This is the basic intuition behind self-attention. The dot product expresses how related two vectors in the input sequence are, with "related" defined by the learning task, and the output vectors are weighted sums over the whole input sequence, with the weights determined by these dot products.

Before we move on, it's worthwhile to note the following properties, which are unusual for a sequence-to-sequence operation:

- There are no parameters (yet). What the basic self-attention actually does is entirely determined by whatever mechanism creates the input sequence. Upstream mechanisms, like an embedding layer, drive the self-attention by learning representations with particular dot products (although we'll add a few parameters later).
- Self attention sees its input as a *set*, not a sequence. If we permute the input sequence, the output sequence will be exactly the same, except permuted also (i.e. self-attention is *permutation equivariant*). We will mitigate this somewhat when we build the full

transformer, but the self-attention by itself actually *ignores* the sequential nature of the input.

In Pytorch: basic self-attention

What I cannot create, I do not understand, as Feynman said. So we'll build a simple transformer as we go along. We'll start by implementing this basic self-attention operation in Pytorch.

The first thing we should do is work out how to express the self attention in matrix multiplications. A naive implementation that loops over all vectors to compute the weights and outputs would be much too slow.

We'll represent the input, a sequence of $\mathtt{t}$ vectors of dimension $\mathtt{k}$ as a $\mathtt{t}$ by $\mathtt{k}$ matrix $\mathbf{X}$. Including a minibatch dimension $\mathtt{b}$, gives us an input tensor of size $(\mathtt{b}, \mathtt{t}, \mathtt{k})$.

The set of all raw dot products w'_{ij} forms a matrix, which we can compute simply by multiplying $\mathbf{X}$ by its transpose:

```
import torch
import torch.nn.functional as F

# assume we have some tensor x with size (b, t, k)
x = ...
```

```
raw_weights = torch.bmm(x, x.transpose(1, 2))
# - torch.bmm is a batched matrix multiplication. It
#   applies matrix multiplication over batches of
#   matrices.
```

Then, to turn the raw weights w'_{ij} into positive values that sum to one, we apply a *row-wise* softmax:

```
weights = F.softmax(raw_weights, dim=2)
```

Finally, to compute the output sequence, we just multiply the weight matrix by $\mathbf{X}$. This results in a batch of output matrices $\mathbf{Y}$ of size `(b, t, k)` whose rows are weighted sums over the rows of $\mathbf{X}$.

```
y = torch.bmm(weights, x)
```

That's all. Two matrix multiplications and one softmax gives us a basic self-attention.

Additional tricks

The actual self-attention used in modern transformers relies on three additional tricks.

1) Queries, keys and values

Every input vector $\mathbf{x}_i$ is used in three different ways in the self attention operation:

- It is compared to every other vector to establish the weights for its own output $\mathbf{y}_i$
- It is compared to every other vector to establish the weights for the output of the j-th vector $\mathbf{y}_j$
- It is used as part of the weighted sum to compute each output vector once the weights have been established.

These roles are often called the **query**, the **key** and the **value** (we'll explain where these names come from later). In the basic self-attention we've seen so far, each input vector must play all three roles. We make its life a little easier by deriving new vectors for each role, by applying a linear transformation to the original input vector. In other words, we add three $k \times k$ weight matrices $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v$ and compute three linear transformations of each x_i, for the three different parts of the self attention:

$$\mathbf{q}_i = \boldsymbol{W}_q \mathbf{x}_i \qquad \mathbf{k}_i = \boldsymbol{W}_k \mathbf{x}_i \qquad \boldsymbol{v}_i = \boldsymbol{W}_v \mathbf{x}_i$$

$$
\begin{aligned}
w'_{ij} &= \mathbf{q}_i{}^\mathsf{T} \mathbf{k}_j \\
w_{ij} &= \text{softmax}(w'_{ij}) \\
\mathbf{y}_i &= \sum_j w_{ij} \mathbf{v}_j \, .
\end{aligned}
$$

This gives the self-attention layer some controllable parameters, and allows it to modify the incoming vectors to suit the three roles they must play.

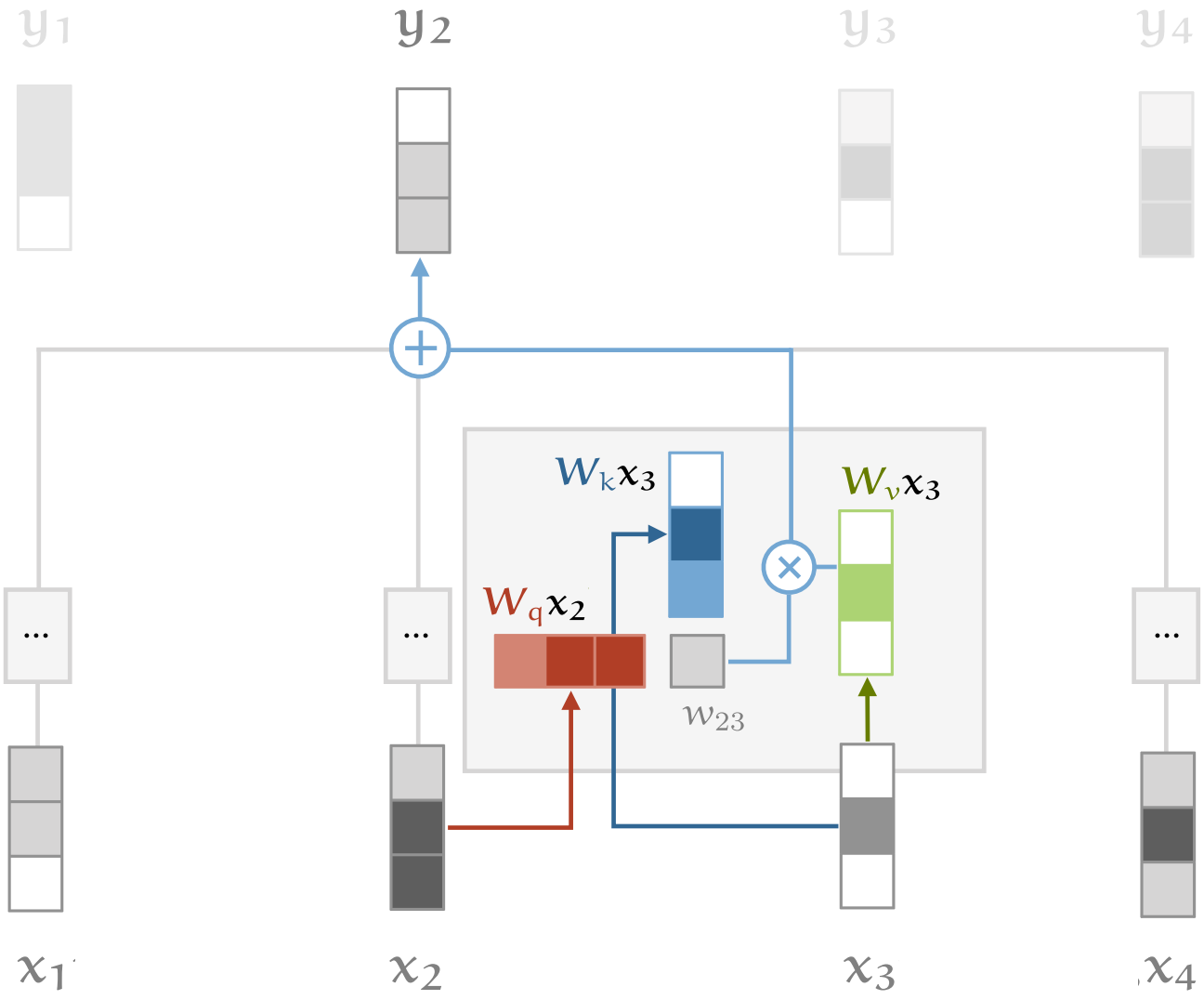


Illustration of the self-attention with key, query and value transformations.

2) Scaling the dot product

The softmax function can be sensitive to very large input values. These kill the gradient, and slow down learning, or cause it to stop altogether. Since the average value of the dot product grows with the embedding dimension k, it helps to scale the dot product back a little to stop the inputs to the softmax function from growing too large:

$$w'_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{k}}$$

> *Why $\sqrt{k}$? Imagine a vector in $\mathbb{R}^k$ with values all c. Its Euclidean length is $\sqrt{k}c$. Therefore, we are dividing out the amount by which the increase in dimension increases the length of the average vectors.*

3) Multi-head attention

Finally, we must account for the fact that a word can mean different things to different neighbours. Consider the following example.

mary, gave, roses, to, susan

We see that the word gave has different relations to different parts of the sentence. mary expresses who's doing the giving,

roses expresses what's being given, and susan expresses who the recipient is.

In a single self-attention operation, all this information just gets summed together. The inputs $\mathbf{x}_{\text{mary}}$ and $\mathbf{x}_{\text{susan}}$ can influence the output $\mathbf{y}_{\text{gave}}$ by different amounts, depending on their dot-product with $\mathbf{x}_{\text{gave}}$, but they can't influence it *in different ways*. If, for instance, we want the information about who gave the roses and who received them to end up in different parts of $\mathbf{y}_{\text{gave}}$, we need a little more flexibility.

> *This leaves aside how we figure out who gave the roses. We can do that based on prior knowledge about Mary and Susan, encoded in the embeddings. We can also look at the order of the words, but we'll look at how to achieve that later.*

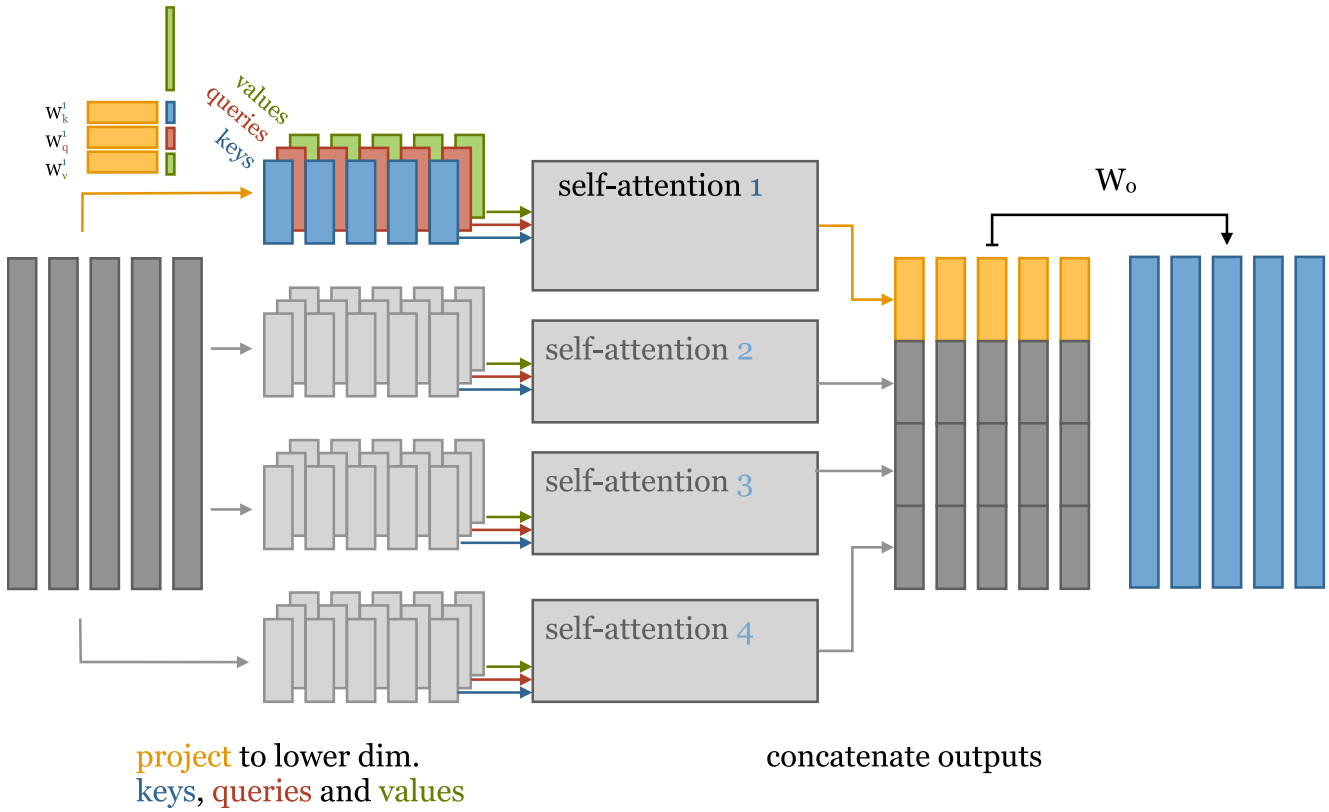
We can give the self attention greater power of discrimination, by combining several self-attention mechanisms (which we'll index with r), each with different matrices $\mathbf{W}_{\text{q}}^{r}$, $\mathbf{W}_{\text{k}}^{r}$, $\mathbf{W}_{v}^{r}$. These are called *attention heads*.

For input $\mathbf{x}_i$ each attention head produces a different output vector $\mathbf{y}_i^r$. We concatenate these, and pass them through a linear transformation to reduce the dimension back to k.

Efficient multi-head self-attention. The simplest way to understand multi-head self-attention is to see it as a small number of copies of the self-attention mechanism applied in parallel, each with their own key, value and query transformation. This works well, but for R heads, the self-attention operation is R times as slow.

It turns out we can have our cake and eat it too: there is a way to implement multi-head self-attention so that it is roughly as fast as the single-head version, but we still get the benefit of having different self-attention operations in parallel. To accomplish this, each head receives low-dimensional keys queries and values. If the input vector has $\mathsf{k} = 256$ dimensions, and we have $\mathsf{h} = 4$ attention heads, we multiply the input vectors by a 256×64 matrix to project them down to a sequence of 64 dimansional vectors. For every head, we do this 3 times: for the keys, the queries and the values.

Here is the whole process illustrated in one image.



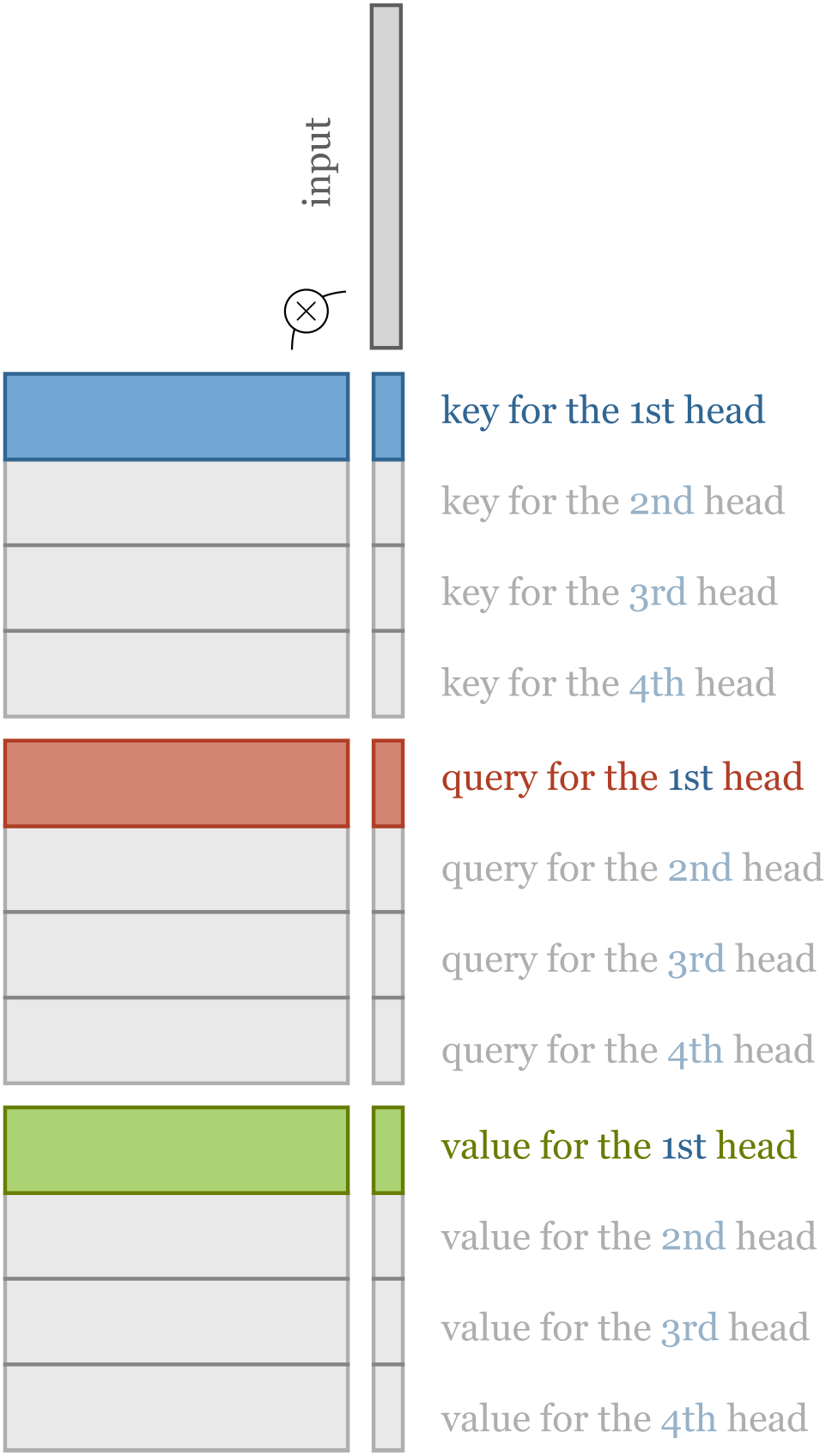
The basic idea of multi-head self-attention with 4 heads. To get our keys, queries and values, we project the input down to vector sequences of smaller dimension.

This requires $3h$ matrices of size k by k/h. In total, this gives us $3hk\frac{k}{h} = 3k^2$ parameters to compute the inputs to the multi-head self-attention: the same as we had for the single-head self-attention.

> *The only difference is the matrix W_o, used at the end of the multi-head self attention. This adds k^2 parameters compared to the single-head version. In most transformers, the first thing that happens after each self attention is a feed-forward layer, so this may not be*

strictly necessary. I've never seen a proper ablation to test whether W_o can be removed.

We can even implement this with just three $k \times k$ matrix multiplications as in the single-head self-attention. The only extra operation we need is to slice the resulting sequence of vectors into chunks.



To compute multi-head attention efficiently, we combine the computation of the projections down to a lower dimensional

representation and the computations of the keys, queries and values into three $k \times k$ matrices.

In Pytorch: complete self-attention

Let's now implement a self-attention module with all the bells and whistles. We'll package it into a Pytorch module, so we can reuse it later.

```
import torch
from torch import nn
import torch.nn.functional as F

class SelfAttention(nn.Module):
    def __init__(self, k, heads=4, mask=False):

    super().__init__()

    assert k % heads == 0

    self.k, self.heads = k, heads
```

Note the assert: the embedding dimension needs to be divisible by the number of heads.

Next, we set up some linear transformations with `emb` by `emb` matrices. The `nn.Linear` module with the bias disabled gives us such a projection, and provides a reasonable initialization for us.

```
        # These compute the queries, keys and values for all
        # heads
        self.tokeys    = nn.Linear(k, k, bias=False)
        self.toqueries = nn.Linear(k, k, bias=False)
        self.tovalues  = nn.Linear(k, k, bias=False)

            # This will be applied after the multi-head self-atte
        self.unifyheads = nn.Linear(k, k)
```

We can now implement the computation of the self-attention (the module's `forward` function). First, we compute the queries, keys and values for all heads:

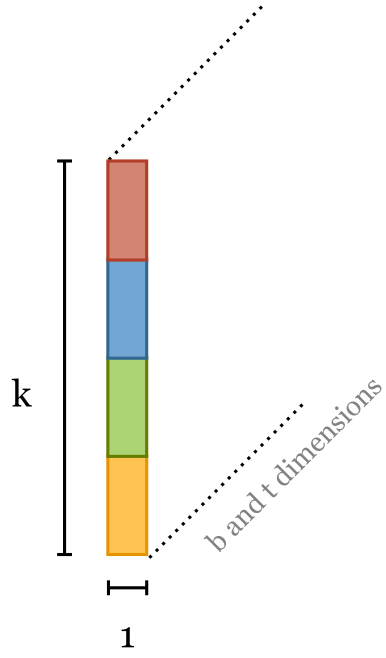
```
    def forward(self, x):

        b, t, k = x.size()
        h = self.heads

        queries = self.toqueries(x)
```

```
        keys    = self.tokeys(x)
        values  = self.tovalues(x)
```

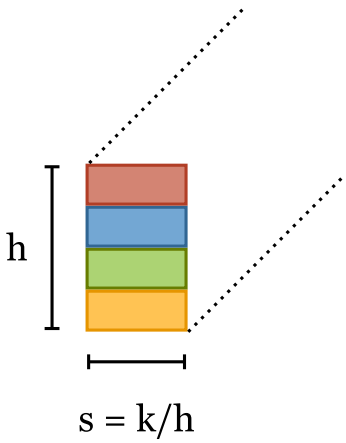
This gives us three vector sequences of the full embedding dimension `k`. As we saw above we can now cut these into `h` chunks. we can do this with a simple view operation:

```
        s = k // h

        keys    = keys.view(b, t, h, s)
        queries = queries.view(b, t, h, s)
        values  = values.view(b, t, h, s)
```

This simply reshapes the tensors to add a dimension that iterations over the heads. For a single vector in our sequence you can think of it as reshaping a vector of dimension `k` into a matrix of `h` by `k//h`:



shape: b, t, k



shape: b, t, h, s

Next, we need to compute the dot products. This is the same operation for every head, so we fold the heads into the batch dimension. This ensures that we can use `torch.bmm()` as before, and the whole collection of keys, queries and values will just be seen as a slightly larger batch.

Since the head and batch dimension are not next to each other, we need to transpose before we reshape. (This is costly, but it seems to be unavoidable.)

```
# - fold heads into the batch dimension
keys = keys.transpose(1, 2).contiguous().view(b * h, t, s)
queries = queries.transpose(1, 2).contiguous().view(b * h,
values = values.transpose(1, 2).contiguous().view(b * h, t
```

You can avoid these calls to `contiguous()` by using `reshape()` instead of `view()` but I prefer to make it explicit when we are copying a tensor, and when we are just viewing it. See this notebook for an explanation of the difference.

As before, the dot products can be computed in a single matrix multiplication, but now between the queries and the keys.

```
# Get dot product of queries and keys, and scale
dot = torch.bmm(queries, keys.transpose(1, 2))
# -- dot has size (b*h, t, t) containing raw weights

# scale the dot product
dot = dot / (k ** (1/2))

# normalize
dot = F.softmax(dot, dim=2)
# - dot now contains row-wise normalized weights
```

We apply the self attention weights `dot` to the values, giving us the output for each attention head

```
        # apply the self attention to the values
        out = torch.bmm(dot, values).view(b, h, t, s)
```

To unify the attention heads, we transpose again, so that the head dimension and the embedding dimension are next to each other, and reshape to get concatenated vectors of dimension `e`. We then pass these through the `unifyheads` layer for a final projection.

```
        # swap h, t back, unify heads
        out = out.transpose(1, 2).contiguous().view(b, t, s * h)

        return self.unifyheads(out)
```

And there you have it: multi-head, scaled dot-product self attention. You can see the complete implementation here.

> *The implementation can be made more concise using einsum notation (see an example here).*

Building *transformers*

A transformer is not just a self-attention layer, it is an *architecture*. It's not quite clear what does and doesn't

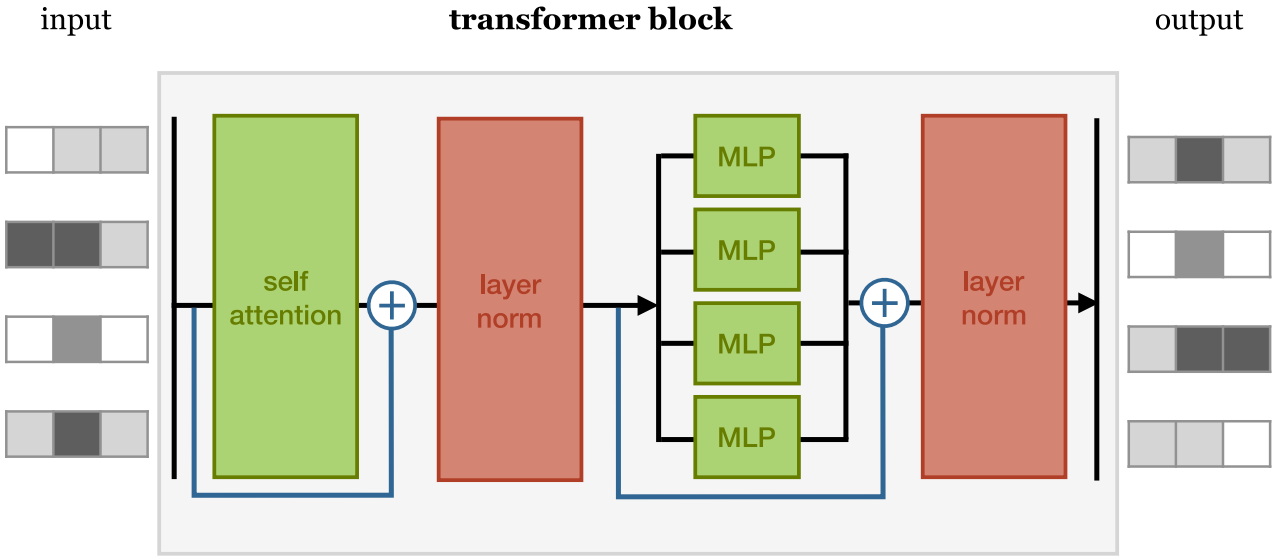
qualify as a transformer, but here we'll use the following definition:

> Any architecture designed to process a connected set of units—such as the tokens in a sequence or the pixels in an image—where the only interaction between units is through self-attention.

As with other mechanisms, like convolutions, a more or less standard approach has emerged for how to build self-attention layers up into a larger network. The first step is to wrap the self-attention into a *block* that we can repeat.

The transformer block

There are some variations on how to build a basic transformer block, but most of them are structured roughly like this:



That is, the block applies, in sequence: a self attention layer, layer normalization, a feed forward layer (a single MLP applied independently to each vector), and another layer normalization. Residual connections are added around both, before the normalization. The order of the various components is not set in stone; the important thing is to combine self-attention with a local feedforward, and to add normalization and residual connections.

> *Normalization and residual connections are standard tricks used to help deep neural networks train faster and more accurately. The layer normalization is applied over the embedding dimension only.*

Here's what the transformer block looks like in pytorch.

```
class TransformerBlock(nn.Module):
  def __init__(self, k, heads):
```

```
        super().__init__()

        self.attention = SelfAttention(k, heads=heads)

        self.norm1 = nn.LayerNorm(k)
        self.norm2 = nn.LayerNorm(k)

        self.ff = nn.Sequential(
          nn.Linear(k, 4 * k),
          nn.ReLU(),
          nn.Linear(4 * k, k))

    def forward(self, x):
        attended = self.attention(x)
        x = self.norm1(attended + x)

        fedforward = self.ff(x)
        return self.norm2(fedforward + x)
```

We've made the relatively arbitrary choice of making the hidden layer of the feedforward 4 times as big as the input and output. Smaller values may work as well, and save memory, but it should be bigger than the input/output layers.

Classification transformer

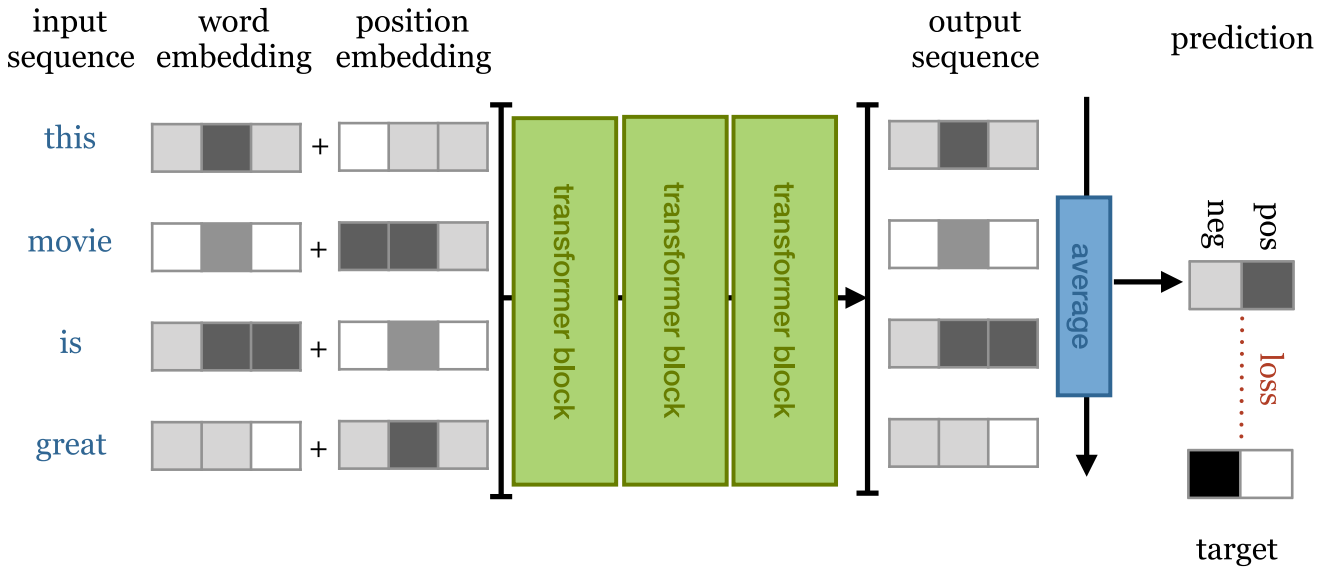
The simplest transformer we can build is a *sequence classifier*. We'll use the IMDb sentiment classification dataset: the instances are movie reviews, tokenized into sequences of words, and the classification labels are `positive` and `negative` (indicating whether the review was positive or negative about the movie).

The heart of the architecture will simply be a large chain of transformer blocks. All we need to do is work out how to feed it the input sequences, and how to transform the final output sequence into a a single classification.

The whole experiment can be found here. We won't deal with the data wrangling in this blog post. Follow the links in the code to see how the data is loaded and prepared.

Output: producing a classification

The most common way to build a sequence classifier out of sequence-to-sequence layers, is to apply global average pooling to the final output sequence, and to map the result to a softmaxed class vector.



Overview of a simple sequence classification transformer. The output sequence is averaged to produce a single vector representing the whole sequence. This vector is projected down to a vector with one element per class and softmaxed to produce probabilities.

Input: using the positions

We've already discussed the principle of an embedding layer. This is what we'll use to represent the words.

However, as we've also mentioned already, we're stacking permutation equivariant layers, and the final global average pooling is permutation *in*variant, so the network as a whole is also permutation invariant. Put more simply: if we shuffle up the words in the sentence, we get the exact same classification, whatever weights we learn. Clearly, we want

our state-of-the-art language model to have at least some sensitivity to word order, so this needs to be fixed.

The solution is simple: we create a second vector of equal length, that represents the position of the word in the current sentence, and add this to the word embedding. There are two options.

position embeddings We simply *embed* the positions like we did the words. Just like we created embedding vectors $\boldsymbol{v}_{\text{cat}}$ and $\boldsymbol{v}_{\text{susan}}$, we create embedding vectors $\boldsymbol{v}_{12}$ and $\boldsymbol{v}_{25}$. Up to however long we expect sequences to get. The drawback is that we have to see sequences of every length during training, otherwise the relevant position embeddings don't get trained. The benefit is that it works pretty well, and it's easy to implement.

position encodings Position encodings work in the same way as embeddings, except that we don't *learn* the position vectors, we just choose some function $f : \mathbb{N} \to \mathbb{R}^k$ to map the positions to real valued vectors, and let the network figure out how to interpret these encodings. The benefit is that for a well chosen function, the network should be able to deal with sequences that are longer than those it's seen during training (it's unlikely to perform well on them, but at least we can check). The drawbacks are that the choice of encoding

function is a complicated hyperparameter, and it complicates the implementation a little.

For the sake of simplicity, we'll use position embeddings in our implementation.

Pytorch

Here is the complete text classification transformer in pytorch.

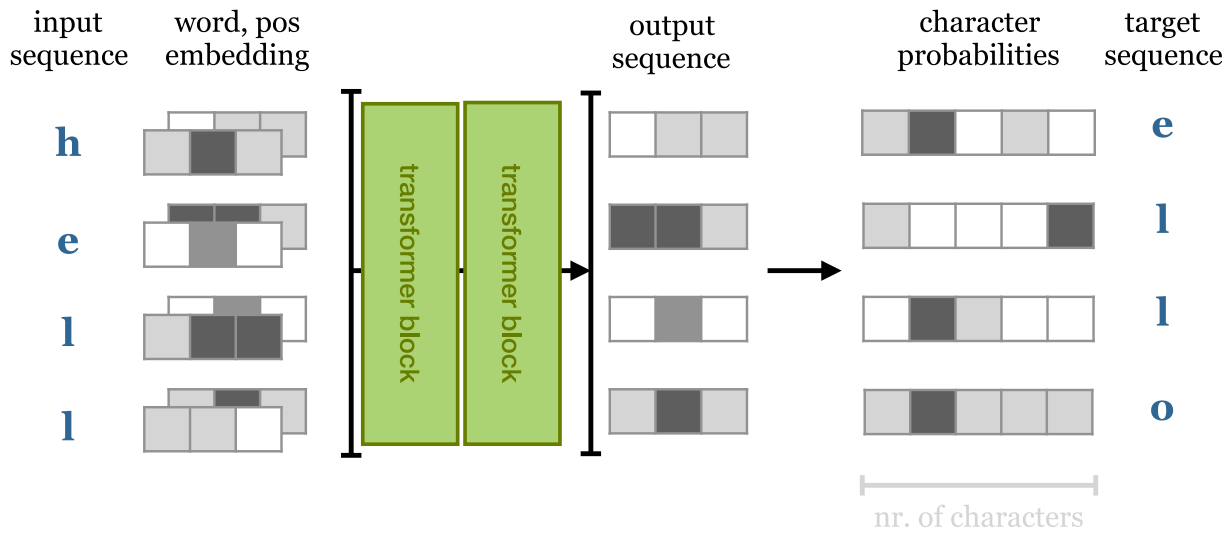
```
s Transformer(nn.Module):
def __init__(self, k, heads, depth, seq_length, num_tokens, nu
    super().__init__()

    self.num_tokens = num_tokens
    self.token_emb = nn.Embedding(num_tokens, k)
    self.pos_emb = nn.Embedding(seq_length, k)

            # The sequence of transformer blocks that does all
            # heavy lifting
    tblocks = []
    for i in range(depth):
        tblocks.append(TransformerBlock(k=k, heads=heads))
    self.tblocks = nn.Sequential(*tblocks)
```

```
            # Maps the final output sequence to class logits
        self.toprobs = nn.Linear(k, num_classes)

    def forward(self, x):
        """
        :param x: A (b, t) tensor of integer values representing
                  words (in some predetermined vocabulary).
        :return: A (b, c) tensor of log-probabilities over the
                 classes (where c is the nr. of classes).
        """
            # generate token embeddings
        tokens = self.token_emb(x)
        b, t, k = tokens.size()

            # generate position embeddings
            positions = torch.arange(t)
        positions = self.pos_emb(positions)[None, :, :].expand(b,

        x = tokens + positions
        x = self.tblocks(x)

        # Average-pool over the t dimension and project to class
        # probabilities
        x = self.toprobs(x.mean(dim=1))
        return F.log_softmax(x, dim=1)
```

At depth 6, with a maximum sequence length of 512, this transformer achieves an accuracy of about 85%, competitive with results from RNN models, and much faster to train. To see the real near-human performance of transformers, we'd need to train a much deeper model on much more data. More about how to do that later.

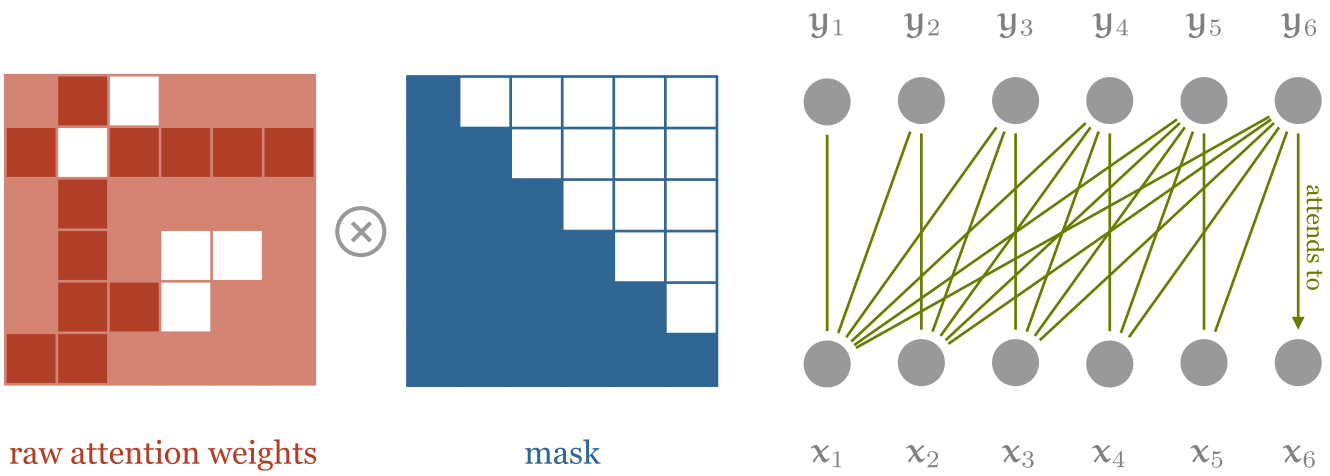
Text generation transformer

The next trick we'll try is an *autoregressive* model. We'll train a *character* level transformer to predict the next character in a sequence. The training regime is simple (and has been around for far longer than transformers have). We give the sequence-to-sequence model a sequence, and we ask it to predict the next character at each point in the sequence. In other words, the target output is the same sequence shifted one character to the left:



With RNNs this is all we need to do, since they cannot look forward into the input sequence: output i depends only on inputs 0 to i. With a transformer, the output depends on the entire input sequence, so prediction of the next character becomes vacuously easy, just retrieve it from the input.

To use self-attention as an autoregressive model, we'll need to ensure that it cannot look forward into the sequence. We do this by applying a mask to the matrix of dot products, before the softmax is applied. This mask disables all elements above the diagonal of the matrix.



> Masking the self attention, to ensure that elements can only attend to input elements that precede them in the sequence. Note that the multiplication symbol is slightly misleading: we actually set the masked out elements (the white squares) to $-\infty$

Since we want these elements to be zero after the softmax, we set them to $-\infty$. Here's how that looks in pytorch:

```
dot = torch.bmm(queries, keys.transpose(1, 2))

indices = torch.triu_indices(t, t, offset=1)
dot[:, indices[0], indices[1]] = float('-inf')

dot = F.softmax(dot, dim=2)
```

After we've handicapped the self-attention module like this, the model can no longer look forward in the sequence.

We train on the standard `enwik8` dataset (taken from the Hutter prize), which contains 10^8 characters of Wikipedia text (including markup). During training, we generate batches by randomly sampling subsequences from the data.

We train on sequences of length 256, using a model of 12 transformer blocks and 256 embedding dimension. After about 24 hours training on an RTX 2080Ti (some 170K batches of size 32), we let the model generate from a 256-character seed: for each character, we feed it the preceding 256 characters, and look what it predicts for the next character (the last output vector). We sample from that with a temperature of 0.5, and move to the next character.

The output looks like this:

```
1228X Human & Rousseau. Because many of his stories were
originally published in long-forgotten magazines and
journals, there are a number of [[anthology|anthologies]] by
different collators each containing a different selection.
His original books have been considered an anthologie in the
[[Middle Ages]], and were likely to be one of the most common
in the [[Indian Ocean]] in the [[1st century]]. As a result
of his death, the Bible was recognised as a counter-attack by
the [[Gospel of Matthew]] (1177-1133), and the
[[Saxony|Saxons]] of the [[Isle of Matthew]] (1100-1138), the
third was a topic of the [[Saxony|Saxon]] throne, and the
[[Roman Empire|Roman]] troops of [[Antiochia]] (1145-1148).
The [[Roman Empire|Romans]] resigned in [[1148]] and [[1148]]
began to collapse. The [[Saxony|Saxons]] of the [[Battle of
Valasander]] reported the y
```

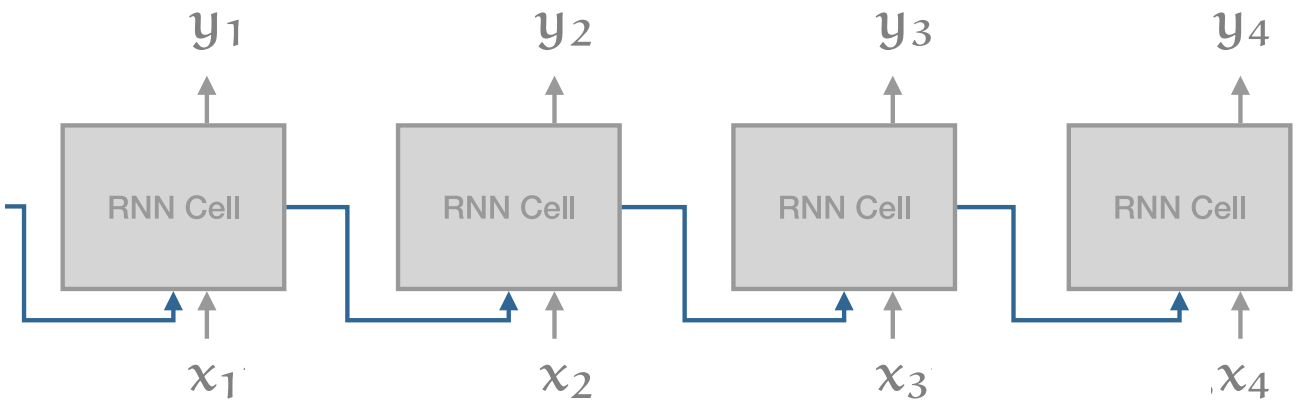
Note that the Wikipedia link tag syntax is correctly used, that the text inside the links represents reasonable subjects for links. Most importantly, note that there is a rough thematic consistency; the generated text keeps on the subject of the bible, and the Roman empire, using different related terms at different points. While this is far from the performance of a model like GPT-2, the benefits over a similar RNN model are clear already: faster training (a similar RNN model would take many days to train) and better long-term coherence.

In case you're curious, the Battle of Valasander seems to be an invention of the network.

At this point, the model achieves a compression of 1.343 bits per byte on the validation set, which is not too far off the state of the art of 0.93 bits per byte, achieved by the GPT-2 model (described below).

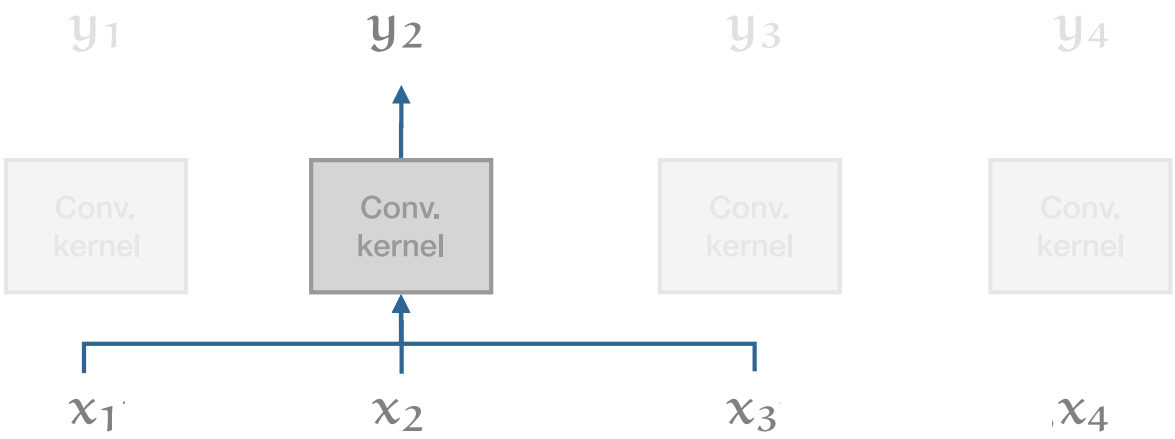
Design considerations

To understand why transformers are set up this way, it helps to understand the basic design considerations that went into them. The main point of the transformer was to overcome the problems of the previous state-of-the-art architecture, the RNN (usually an LSTM or a GRU). Unrolled, an RNN looks like this:



The big weakness here is the recurrent connection. while this allows information to propagate along the sequence, it also means that we cannot compute the cell at time step i until

we've computed the cell at timestep $i - 1$. Contrast this with a 1D convolution:



In this model, every output vector can be computed in parallel with every other output vector. This makes convolutions much faster. The drawback with convolutions, however, is that they're severely limited in modeling *long range dependencies*. In one convolution layer, only words that are closer together than the kernel size can interact with each other. For longer dependence we need to stack many convolutions.

The transformer is an attempt to capture the best of both worlds. They can model dependencies over the whole range of the input sequence just as easily as they can for words that are next to each other (in fact, without the position vectors, they can't even tell the difference). And yet, there are no recurrent connections, so the whole model can be computed in a very efficient feedforward fashion.

The rest of the design of the transformer is based primarily on one consideration: depth. Most choices follow from the desire to train big stacks of transformer blocks. Note for instance that there are only two places in the transformer where non-linearities occur: the softmax in the self-attention and the ReLU in the feedforward layer. The rest of the model is entirely composed of linear transformations, which perfectly preserve the gradient.

> *I suppose the layer normalization is also nonlinear, but that is one nonlinearity that actually helps to keep the gradient stable as it propagates back down the network.*

Historical baggage

If you've read other introductions to transformers, you may have noticed that they contain some bits I've skipped. I think these are not necessary to understand modern transformers. They are, however, helpful to understand some of the terminology and some of the writing *about* modern transformers. Here are the most important ones.

Why is it called self-*attention*?

Before self-attention was first presented, sequence models consisted mostly of recurrent networks or convolutions

stacked together. At some point, it was discovered that these models could be helped by adding *attention mechanisms*: instead of feeding the output sequence of the previous layer directly to the input of the next, an intermediate mechanism was introduced, that decided which elements of the input were relevant for a particular word of the output.

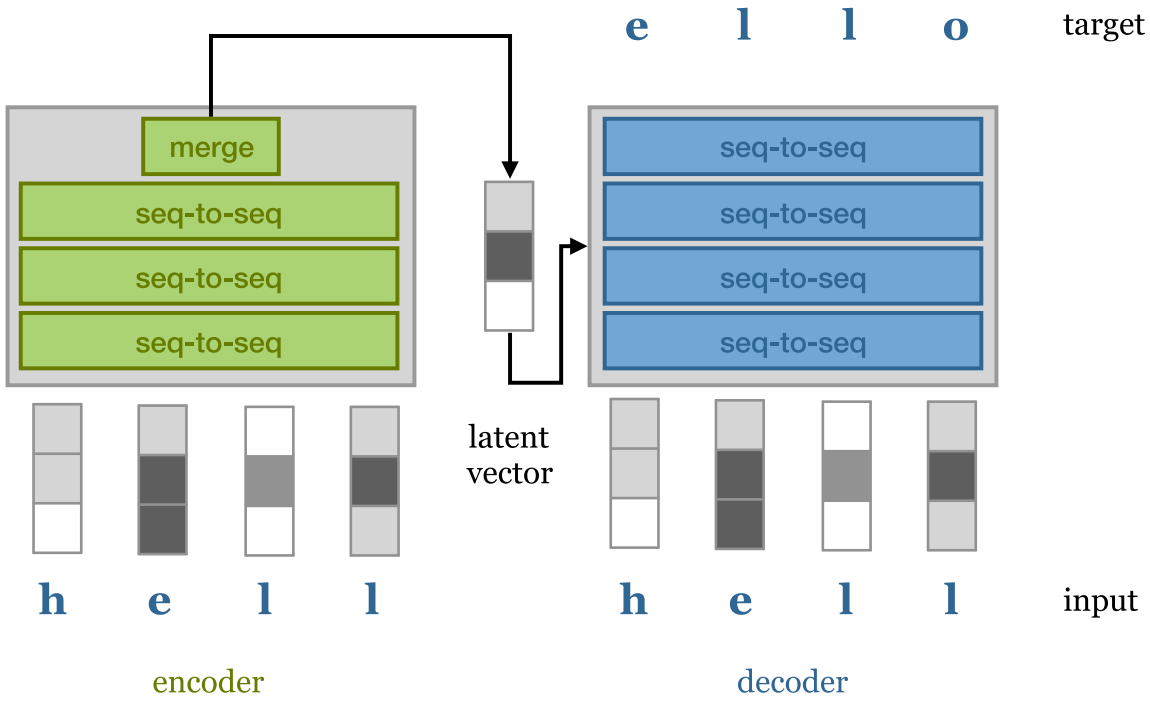
The general mechanism was as follows. We call the input the **values**. Some (trainable) mechanism assigns a **key** to each value. Then to each output, some other mechanism assigns a **query**.

These names derive from the datastructure of a key-value store. In that case we expect only one item in our store to have a key that matches the query, which is returned when the query is executed. Attention is a softened version of this: *every* key in the store matches the query to some extent. All are returned, and we take a sum, weighted by the extent to which each key matches the query.

The great breakthrough of self-attention was that attention by itself is a strong enough mechanism to do all the learning. Attention is all you need, as the authors put it. The key, query and value are all the same vectors (with minor linear transformations). They *attend to themselves* and stacking such self-attention provides sufficient nonlinearity and representational power to learn very complicated functions.

The original transformer: encoders and decoders

But the authors did not dispense with all the complexity of contemporary sequence modeling. The standard structure of sequence-to-sequence models in those days was an encoder-decoder architecture, with teacher forcing.



The *encoder* takes the input sequence and maps it to a *latent* representation of the whole sequence. This can be either a sequence of latent vectors, or a single one as in the image above. This vector is then passed to a *decoder* which unpacks it to the desired target sequence (for instance, the same sentence in another language).

Teacher forcing refers to the technique of also allowing the decoder access to the input sentence, but in an

autoregressive fashion. That is, the decoder generates the output sentence word for word based both on the latent vector and the words it has already generated. This takes some of the pressure off the latent representation: the decoder can use word-for-word sampling to take care of the low-level structure like syntax and grammar and use the latent vector to capture more high-level semantic structure. Decoding twice with the same latent representation would, ideally, give you two different sentences with the same meaning.

In later transformers, like BERT and GPT-2, the encoder/decoder configuration was entirely dispensed with. A simple stack of transformer blocks was found to be sufficient to achieve state of the art in many sequence based tasks.

> *This approach is sometimes called a decoder-only transformer (for an autoregressive model) or an encoder-only transformer (for a model without masking).*

Modern transformers

Here's a small selection of some modern transformers and their most characteristic details.

BERT

BERT was one of the first models to show that transformers could reach human-level performance on a variety of language based tasks: question answering, sentiment classification or classifying whether two sentences naturally follow one another.

BERT consists of a simple stack of transformer blocks, of the type we've described above. This stack is *pre-trained* on a large general-domain corpus consisting of 800M words from English books (modern work, from unpublished authors), and 2.5B words of text from English Wikipedia articles (without markup).

Pretraining is done through two tasks:

Masking A certain number of words in the input sequence are: masked out, replaced with a random word or kept as is. The model is then asked to predict, for these words, what the original words were. Note that the model doesn't need to predict the entire denoised sentence, just the modified words. Since the model doesn't know which words it will be asked about, it learns a representation for every word in the sequence.

Next sequence classification Two sequences of about 256 words are sampled that either (a) follow each other

directly in the corpus, or (b) are both taken from random places. The model must then predict whether a or b is the case.

BERT uses WordPiece tokenization, which is somewhere in between word-level and character level sequences. It breaks words like walking up into the tokens walk and ##ing. This allows the model to make some inferences based on word structure: two verbs ending in -ing have similar grammatical functions, and two verbs starting with walk- have similar semantic function.

The input is prepended with a special <cls> token. The output vector corresponding to this token is used as a sentence representation in sequence classification tasks like the next sentence classification (as opposed to the global average pooling over all vectors that we used in our classification model above).

After pretraining, a single task-specific layer is placed after the body of transformer blocks, which maps the general purpose representation to a task specific output. For classification tasks, this simply maps the first output token to softmax probabilities over the classes. For more complex tasks, a final sequence-to-sequence layer is designed specifically for the task.

The whole model is then re-trained to finetune the model for the specific task at hand.

In an ablation experiment, the authors show that the largest improvement as compared to previous models comes from the bidirectional nature of BERT. That is, previous models like GPT used an autoregressive mask, which allowed attention only over previous tokens. The fact that in BERT all attention is over the whole sequence is the main cause of the improved performance.

> *This is why the B in BERT stands for "bidirectional".*

The largest BERT model uses 24 transformer blocks, an embedding dimension of 1024 and 16 attention heads, resulting in 340M parameters.

GPT-2

GPT-2 is the first transformer model that actually made it into the mainstream news, after the controversial decision by OpenAI not to release the full model.

> *The reason was that GPT-2 could generate sufficiently believable text that large-scale fake news campaigns of the kind seen in the 2016 US presidential election would become effectively a one-person job.*

The first trick that the authors of GPT-2 employed was to create a new high-quality dataset. While BERT used high-quality data, their sources (lovingly crafted books and well-edited wikipedia articles) had a certain lack of diversity in the writing style. To collect more diverse data without sacrificing quality the authors used links posted on the social media site *Reddit* to gather a large collection of writing with a certain minimum level of social support (expressed on Reddit as *karma*).

GPT2 is fundamentally a language *generation* model, so it uses masked self-attention like we did in our model above. It uses byte-pair encoding to tokenize the language, which , like the WordPiece encoding breaks words up into tokens that are slightly larger than single characters but less than entire words.

GPT2 is built very much like our text generation model above, with only small differences in layer order and added tricks to train at greater depths. The largest model uses 48 transformer blocks, a sequence length of 1024 and an embedding dimension of 1600, resulting in 1.5B parameters.

They show state-of-the art performance on many tasks. On the wikipedia compression task that we tried above, they achieve 0.93 bits per byte.

Transformer-XL

While the transformer represents a massive leap forward in modeling long-range dependency, the models we have seen so far are still fundamentally limited by the size of the input. Since the size of the dot-product matrix grows quadratically in the sequence length, this quickly becomes the bottleneck as we try to extend the length of the input sequence. Transformer-XL is one of the first succesful transformer models to tackle this problem.

During training, a long sequence of text (longer than the model could deal with) is broken up into shorter segments. Each segment is processed in sequence, with self-attention computed over the tokens in the curent segment *and the previous segment*. Gradients are only computed over the current segment, but information still propagates as the segment window moves through the text. In theory at layer n, information may be used from n segments ago.

> *A similar trick in RNN training is called truncated backpropagation through time. We feed the model a very long sequence, but backpropagate only over part of it. The first part of the sequence, for which no gradients are computed, still influences the values of the hidden states in the part for which they are.*

To make this work, the authors had to let go of the standard position encoding/embedding scheme. Since the position encoding is *absolute*, it would change for each segment and not lead to a consistent embedding over the whole sequence. Instead they use a *relative* encoding. For each output vector, a different sequence of position vectors is used that denotes not the absolute position, but the distance to the current output.

This requires moving the position encoding into the attention mechanism (which is detailed in the paper). One benefit is that the resulting transformer will likely generalize much better to sequences of unseen length.

Sparse transformers

Sparse transformers tackle the problem of quadratic memory use head-on. Instead of computing a dense matrix of attention weights (which grows quadratically), they compute the self-attention only for particular pairs of input tokens, resulting in a *sparse* attention matrix, with only $n\sqrt{n}$ explicit elements.

This allows models with very large context sizes, for instance for generative modeling over images, with large dependencies between pixels. The tradeoff is that the sparsity structure is not learned, so by the choice of sparse

matrix, we are disabling some interactions between input tokens that might otherwise have been useful. However, two units that are not directly related may still interact in higher layers of the transformer (similar to the way a convolutional net builds up a larger receptive field with more convolutional layers).

Beyond the simple benefit of training transformers with very large sequence lengths, the sparse transformer also allows a very elegant way of designing an inductive bias. We take our input as a collection of units (words, characters, pixels in an image, nodes in a graph) and we specify, through the sparsity of the attention matrix, which units we believe to be related. The rest is just a matter of building the transformer up as deep as it will go and seeing if it trains.

Going big

The big bottleneck in training transformers is the matrix of dot products in the self attention. For a sequence length t, this is a dense matrix containing t^2 elements. At standard 32-bit precision, and with $t = 1000$ a batch of 16 such matrices takes up about 64Mb of memory. Since we need to store at least four of them for each head of each self-attention operation (before and after softmax, plus their gradients), that limits us to at most twelve 4-head layers in a standard 12Gb GPU. In practice, we get even less, since the

inputs and outputs also take up a lot of memory (although the dot product dominates).

And yet models reported in the literature contain much longer sequence lengths, with 48 or more layers, using dense dot product matrices. These models are trained on clusters, of course, but a single GPU is still required to do a single forward/backward pass. How do we fit such humongous transformers into 12Gb of memory? There are three main tricks:

Half precision On modern GPUs and on TPUs, tensor computations can be done efficiently on 16-bit float tensors. This isn't quite as simple as just setting the dtype of the tensor to `torch.float16`. For some parts of the network, like the loss, 32 bit precision is required. But most of this can be handled with relative ease by existing libraries. Practically, this doubles your effective memory.

Gradient accumulation For a large model, we may only be able to perform a forward/backward pass on a single instance. Batch size 1 is not likely to lead to stable learning. Luckily, we can perform a single forward/backward for each instance in a larger batch, and simply sum the gradients we find (this is a consequence of the multivariate chain rule). When we hit the end of the batch, we do a single step of gradient descent, and zero out the gradient. In Pytorch this is

particulary easy: you know that `optimizer.zero_grad()` call in your training loop that seems so superfluous? If you don't make that call, the new gradients are simply added to the old ones.

Gradient checkpointing If your model is so big that even a single forward/backward won't fit in memory, you can trade off even more computation for memory efficiency. In gradient checkpointing, you separate your model into sections. For each section, you do a separate forward/backward to compute the gradients, without retaining the intermediate values for the rest. Pytorch has special utilities for gradient checkpointng.

For more information on how to do this, see this blogpost.

Conclusion

The transformer may well be the simplest machine learning architecture to dominate the field in decades. There are good reasons to start paying attention to them if you haven’t been already.

Firstly, **the current performance limit is purely in the hardware**. Unlike convolutions or LSTMs the current limitations to what they can do are entirely determined by how big a model we can fit in GPU memory and how much data we can push through it in a reasonable amount of time.

I have no doubt, we will eventually hit the point where more layers and and more data won't help anymore, but we don't seem to have reached that point yet.

Second, **transformers are extremely generic**. So far, the big successes have been in language modelling, with some more modest achievements in image and music analysis, but the transformer has a level of generality that is waiting to be exploited. The basic transformer is a *set-to-set* model. So long as your data is a set of units, you can apply a transformer. Anything else you know about your data (like local structure) you can add by means of position embeddings, or by manipulating the structure of the attention matrix (making it sparse, or masking out parts).

This is particularly useful in multi-modal learning. We could easily combine a captioned image into a set of pixels and characters and design some clever embeddings and sparsity structure to help the model figure out how to combine and align the two. If we combine the entirety of our knowledge about our domain into a relational structure like a multi-modal knowledge graph (as discussed in [3]), simple transformer blocks could be employed to propagate information between multimodal units, and to align them with the sparsity structure providing control over which units directly interact.

So far, transformers are still primarily seen as a language model. I expect that in time, we'll see them adopted much more in other domains, not just to increase performance, but to simplify existing models, and to allow practitioners more intuitive control over their models' inductive biases.

References

[1] The illustrated transformer, Jay Allamar.

[2] The annotated transformer, Alexander Rush.

[3] The knowledge graph as the default data model for learning on heterogeneous knowledge Xander Wilcke, Peter Bloem, Victor de Boer

[4] Matrix factorization techniques for recommender systems Yehuda Koren et al.

Updates

19 October 2019 Added a section on the difference between wide and narrow self-attention. Thanks to Sidney Melo for spotting the mistake in the original implementation.
9 December 2022 Clarified the example in the section on multi-head attention.
4 March 2023

Fixed a persistent mistake in the defintition of multi-head self-attention. Rewrote the final self-attention to be the canonical form (rather than the earlier "wide" variety we used for the sake of simplicity).

1 Aug 2023 Fixed some small mistakes in the section "Going big".

TRANSFORMERS FROM SCRATCH