mlexplained.com

Paper Dissected: “Attention is All You Need” Explained

24-30 minutes

"Attention is All You Need", is an influential paper with a catchy title that fundamentally changed the field of machine translation. Previously, RNNs were regarded as the go-to architecture for translation. This paper surprised everyone by introducing the Transformer, a network with no recurrence that only used attention (as well as a couple of other components) only.

Whether attention really is all you need, this paper is a huge milestone in neural NLP, and this post is an attempt to dissect and explain it.

Update: I've heavily updated this post to include code and better explanations regarding the intuition behind how the Transformer works. I've also implemented the Transformer from scratch in a Jupyter notebook which you can view here.

TL;DR

- RNN based architectures are **hard to parallelize** and can have difficulty learning **long-range dependencies** within the input and output sequences
- The Transformer models all these dependencies using **attention**

- Instead of using one sweep of attention, the Transformer uses multiple "**heads**" (multiple attention distributions and multiple outputs for a single input).
- In addition to attention, the Transformer uses layer normalization and residual connections to make optimization easier.
- Attention cannot utilize the positions of the inputs. To solve this, the Transformer uses **explicit position encoding**s are added to the input embeddings
- This architecture achieves state-of-the-art performance on English-to-German and English-to-French translation and performs well on constituency parsing

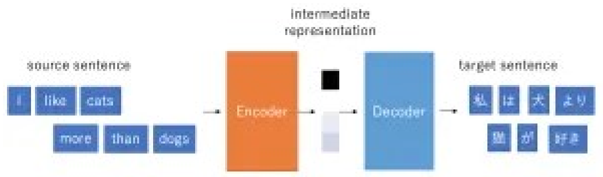
  For more details, please read on!

1. A Bit of Background on Neural Machine Translation

For those unfamiliar with neural machine translation, I'll provide a quick overview in this section that should be enough to understand the paper "Attention is All You Need". If you're already familiar with this content, please read ahead to the next section!

Machine translation is - at its core - simply a task where you map a sentence to another sentence. Sentences are comprised of words, so this is equivalent to mapping a **sequence** to another **sequence**. Mapping sequences to sequences is a ubiquitous task structure in NLP (other tasks with this structure include language modeling and part-of-speech tagging), so people have developed many methods for performing such a mapping: these methods are referred to as **sequence-to-sequence** methods.

Generally, sequence-to-sequence tasks are performed using an **encoder-decoder** model. The basic idea is that the encoder takes the sequence of input words (e.g. "I like cats more than dogs"), converts it to some intermediate representation, then passes that representation to the decoder which produces the output sequence (e.g. 私は犬より猫が好き). These models are trained to maximize the likelihood of generating the correct output sequence: at each step, the decoder is rewarded for predicting the next word correctly and penalized for making mistakes.



Before the Transformer, RNNs were the most widely-used and successful architecture for both the encoder and decoder. RNNs seemed to be born for this task: their recurrent nature perfectly matched the sequential nature of language. Humans read sentences from left to right (or right to left depending on where you live), so it made sense to use RNNs to encode and decode language.

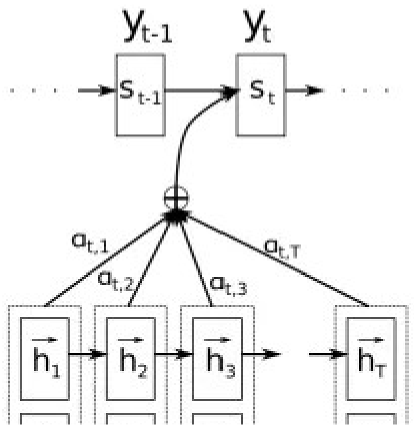
Now, this is all great when the sentences are short, but when they become longer we encounter a problem. If you look at the image above, you'll see that the intermediate representation is a bottleneck in terms of information: all the semantics of the source sentence, including the words cats, docs, and like, must all be included in that single vector. This becomes really hard, really quickly: as famously said at the ACL 2014 workshop:

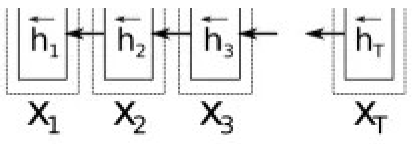
> "You can’t cram the meaning of a whole %&!$# sentence into a single $&!#* vector!"

The problem with the encoder-decoder approach above is that the decoder needs different information at different timesteps.For instance, in the example of translating the sentence "I like cats more than dogs" to "私は犬より猫が好き", the second token in the input ("like") corresponds to the last token in the output ("好き"), creating a long-term dependency that the RNN has to carry all while reading the source sentence and generating the target sentence. In contrast, after it outputs the word "dog" (which is "犬" in Japanese), it needs to know what the dog was being compared against, but no longer needs to remember about the dog. This problem is the original motivation behind the attention mechanism.

Intuitively, the attention mechanism allows the decoder to "look back" at the entire sentence and selectively extract the information it needs during decoding.

Concretely, attention gives the decoder access to **all** the encoder's hidden states. The decoder still needs to make a single prediction for the next word though, so we can't just pass it a whole sequence: we need to pass it some kind of summary vector. So what attention does is it asks the decoder to **choose** which hidden states to use and which to ignore by weighting the hidden states. The decoder is then passed a weighted sum of hidden states to use to predict the next word.





The original attention mechanism. The decoder state is used to compute the attention weights of the hidden encoder states. The attention weights change for each decoder state, and the model learns to "focus" on the relevant parts of the input.

The attention weight can be computed in many ways, but the original attention mechanism used a simple feed-forward neural network.

This was all very high-level and hand-wavy, but I hope you got the gist of attention. Attention basically gives the decoder access to all of the original information instead of just a summary and allows the decoder to pick and choose what information to use.

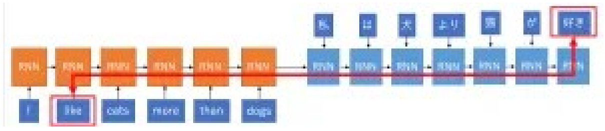
2. What's Wrong with RNNs?

Given what we just learned above, it would seem like attention solves all the problems with RNNs and encoder-decoder architectures. However, there are a few shortcomings of RNNs that the Transformer tries to address.

One is the **sequential nature** of RNNs. When we process a sequence using RNNs, each hidden state depends on the previous hidden state. This becomes a major pain point on GPUs: GPUs have a lot of computational capability and they hate having to wait for data to become available. Even with technologies like CuDNN, RNNs are painfully inefficient and slow on the GPU.

The other is the difficulty of learning **long-range dependencies** in the network. Now, you may be wondering, didn't LSTMs handle the long-range dependency problem in RNNs? And didn't we introduce attention to handle this problem a few paragraphs ago?

Well, theoretically, LSTMs (and RNNs in general) can have long-term memory. But remembering things for long periods is still a challenge, and RNNs can still have short-term memory problems. Furthermore, some words have multiple meanings that only become apparent in context. For instance, the word "than" as in "She is taller than me" and "I have no choice other than to write this blog post" use "than" in different ways. The output tokens are also dependent on each other. If the target sentence was "Neither he nor I knew about deep learning", the token "nor" is dependent on the negative clause "neither".



The dependency that the RNN has to handle. The path length is proportional to the length of the input + output.

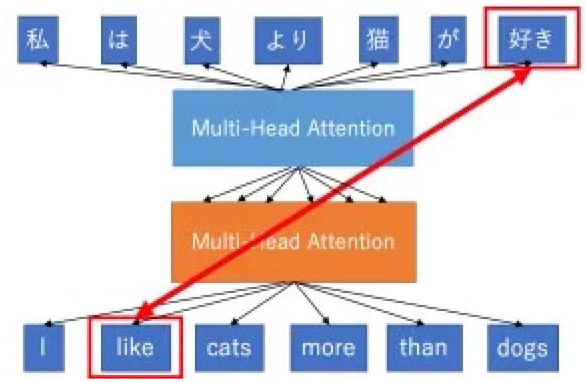
In essence, there are three kinds of dependencies in neural machine translations: dependencies between

(1) the input and output tokens

(2) the input tokens themselves

(3) the output tokens themselves.

The traditional attention mechanism largely solved the first dependency by giving the decoder access to the entire input sequence. The novel idea of the Transformer is to extend this mechanism to the processing input and output sentences as well. Instead of going from left to right using RNNs, why don't we just allow the encoder and decoder to **see the entire input sequence all at once**, directly modeling these dependencies using attention?

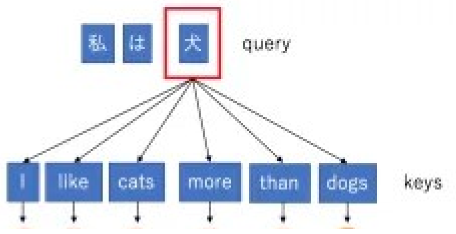


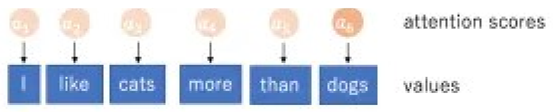
The dependency that the Transformer has to learn. Now the path length is independent of the length of the source and target sentences.

This is the basic idea behind the Transformer. Now, we turn to the details of the implementation. The key component of the Transformer is the **Multi-Head Attention** block. Let's look at this component in detail.

3. The Key Component: Multi-Head Attention

The attention mechanism in the Transformer is interpreted as a way of computing the relevance of a set of **values**(information)based on some **keys** and **queries**. Basically, the attention mechanism is used as a way for the model to **focus on relevant information** based on what it is currently processing. Traditionally, the attention weights were the *relevance* of the encoder hidden states (*values*) in processing the decoder state (*query*) and were calculated based on the encoder hidden states (*keys*) and the decoder hidden state (*query*).



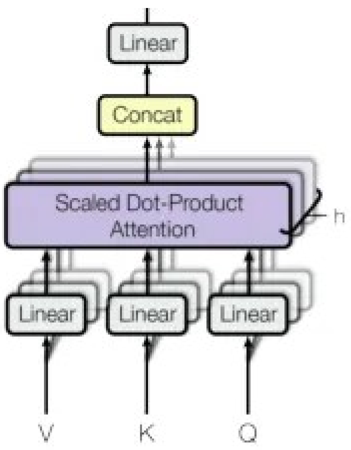


In this example, the query is the word being decoded ("犬" which means dog) and both the keys and values are the source sentence. The attention score represents the relevance, and in this case is large for the word "dog" and small for others.

When we think of attention this way, we can see that the keys, values, and queries could be anything. They could even be the same! For instance, both values and queries could be input embeddings. Though I'll discuss the details later, the encoder uses source sentence's embeddings for its keys, values, and queries, whereas the decoder uses the encoder's outputs for its keys and values and the target sentence's embeddings for its queries (technically this is slightly inaccurate, but again, I'll get to this later).

If we only computed a single attention weighted sum of the values, it would be difficult to capture various different aspects of the input. For instance, in the sentence "I like cats more than dogs", you might want to capture the fact that the sentence compares two entities, while also retaining the actual entities being compared. To solve this problem the Transformer uses the **Multi-Head Attention** block. This block computes *multiple* attention weighted sums instead of a *single* attention pass over the values - hence the name "Multi-Head" Attention.

To learn diverse representations, the Multi-Head Attention applies **different linear transformations** to the values, keys, and queries for each "head" of attention. This is illustrated in the following figure:



The Multi-Head Attention block

This image captures the overall idea fairly well. Now, let's delve into the details with some PyTorch code:

```
class AttentionHead(nn.Module):
    def __init__(self, d_model, d_feature, dropout=0.1):
        super().__init__()
        self.attn = ScaledDotProductAttention(dropout)
        self.query_tfm = nn.Linear(d_model, d_feature)
        self.key_tfm = nn.Linear(d_model, d_feature)
        self.value_tfm = nn.Linear(d_model, d_feature)
    def forward(self, queries, keys, values, mask=None):
        Q = self.query_tfm(queries)
        K = self.key_tfm(keys)
        V = self.value_tfm(values)
        x = self.attn(Q, K, V)
        return x
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, d_feature,
n_heads, dropout=0.1):
        super().__init__()
        self.d_model = d_model
        self.d_feature = d_feature
        self.n_heads = n_heads
        assert d_model == d_feature * n_heads
        self.attn_heads = nn.ModuleList([
            AttentionHead(d_model,
d_feature, dropout) for _ in range(n_heads)
        ])
        self.projection = nn.Linear(d_feature
* n_heads, d_model)
    def forward(self, queries, keys, values,
mask=None):
        log_size(queries, "Input queries")
        x = [attn(queries, keys, values,
mask=mask)
```

```
                 for i, attn in
enumerate(self.attn_heads)]
         x = torch.cat(x, dim=Dim.feature)
         log_size(x, "concatenated output")
         x = self.projection(x)
         return x
```

As you can see, a single attention head has a very simple structure: it applies a unique linear transformation to its input queries, keys, and values, computes the attention score between each query and key, then uses it to weight the values and sum them up. The Multi-Head Attention block just applies multiple blocks in parallel, concatenates their outputs, then applies one single linear transformation.

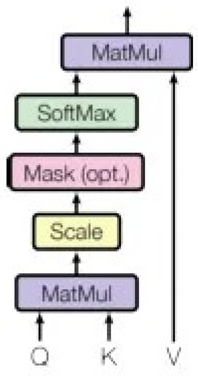
Scaled Dot Product Attention

As for the attention mechansim, the Transformer uses a particular form of attention called the "Scaled Dot-Product Attention" which is computed according to the following equation:

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

where Q is the matrix of queries packed together and K and V are the matrices of keys and values packed together. d_k represents the dimensionality of the queries and keys.

Scaled Dot-Product Attention



The basic attention mechanism is simply a dot product between the query and the key. The size of the dot product tends to grow with the dimensionality of the query and key vectors though, so the Transformer rescales the dot product to prevent it from exploding into huge values.

Here is a PyTorch implementation:

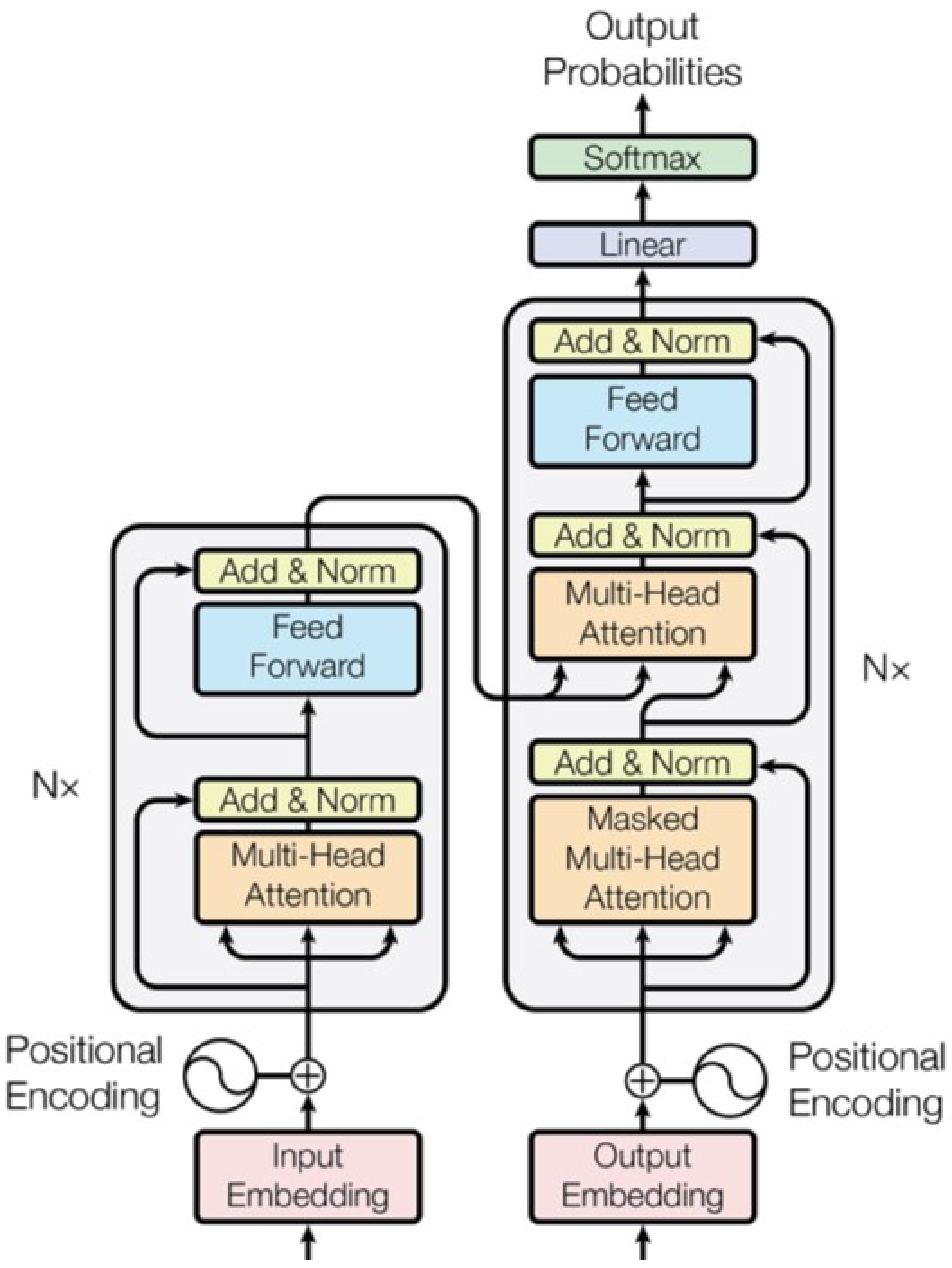
```
from enum import IntEnum
class Dim(IntEnum):
    batch = 0
    seq = 1
    feature = 2
class ScaledDotProductAttention(nn.Module):
    def __init__(self, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
    def forward(self, q, k, v, mask=None):
        d_k = k.size(-1)

        assert q.size(-1) == d_k
        attn = torch.bmm(q,
k.transpose(Dim.seq, Dim.feature))
        attn = attn / math.sqrt(d_k)
        attn = torch.exp(attn)
        if mask is not None: attn =
attn.masked_fill(mask, 0)
        attn = attn / attn.sum(-1,
keepdim=True)
        attn = self.dropout(attn)
        output = torch.bmm(attn, v)
        return output
```

The code above uses a lot of linear algebra/PyTorch tricks, but the essence is simple: for each query, the attention score of each value is the dot product between the query and the corresponding key.

Now, we have (almost) all the components necessary to build the Transformer ourselves.

4. The Overall Architecture

The overall Transformer looks like this (don't be intimidated, we'll dissect this diagram piece by piece):



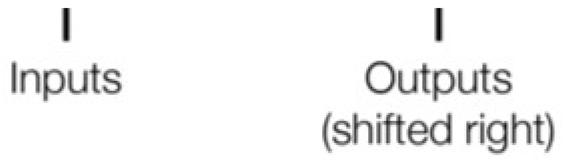
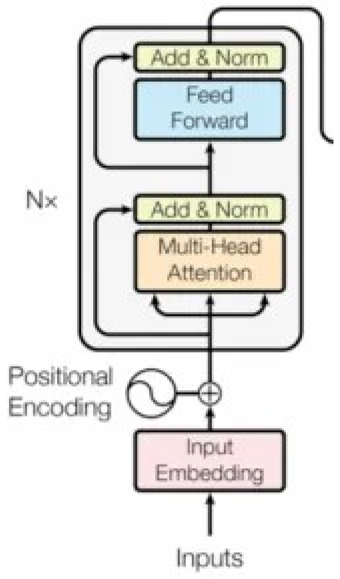


Figure 1: The Transformer - model architecture.

As you can see, the Transformer still uses the basic encoder-decoder design of traditional neural machine translation systems. The left-hand side is the encoder, and the right-hand side is the decoder. The initial inputs to the encoder are the embeddings of the input sequence, and the initial inputs to the decoder are the embeddings of the outputs up to that point. We'll talk about the positional encodings later.

The encoder and decoder are composed of N blocks (where $N = 6$ for both networks), which are composed of smaller blocks as well. Let's look at each block in further detail.

The Encoder



Encoder

The encoder is composed of two blocks (which we will call sub-layers to distinguish from the

N
blocks composing the encoder and decoder).

One is the Multi-Head Attention sub-layer over the inputs, mentioned above. The other is a simple feed-forward network. Between each sub-layer, there is a residual connection followed by a layer normalization. In case you are not familiar, a residual connection is basically just taking the input and adding it to the output of the sub-network, and is a way of making training deep networks easier. Layer normalization is a normalization method in deep learning that is similar to batch normalization (for a more detailed explanation, please refer to this blog post).

Expressed in an equation, it can be written as:

$$y = \mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$$

where
$\mathrm{Sublayer}$
is either the feed-forward network of multi-head attention network.

Here's how we would implement a single Encoder block in PyTorch (using the components we implemented above, of course):

```
class EncoderBlock(nn.Module):
    def __init__(self, d_model=512, d_feature=64,
                 d_ff=2048, n_heads=8, dropout=0.1):
        super().__init__()
        self.attn_head = MultiHeadAttention(d_model, d_feature, n_heads, dropout)

        self.layer_norm1 = LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
        self.position_wise_feed_forward = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model),
        )
        self.layer_norm2 = LayerNorm(d_model)
    def forward(self, x, mask=None):
        att = self.attn_head(x, x, x, mask=mask)
        x = x + self.dropout(self.layer_norm1(att))
        pos = self.position_wise_feed_forward(x)
        x = x + self.dropout(self.layer_norm2(pos))
        return x
```

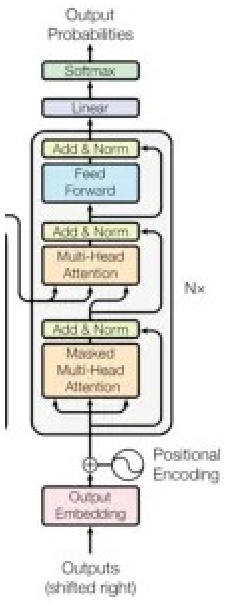
As you can see, what each encoder block is doing is **actually just a bunch of matrix multiplications** followed by a couple of element-wise transformations. This is why the Transformer is so fast: everything is just parallelizable matrix multiplications. The point is that by stacking these transformations on top of each other, we can create a very powerful network. The core of this is the attention mechanism which modifies and attends over a wide range of information.

The entire encoder is very simple to implement once we have the EncoderBlock:

```
class TransformerEncoder(nn.Module):
    def __init__(self, n_blocks=6, d_model=512,
                 n_heads=8, d_ff=2048, dropout=0.1):
        super().__init__()
        self.encoders = nn.ModuleList([
            EncoderBlock(d_model=d_model, d_feature=d_model // n_heads,
                         d_ff=d_ff, dropout=dropout)
            for _ in range(n_blocks)
        ])
    def forward(self, x: torch.FloatTensor, mask=None):
        for encoder in self.encoders:
            x = encoder(x)
        return x
```

See? The Transformer seems very intimidating at first glance, but when we pick it apart it isn't that complex!

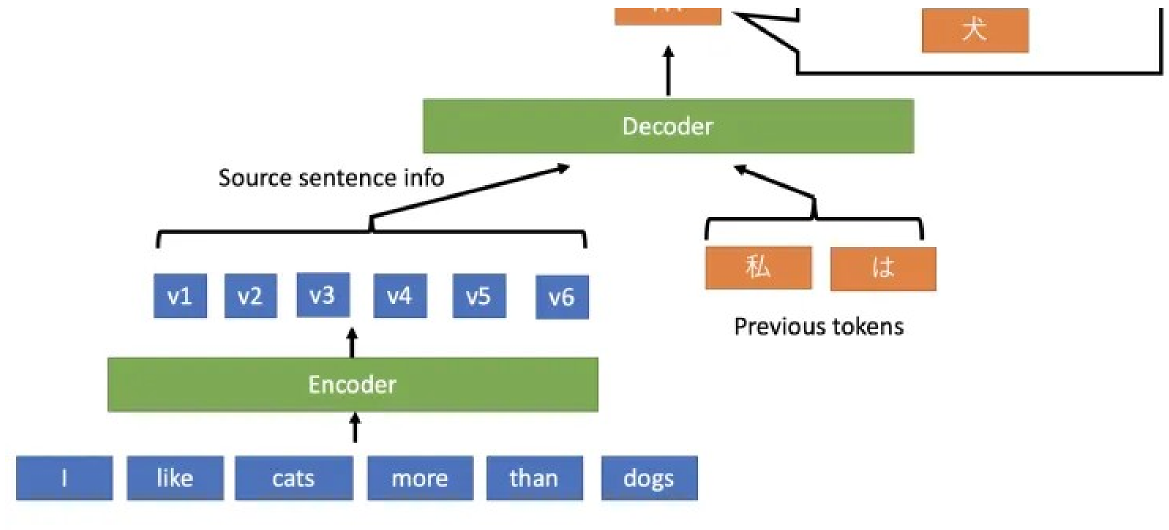
The Decoder



Decoder

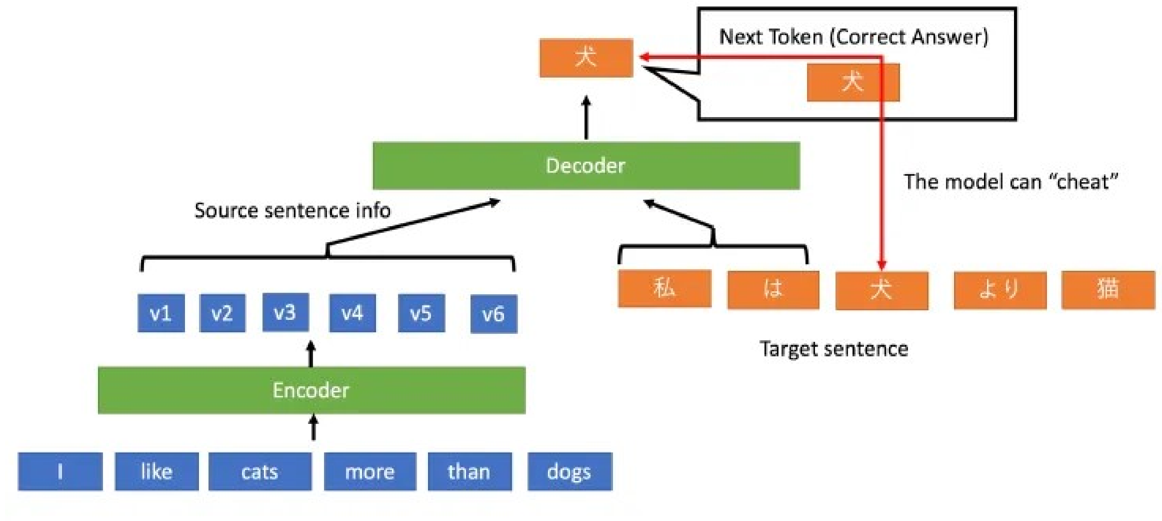
The decoder is very similar to the encoder but has one Multi-Head Attention layer labeled the "masked multi-head attention" network. This network attends over the *previous decoder states*, so plays a similar role to the decoder hidden state in traditional machine translation architectures.

The reason this is called the *masked* multi-head attention block is that we need to mask the inputs to the decoder from future time-steps. Remember, decoders are generally trained to predict sentences based on all the words before the current word. In other words, when we train the network to map the sentence "I like cats more than dogs" to "私は犬よりも猫が好き", we train the network to predict the word "犬" comes after "私は" when the source sentence is "I like cats more than dogs"



The correct way to train a decoder

However, when we train the Transformer, we want to process all the sentences at the same time. When we do this, if we give the decoder access to the entire target sentence, the model can just repeat the target sentence (in other words, it doesn't need to learn anything).



The wrong way to train a decoder

To prevent this from happening, the decoder masks the "future" tokens when decoding a certain word.

Aside from this masking, the Decoder is relatively simple. Here's the code for the DecoderBlock:

```
class DecoderBlock(nn.Module):
    def __init__(self, d_model=512, d_feature=64,
                 d_ff=2048, n_heads=8, dropout=0.1):
        super().__init__()
        self.masked_attn_head = MultiHeadAttention(d_model, d_feature, n_heads, dropout)
        self.attn_head = MultiHeadAttention(d_model, d_feature, n_heads, dropout)
        self.position_wise_feed_forward = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model),
        )
        self.layer_norm1 = LayerNorm(d_model)
        self.layer_norm2 = LayerNorm(d_model)
        self.layer_norm3 = LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
    def forward(self, x, enc_out,
                src_mask=None,
tgt_mask=None):
        att = self.masked_attn_head(x, x, x,
mask=src_mask)
        x = x +
self.dropout(self.layer_norm1(att))
        att = self.attn_head(queries=att,
keys=x, values=x, mask=tgt_mask)
        x = x +
self.dropout(self.layer_norm2(att))
        pos =
self.position_wise_feed_forward(x)
        x = x +
self.dropout(self.layer_norm2(pos))
        return x
```

The code is mostly the same as the EncoderBlock except for one more Multihead Attention Block. Again, once we have the DecoderBlock implemented, the Decoder is very simple.

```
class TransformerDecoder(nn.Module):
    def __init__(self, n_blocks=6,
d_model=512, d_feature=64,
                 d_ff=2048, n_heads=8,
dropout=0.1):
        super().__init__()
        self.position_embedding = PositionalEmbedding(d_model)
        self.decoders = nn.ModuleList([
            DecoderBlock(d_model=d_model, d_feature=d_model // n_heads,
                         d_ff=d_ff, dropout=dropout)
            for _ in range(n_blocks)
        ])
    def forward(self, x: torch.FloatTensor,
                enc_out: torch.FloatTensor,
                src_mask=None, tgt_mask=None):
        for decoder in self.decoders:
            x = decoder(x, enc_out, src_mask=src_mask, tgt_mask=tgt_mask)
        return x
```

We're almost finished now. The final component we need is the positional encoding.

Positional Encodings

Unlike recurrent networks, the multi-head attention network cannot naturally make use of the position of the words in the input sequence. Without positional encodings, the output of the multi-head attention network would be the same for the sentences "I like cats more than dogs" and "I like dogs more than cats". Positional encodings explicitly encode the relative/absolute positions of the inputs as vectors and are then added to the input embeddings.

The paper uses the following equation to compute the positional encodings:

$$PE[pos, 2i] = sin(pos/10000^{2i/d_model})$$
$$PE[pos, 2i+1] = cos(pos/10000^{2i/d_model})$$

where pos represents the position, and i is the dimension. Basically, each dimension of the positional encoding is a wave with a different frequency. This allows the model to easily learn to attend to relative positions, since $PE[pos + k]$ can be represented as a linear function of $PE[pos]$, so the relative positon between different embeddings can be easily inferred. Though the authors attempted to use learned positional encodings, they found that these pre-set encodings performed just as well.

Here's some code to implement the positional encodings:

```
class PositionalEmbedding(nn.Module):
    def __init__(self, d_model, max_len=512):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0,
max_len).unsqueeze(1).float()
        div_term = torch.exp(torch.arange(0,
d_model, 2).float() *
                             -(math.log(10000.0)
/ d_model))
        pe[:, 0::2] = torch.sin(position *
div_term)
        pe[:, 1::2] = torch.cos(position *
div_term)
        pe = pe.unsqueeze(0)
        self.weight = nn.Parameter(pe,
requires_grad=False)
    def forward(self, x):
        return self.weight[:, :x.size(1), :]
```

And this basically finishes our discussion of the Transformer!

6. Practical Techniques

Training Details

Optimizer

The authors used the Adam optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.98$. They used a learning rate schedule where they gradually warmed up the learning rate, then decreased it according to the following formula:

$$lrate = d_{model}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

Residual Dropout

The authors applied dropout to each sublayer before adding it to the original input. They also applied dropout to the sum of the embeddings and to the positional encodings. The dropout rate was 0.1 by default.

Label Smoothing

To penalize the model when it becomes too confident in its predictions, the authors performed label smoothing.

Other techniques

Through experiments, the authors of the papers concluded that the following factors were important in achieving the best performance on the Transformer:

- Choosing a good number of attention heads (both too little and too many heads hurt performance)
- Applying dropout to the output of each sub-layer as well as the attention outputs
- Using a sufficiently large key size d_k for computing attention

The final factor (using a sufficiently large key size) implies that computing the attention weights by determining the compatibility between the keys and queries is a sophisticated task, and a more complex compatibility function than the dot product might improve performance.

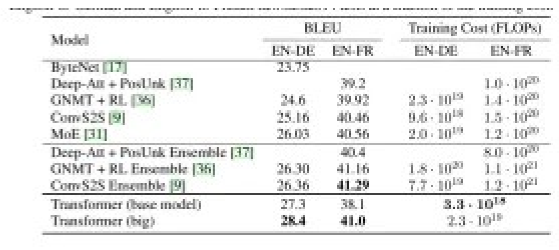
7. Results and Discussion

Detailed results are not the focus of this post, so for details, please refer to the actual paper which summarizes the results and discussion very well.

Here, I will present the most impressive results as well as some practical insights that were inferred from the experiments.

Results for English-to-German translation and English-to-French translation:

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost



Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [17]	23.75			
Deep-Att + PosUnk [37]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [36]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [31]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [37]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [36]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

The Transformer achieves better BLUE scores than previous state-of-the-art models at a fraction of the training cost.

8. Conclusion and Further Readings

This paper demonstrates that attention is a powerful and efficient way to replace recurrent networks as a method of modeling dependencies. This is exciting, as it hints that there are probably far more use cases of attention that are waiting to be explored.

The actual paper gives further details on the hyperparameters and training settings that were necessary to achieve state-of-the-art results, as well as more experimental results on other tasks. If you want to replicate the results or learn about the evaluation in more detail, I highly recommend you go and read it!

Here are some further readings on this paper:

The original paper

The code for the training and evaluation of the model

A Google Research blog post on this architecture