# Transformer: Intuition

We try to briefly explain what each the "moving parts" of the Encoder-Decoder style Transformer is doing.

At the highest level: we have the Encoder and the Decoder.

In the Encoder-Decoder architecture

- the Encoder completes before the Decoder starts

# Encoder

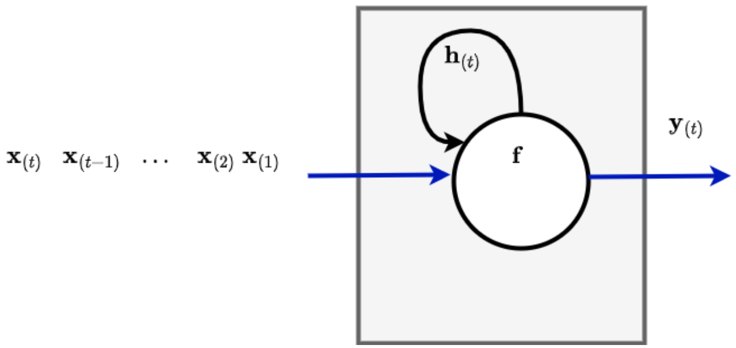
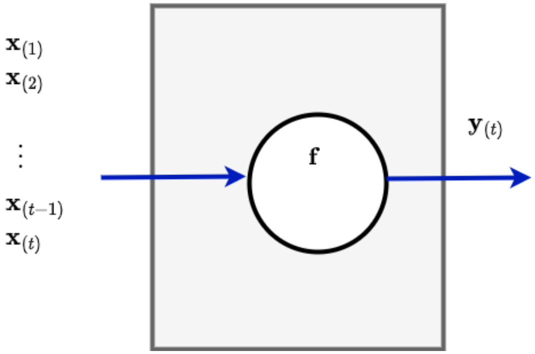
The role of the Encoder is

- to create a Context Sensitive Representation
$$\bar{\mathbf{h}}_{(1:\bar{T})}$$
- of each of the Encoder's input tokens
$$\mathbf{x}_{(1:\bar{T})}$$

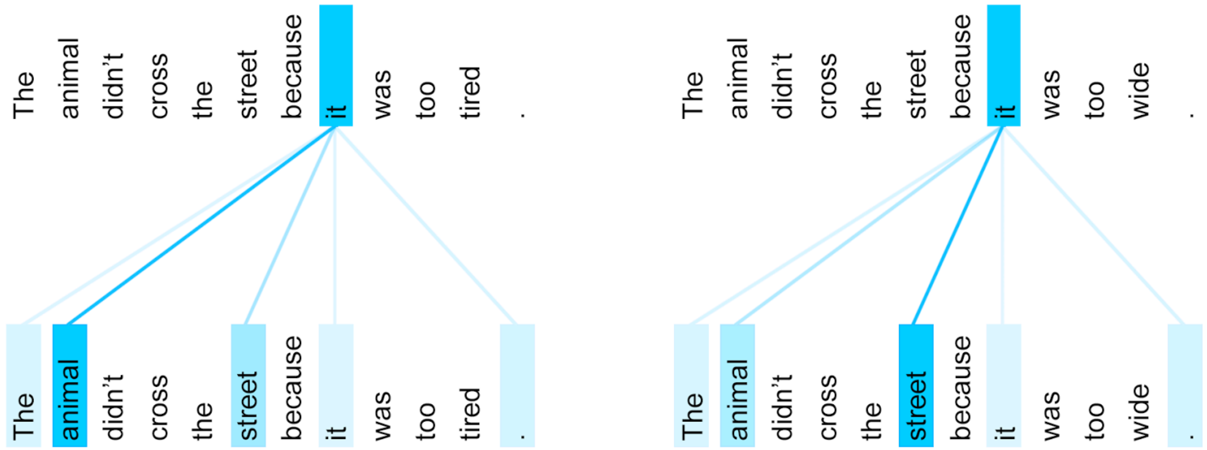
It accomplishes this by the *direct function* approach

- unlike an RNN, it does not process each input token $\mathbf{x}_{(t)}$ sequentially
- it computes $\bar{\mathbf{h}}_{(t)}$ as a function of the entire input $\mathbf{x}_{(1:\bar{T})}$

Encoder Self-Attention is used in the direct function.

| Latent state approach | Direct function approach |
| --- | --- |
|  |  |

By making the meaning dependent on the full context, we can disambiguate the meaning of the world "it"

| Attention weights |
| --- |
| Thickness of the blue lines indicate the strength of attention to other tokens |
|  |
| Picture from: https://1.bp.blogspot.com/-AVGK0ApREtk/WaiAuzddKVI/AAAAAAAAB_A/WPV5ropBU-cxrcMpqJBFHg73K9NX4vywwCLcBGAs/s1600/image2.png |

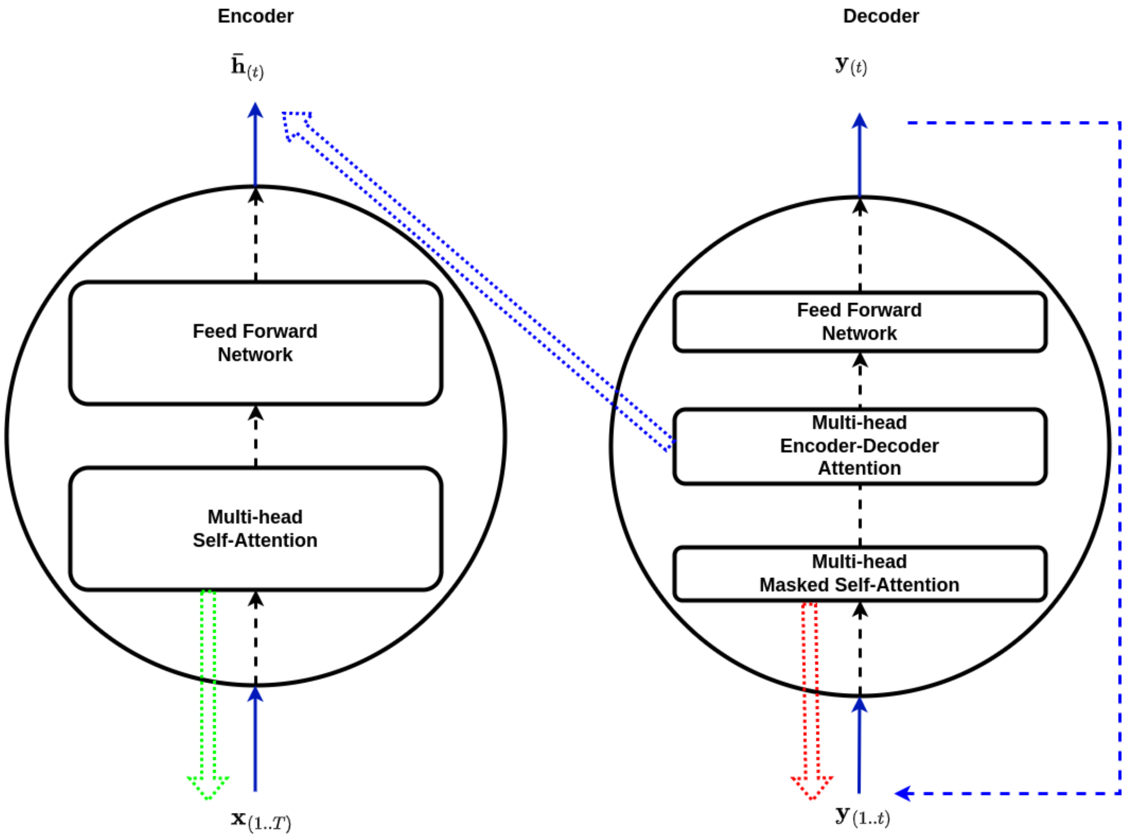
## Decoder

The Decoder works in *auto-regressive* mode

- predicts one output token at a time
- the current output $\hat{\mathbf{y}}_{(t)}$ token is appended to the input for the next position
    - so the input at time step $t$ is
$$\hat{\mathbf{y}}_{(1...t-1)}$$

`

# Encoder/Decoder transformer
## Decoder: Cross-Attention, Auto-regressive mode



It has two inputs at step $t$

- the previously-generated output tokens $t$ is
$$\hat{\mathbf{y}}_{(1...t-1)}$$
- the Encoder output
$$\bar{\mathbf{h}}_{(1:\bar{T})}$$

Self-attention is used on $\hat{\mathbf{y}}_{(1...t-1)}$

Cross-Attention is used on $\bar{\mathbf{h}}_{(1:\bar{T})}$

At step $t$, the Decoder

- uses Self-Attention on $\hat{\mathbf{y}}_{(1...t-1)}$
- to create a *query*
- that is used to attend to $\bar{\mathbf{h}}_{(1:\bar{T})}$

We can think of this use of Self-Attention

- as using a Direct function rather than a loop to implement Sequence to Sequence
    - rather than using the latent state to record
        - what has already been done
        - what is the next step to perform
    - Self-Attention allows direct access to what has already been done: $\hat{\mathbf{y}}_{(1...t-1)}$

The query is used in Cross-Attention

- to attend to the Context Sensitive Representation of the input sequence $\mathbf{x}$

Whatever is returned by Cross-Attention

- is input into the Feed Forward Network (FFN)

Think of the FFN

- as a repository of "world knowledge" accumulated by processing the training data
- "facts"

The FFN produces an output

- which is processed by a Classifier (Linear layer)
- to produce a token in the vocabulary of tokens

That is

- if the vocabulary has $|V|$ tokens
- the Classifier produces a probability distribution vector $\mathbf{p}$ of length $|V|$)
    - such that $\mathbf{p}_j$ is the probability that the output token should be $V_j$

The exact mechanics of this multi-step process

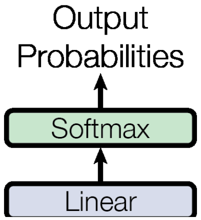
- are controlled by the weights
- that are learned during training

# General

Here is the detailed architecture of the Encoder-Decoder Transformer.

We will review each of the pieces.

**Transformer (Encoder/Decoder)**


Output
Probabilities
Softmax
Linear

Each of the paths in the Transformer is a vector of length $d_{\text{model}}$

- sometimes just referred to as $d$

Having a common length simplifies the architecture

- can stack Transformer blocks (since input and output are same size)
- Self-Attention and Cross-Attention:
    - map a query of size $d$ to an output of size $d$
- Needed for the Residual Connection (Add and Norm)
    - adding the input of Attention to the output of Attention
        - need to be same length

## Residual connections

We observed in passing a curious bit of code in an [earlier module (Functional_Models.ipynb#Residual/skip-connection)](Functional_Models.ipynb#Residual/skip-connection)

- adding the output of a layer to its input

This is called a Residual/Skip connection

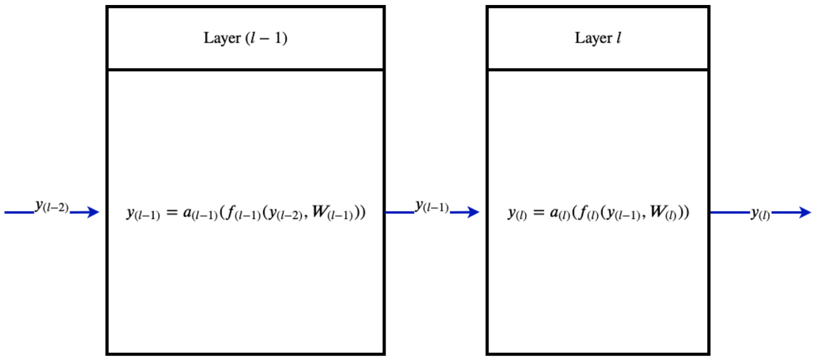
- the input not only goes into a layer
- it also "skips" over the layer
- where the input and output are added

In the lower diagram

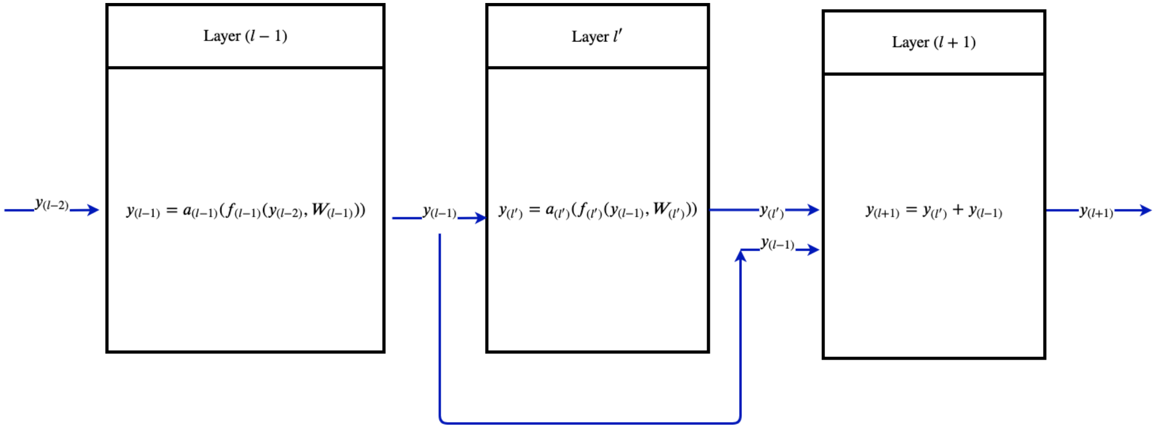
- the input to the middle layer also "skips over" the layer
- and is joined to the middle layer's output in the final layer (which just adds the input and output)

- [Residual connections from Intro course (RNN_Residual_Networks.ipynb)]()

**Network, no Skip Connection**




**Residual Network with Skip Connection**




Suppose we wanted the two networks to compute the same mapping from input $\mathbf{y}_{(l-1)}$ to output

$$\mathbf{y}_{(l+1)} = \mathbf{y}_{(l)}$$

Then

$$
\begin{array}{lll}
\mathbf{y}_{(l+1)} & = & \mathbf{y}_{(l')} + \mathbf{y}_{(l-1)} & \text{definition of } \mathbf{y}_{(l+1)} \text{ in last layer of residual network} \\
\mathbf{y}_{(l)} & = & \mathbf{y}_{(l')} + \mathbf{y}_{(l-1)} & \text{requiring equality of outputs of the two networks } \mathbf{y} \\
\mathbf{y}_{(l')} & = & \mathbf{y}_{(l)} - \mathbf{y}_{(l-1)} & \text{re-arranging terms}
\end{array}
$$

The intermediate layer $l'$ we introduced in the Residual network computes

- the *residual*: of the original networks layer $l$ output wrt to its' input: $(l-1)$ output

Referring to the Transformer diagram above

- the `Add & Norm` layer
- implements the addition of the Multi Head Attention Layer's
    - input
    - output

It is implementing Residual connection

- followed by a Normalization

# Embedding

Words (really: tokens) are *categorical* variables.

Categorical variables are usually encoded as long vectors via One Hot Encoding (OHE)

- very long: number of distinct elements in class
    - e.g., number of words in vocabulary
- *sparse*: only a single non-zero element in the vector

Biggest issue with OHE:

- the similarity (e.g., dot product) of two related words (e.g., "cat", "cats") is zero !
    - same as for two unrelated words (e.g., "cat", "car")

| **word** | **rep(word)** | **Similarity to "dog"** |
|---|---|---|
| dog | [1,0,0,0] | rep(word) · rep(dog) = 1 |
| dogs | [0,1,0,0] | rep(word) · rep(dog) = 0 |
| cat | [0,0,1,0] | rep(word) · rep(dog) = 0 |
| apple | [0,0,0,1] | rep(word) · rep(dog) = 0 |

An *Embedding* is a a *short* and *dense* vector representation of words (tokens).

In addition to being shorter (and dense: many non-zero elements possible) their construction results in

- the similarity of embeddings for two related words being *non-zero*

This makes Embeddings much more valuable for NLP.

| $w$ | $\mathbf{v}_w$ |
|---|---|
| cat | [.7, .5, .01 ] |
| cats | [.7, .5, .95 ] |
| dog | [.7, .2, .01 ] |
| dogs | [.7, .2, .95 ] |
| apple | [.1, .4, .01 ] |
| apples | [.1, .4, .95 ] |

The *Embedding Layer* converts the OHE representation to an Embedding.

See the module from the Intro course (NLP_Embeddings.ipynb) for details.

# Positional Encoding

The Transformer input is a *sequence*

- there is a total ordering between elements based on absolute position

The Transformer needs to be able to discern

- at least: the *relative* ordering of two elements in different positions in the sequence

The *Positional Encoding* layer

- adds a vector that encodes position
- to the Embedding
- such that the Transformer has a representation with both meaning and positions

This is much more involved than simply using an integer to encode the position.

The fundamental operation of a Neural Network is matrix multiplication

- the positional encoding needs to be preserved as it traverses the layers

The details are not trivial.

See the module on Positional Embeddings (Transformer_PositionalEmbedding.ipynb) if you are interested.

# Layer Normalization (part of Add and Norm)

We show in a [module (Training_Neural_Networks_Scaling_and_Initialization.ipynb#Importance-of-unit-variance-across-features)](Training_Neural_Networks_Scaling_and_Initialization.ipynb#Importance-of-unit-variance-across-features) from the Intro course that

- The variance of the *pre-activation distribution* of features grows with the depth of the network.

That is

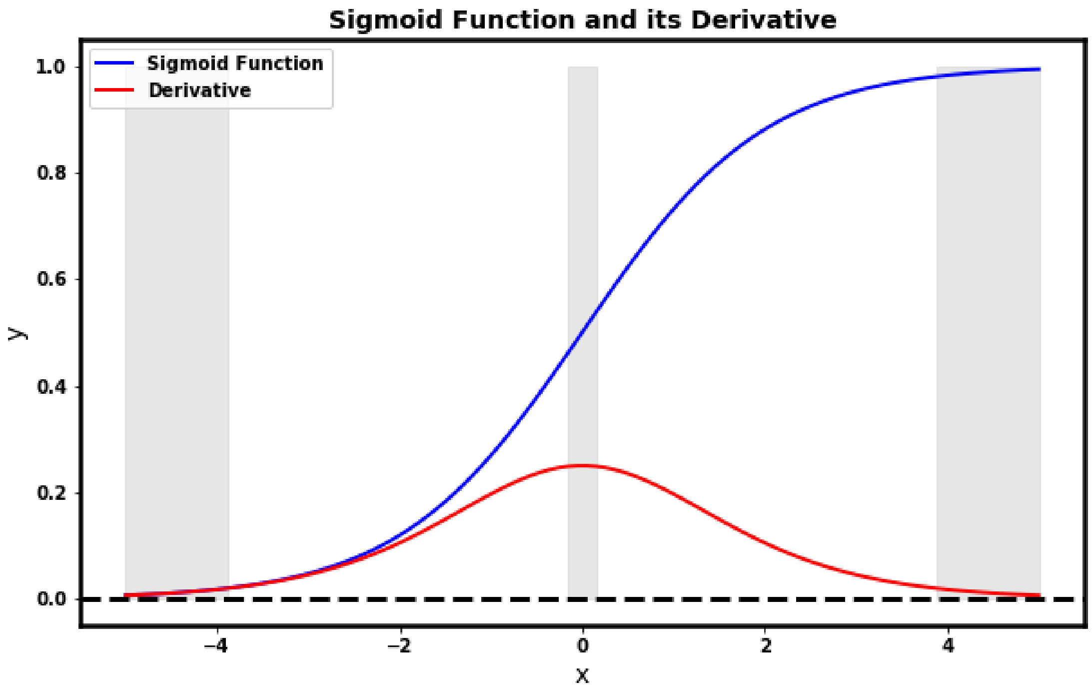
- even if we standardize all the input (Layer $0$) features
- the variance of features in layers $l > 0$ tends to grow

As the variance of the pre-activation gets larger

- we are more likely to be in one of the extremes of the domain of the Activation function
- where derivatives are often near-zero
- and thus: weights don't get updated during Gradient Descent

Hence, we wind up in an unfavorable region of the Activation function.

**Sigmoid and it derivative**
**Shaded regions indicated second derivative near 0**



A [Normalization Layer (Training_Neural_Networks_Scaling_and_Initialization.ipynb#Batch-Normalization-Layer)](Training_Neural_Networks_Scaling_and_Initialization.ipynb#Batch-Normalization-Layer)

- re-normalizes its input features
- to mean $0$ and unit variance

# Feed Forward Network (FFN)

Maps the output of the Decoder-Encoder Attention into the "next output token".

- actually: it is still an embedding of the next token, rather than the true next token
    - that way: it can be appended to the already-generated output to become the Decoder input for next position

This acts as a Classifier

- mapping the input
- to a vector of logits
    - one element per possible element of the Output Vocabulary

There is some evidence that

- the parameters of the FFN are where "world knowledge" is stored
    - every "fact" learned during training

# Linear

This layer is append *only* to the final block in the stacked Transformer blocks.

It acts as a typical Classifier

- "classifies" the final block's output of length $d$
- returning a vector
    - whose length is equal to number of elements of the Vocabulary
    - each element is a logit
        - to be converted into probability distribution over elements of the Vocabulary

## Softmax

Converts the logit for each possible element of the Vocabulary

- into Probability that the element is the next Decoder Output

In [2]:

```
print("Done")
```

```
Done
```