STA 414/2104: Statistical Methods for Machine Learning II

Week 12 Neural Networks

Michal Malyska

University of Toronto

Today

- What are Neural Networks?

Today

- What are Neural Networks?
- Neural Network Building Blocks

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers
  - Recurrent Layers

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers
  - Recurrent Layers
  - **Attention**

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers
  - Recurrent Layers
  - **Attention**
- Neural Networks

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers
  - Recurrent Layers
  - **Attention**
- Neural Networks
  - Feed Forward (Multi Layer Perceptron)

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers
  - Recurrent Layers
  - **Attention**
- Neural Networks
  - Feed Forward (Multi Layer Perceptron)
  - Recurrent

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers
  - Recurrent Layers
  - **Attention**
- Neural Networks
  - Feed Forward (Multi Layer Perceptron)
  - Recurrent
  - **Transformer**

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers
  - Recurrent Layers
  - **Attention**
- Neural Networks
  - Feed Forward (Multi Layer Perceptron)
  - Recurrent
  - **Transformer**
- Transformer

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers
  - Recurrent Layers
  - **Attention**
- Neural Networks
  - Feed Forward (Multi Layer Perceptron)
  - Recurrent
  - **Transformer**
- Transformer
  - Encoder

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers
  - Recurrent Layers
  - **Attention**
- Neural Networks
  - Feed Forward (Multi Layer Perceptron)
  - Recurrent
  - **Transformer**
- Transformer
  - Encoder
  - Decoder

Today

- What are Neural Networks?
- Neural Network Building Blocks
  - Linear (Feed Forward) Layers
  - Activation Functions
  - Residual Layers
  - Recurrent Layers
  - **Attention**
- Neural Networks
  - Feed Forward (Multi Layer Perceptron)
  - Recurrent
  - **Transformer**
- Transformer
  - Encoder
  - Decoder
  - Positional Encoding

What are Neural Networks

Neural networks are what we commonly call any differentiable function that can be expressed as a computation graph. Each node is a primitive operation (e.g. matrix multiplication) and edges represent data flow. In particular, a simple (and quite common) case is where this graph is a chain. Individual nodes, or pre-defined sequences are often referred to as **layers**

Building Blocks of Neural Networks

Linear (Feed Forward) Layers - is the simplest possible type of layer, it consists of 2 operations:

- Matrix multiplication

Building Blocks of Neural Networks

Linear (Feed Forward) Layers - is the simplest possible type of layer, it consists of 2 operations:

- Matrix multiplication
- Vector addition

Building Blocks of Neural Networks

Linear (Feed Forward) Layers - is the simplest possible type of layer, it consists of 2 operations:

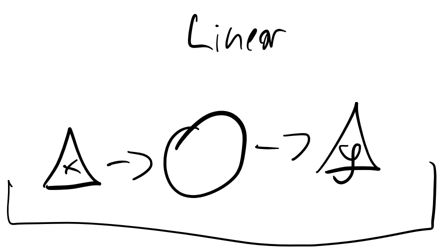
- Matrix multiplication
- Vector addition

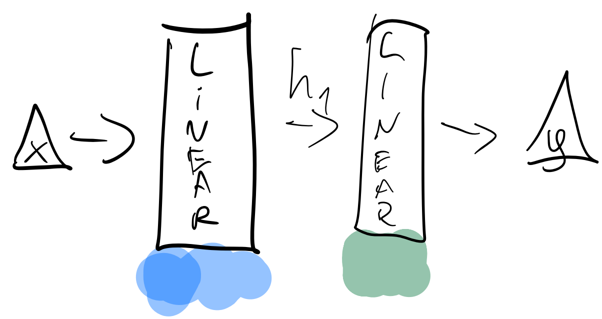
Linear (Feed Forward) Layers - is the simplest possible type of layer, it consists of 2 operations:

- Matrix multiplication
- Vector addition

$$y = f(x; \theta) = Wx + b$$

where θ is the set of parameters $\{W, b\}$





Building Blocks of Neural Networks

- What would happen if we followed up a Linear Layer by another linear layer?

$$y = f(g(x;\theta_1);\theta_2) = W_2(W_1 x + b_1) + b_2 =$$
$$= (W_2 W_1)x + (W_2 b_1 + b_2)$$

Building Blocks of Neural Networks

- What would happen if we followed up a Linear Layer by another linear layer?

$$
\begin{aligned}
y = f(g(x;\theta_1);\theta_2) &= W_2(W_1 x + b_1) + b_2 = \\
&= (W_2 W_1)x + (W_2 b_1 + b_2)
\end{aligned}
$$

Building Blocks of Neural Networks

- What would happen if we followed up a Linear Layer by another linear layer?

$$y = f(g(x;\theta_1);\theta_2) = W_2(W_1x + b_1) + b_2 =$$
$$= (W_2W_1)x + (W_2b_1 + b_2)$$

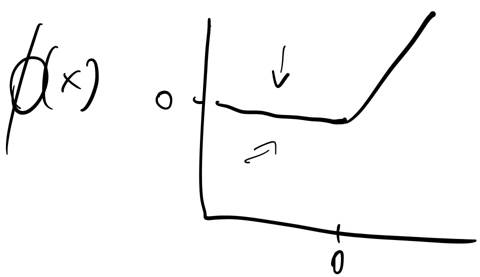
Ok, not very useful. Is there anything we can do about it?

- What would happen if we followed up a Linear Layer by another linear layer?

$$y = f(g(x;\theta_1);\theta_2) = W_2(W_1 x + b_1) + b_2 =$$
$$= (W_2 W_1)x + (W_2 b_1 + b_2)$$

Ok, not very useful. Is there anything we can do about it? **Yes**, to get more expressive power, we can apply a non-linear (element-wise) transformation. We call these functions **Activation functions**. Some common examples include:

- **Rec**tified **L**inear **U**nit: $\phi(x) = max(0, x)$ ReLU



$y = f(\phi(g(x;\theta_1));\theta_2)$

Building Blocks of Neural Networks

- What would happen if we followed up a Linear Layer by another linear layer?

$$y = f(g(x;\theta_1);\theta_2) = W_2(W_1 x + b_1) + b_2 =$$
$$= (W_2 W_1)x + (W_2 b_1 + b_2)$$

Ok, not very useful. Is there anything we can do about it? **Yes**, to get more expressive power, we can apply a non-linear (element-wise) transformation. We call these functions **Activation functions**. Some common examples include:

- **Re**ctified **L**inear **U**nit: $\phi(x) = max(0, x)$
- **Sigmoid**: $\phi(x) = \sigma(x) = \frac{1}{1+e^{-x}}$

Activations Examples

Building Blocks of Neural Networks

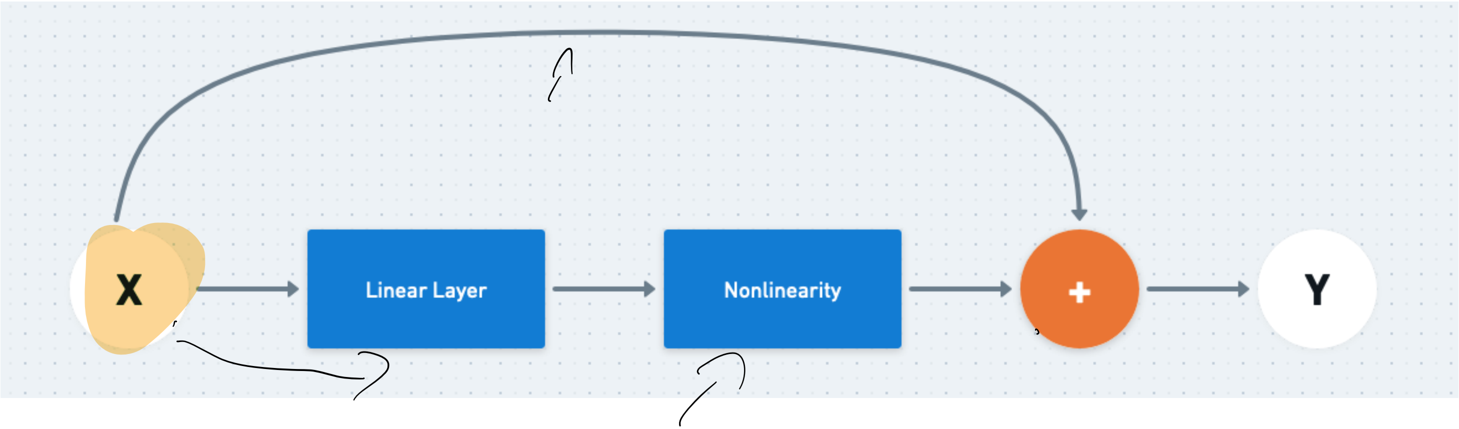
If we begin stacking large number of layers together, the signal may get squashed to zero, or blow up to infinity. Similar problem often happens during the gradient computation back through the graph.

Building Blocks of Neural Networks

If we begin stacking large number of layers together, the signal may get squashed to zero, or blow up to infinity. Similar problem often happens during the gradient computation back through the graph. To reduce the effect of those problems we often propagate the signal to layers further downstream, in what are called **residual connections**

Building Blocks of Neural Networks

If we begin stacking large number of layers together, the signal may get squashed to zero, or blow up to infinity. Similar problem often happens during the gradient computation back through the graph. To reduce the effect of those problems we often propagate the signal to layers further downstream, in what are called **residual connections**



$$y = f(x;\theta) = \phi(Wx + b) + x$$

Building Blocks of Neural Networks

When it comes to modelling sequences (e.g. text, or time series data), it is often useful to make the model stateful in order for it to help "carry" the information through the graph. To do that we simply add a state at timepoint t: s_t, and computing the output and the new state using some function:

Building Blocks of Neural Networks

When it comes to modelling sequences (e.g. text, or time series data), it is often useful to make the model stateful in order for it to help "carry" the information through the graph. To do that we simply add a state at timepoint t: s_t, and computing the output and the new state using some function:

$$(y, s_{t+1}) = f(x, s_t)$$

Building Blocks of Neural Networks

When it comes to modelling sequences (e.g. text, or time series data), it is often useful to make the model stateful in order for it to help "carry" the information through the graph. To do that we simply add a state at timepoint t: s_t, and computing the output and the new state using some function:

$$(y, s_{t+1}) = f(x, s_t)$$

This is then called a **recurrent layer**.

Building Blocks of Neural Networks

When it comes to modelling sequences (e.g. text, or time series data), it is often useful to make the model stateful in order for it to help "carry" the information through the graph. To do that we simply add a state at timepoint t: s_t, and computing the output and the new state using some function:

$$(y, s_{t+1}) = f(x, s_t)$$

This is then called a **recurrent layer**.

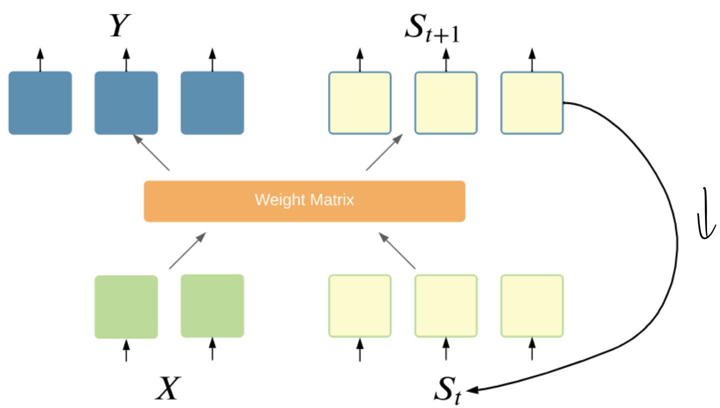
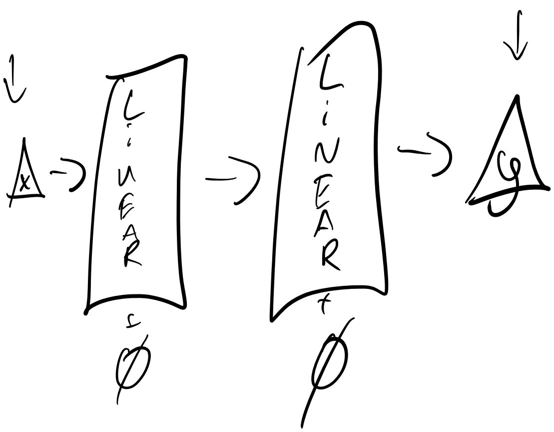


Figure 16.8: Recurrent layer.

Common Architectures

FFNN

A very common type of neural net architecture is a **Feed Forward Neural Network**, also sometimes called a **Multi Layer Perceptron**. It simply consists of a sequence of linear (FF) layers, with nonlinearities between them. MLP



$(y - \hat{y})^2$

Common Architectures

A very common type of neural net architecture is a **Feed Forward Neural Network**, also sometimes called a **Multi Layer Perceptron**. It simply consists of a sequence of linear (FF) layers, with nonlinearities between them.

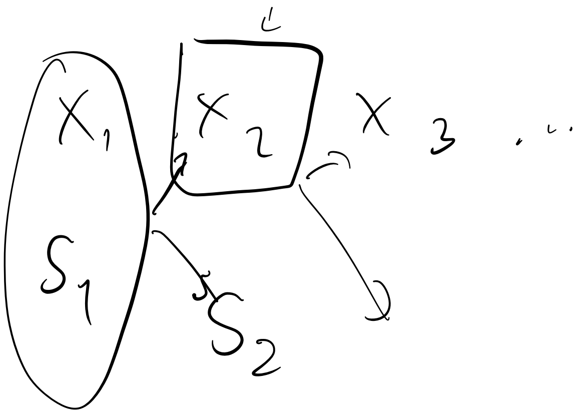
$$f(x;\theta) = \phi(W_L(\phi(W_{L-1}(\phi(W_{L-2}(\dots) + b_{L-2})) + b_{L-1})) + b_L)$$

Common Architectures

If we use recurrent layers in our neural network, the outcome is what we typically call a **Recurrent Neural Network**, (of which there are many variants). In the simplest possible option the function $f(x, h)$ is a simple FFNN.

Common Architectures

If we use recurrent layers in our neural network, the outcome is what we typically call a **Recurrent Neural Network**, (of which there are many variants). In the simplest possible option the function $f(x, h)$ is a simple FFNN. When training RNNs each item in a sequence is used as input, however during inference each item in the sequence will depend on previous predictions.



GRU

LSTM

Common Architectures

If we use recurrent layers in our neural network, the outcome is what we typically call a **Recurrent Neural Network**, (of which there are many variants). In the simplest possible option the function $f(x, h)$ is a simple FFNN. When training RNNs each item in a sequence is used as input, however during inference each item in the sequence will depend on previous predictions.

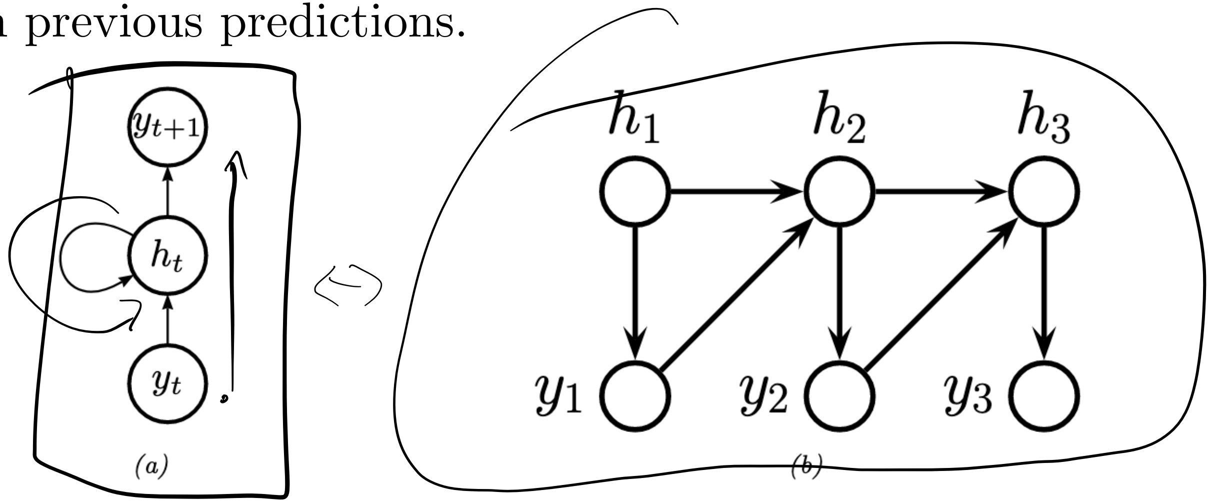
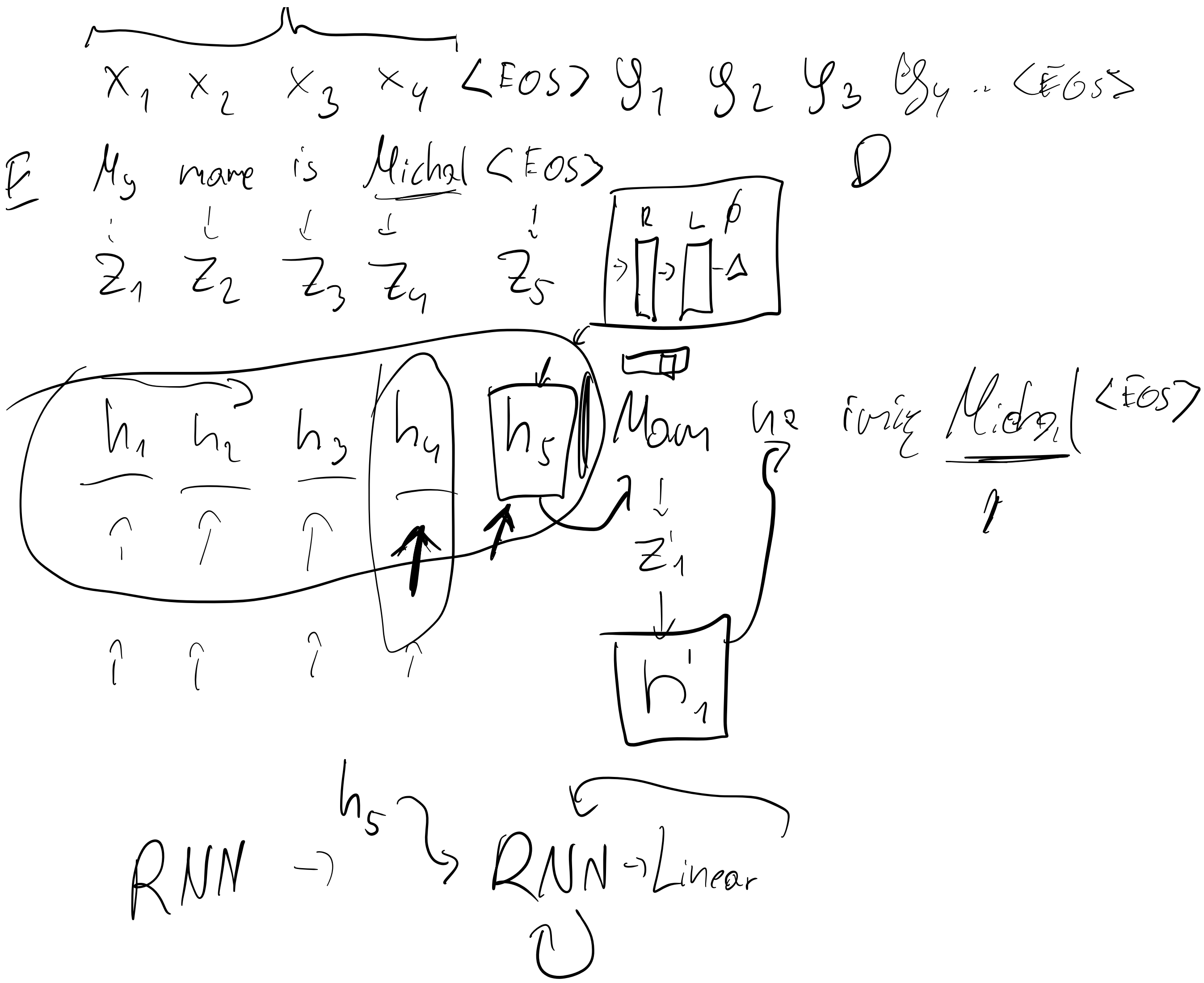


Figure 16.12: Illustration of a recurrent neural network (RNN). (a) With self-loop. (b) Unrolled in time.


x1 x2 x3 x4 <EOS> y1 y2 y3 y4 .. <EOS>
E
My name is Michal <EOS>
D
z1 z2 z3 z4 z5
R L p
h1 h2 h3 h4 h5
Mam na imię Michał <EOS>
z'1
h'1
RNN
h5
RNN
Linear

Attention is all you need

What if instead of getting just the previous hidden state we were able to take a look at a lot of the previous inputs at once?

Attention is all you need

What if instead of getting just the previous hidden state we were able to take a look at a lot of the previous inputs at once? We could combine all the previous hidden states. But can we do better?

Attention is all you need

What if instead of getting just the previous hidden state we were able to take a look at a lot of the previous inputs at once? We could combine all the previous hidden states. But can we do better? We can score each of the hidden states by how well it is associated with the state we will be predicting.

Attention is all you need

What if instead of getting just the previous hidden state we were able to take a look at a lot of the previous inputs at once? We could combine all the previous hidden states. But can we do better? We can score each of the hidden states by how well it is associated with the state we will be predicting. At a high level the attention mechanism consists of 3 simple steps:

1. Generate a score for each of the hidden states

Attention is all you need

What if instead of getting just the previous hidden state we were able to take a look at a lot of the previous inputs at once? We could combine all the previous hidden states. But can we do better? We can score each of the hidden states by how well it is associated with the state we will be predicting. At a high level the attention mechanism consists of 3 simple steps:

1. Generate a score for each of the hidden states
2. Apply the softmax function to the scores

$x_1 \quad x_2 \quad x_3 \quad x_4$

$1 \quad 1 \quad 1 \quad 5$

$\Rightarrow 0.05 \quad 0.05 \quad 0.05 \quad 0.85$

Attention is all you need

What if instead of getting just the previous hidden state we were able to take a look at a lot of the previous inputs at once? We could combine all the previous hidden states. But can we do better? We can score each of the hidden states by how well it is associated with the state we will be predicting. At a high level the attention mechanism consists of 3 simple steps:

1. Generate a score for each of the hidden states
2. Apply the softmax function to the scores
3. Multiply each of the hidden states by the output of the softmax and add them together.

Attention is all you need

Now the hard problem remains: how do we score each of the hidden states?

Attention is all you need

Now the hard problem remains: how do we score each of the hidden states? We will begin by creating 3 separate embeddings from each of our inputs, by simply multipling them by (learned) matrices:

$$q = W^Q x$$
$$k = W^K x$$
$$v = W^V x$$

Attention is all you need

Now the hard problem remains: how do we score each of the hidden states? We will begin by creating 3 separate embeddings from each of our inputs, by simply multipling them by (learned) matrices:

$$q = W^Q x$$
$$k = W^K x$$
$$v = W^V x$$

We then define the **Attention Layer** as:

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

Where α is the scoring function.

(Dot product) Attention is all you need

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

(Dot product) Attention is all you need

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

The most common choice of the attention function is called the **dot product attention**. We obtain the scores by a normalized dot product of the k and q vectors.

(Dot product) Attention is all you need

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

The most common choice of the attention function is called the **dot product attention**. We obtain the scores by a normalized dot product of the k and q vectors.

$$b(q, k) = \frac{q^T k}{\sqrt{d}}$$

(Dot product) Attention is all you need

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i)v_i$$

The most common choice of the attention function is called the **dot product attention**. We obtain the scores by a normalized dot product of the k and q vectors.

$$b(q, k) = \frac{q^T k}{\sqrt{d}}$$

where d is a normalizing constant, usually the dimensionality of the vectors. We then set our attention weights α_i to be the softmax of all the scores:

(Dot product) Attention is all you need

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

The most common choice of the attention function is called the **dot product attention**. We obtain the scores by a normalized dot product of the k and q vectors.

$$b(q, k) = \frac{q^T k}{\sqrt{d}}$$

where d is a normalizing constant, usually the dimensionality of the vectors. We then set our attention weights α_i to be the softmax of all the scores:

$$\alpha_i(q, k_i) = \frac{exp(b(q, k_i))}{\sum_{j=1}^{m} exp(b(q, k_j))}$$

(Dot product) Attention is all you need

The entire process then reduces to:

$$\text{Attn}(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

(Dot product) Attention is all you need

The entire process then reduces to:

$$Y = Attn(Q, K, V) = \sigma(\frac{QK^T}{\sqrt{d}})V$$

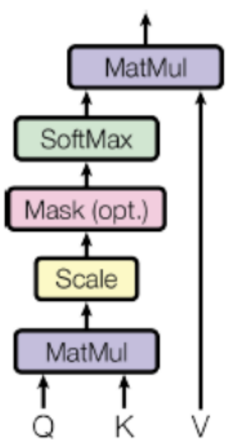
$$= \sigma(\frac{W^Q X (W^K X)^T}{\sqrt{d}}) W^V X$$

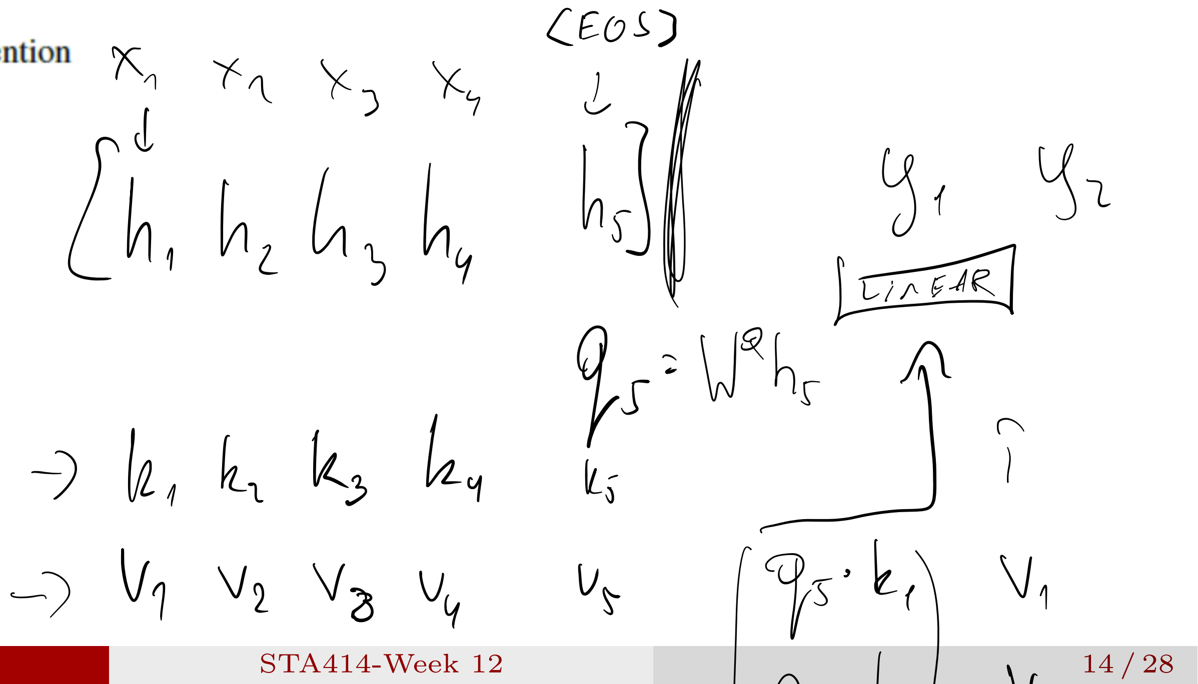
(Dot product) Attention is all you need

The entire process then reduces to:

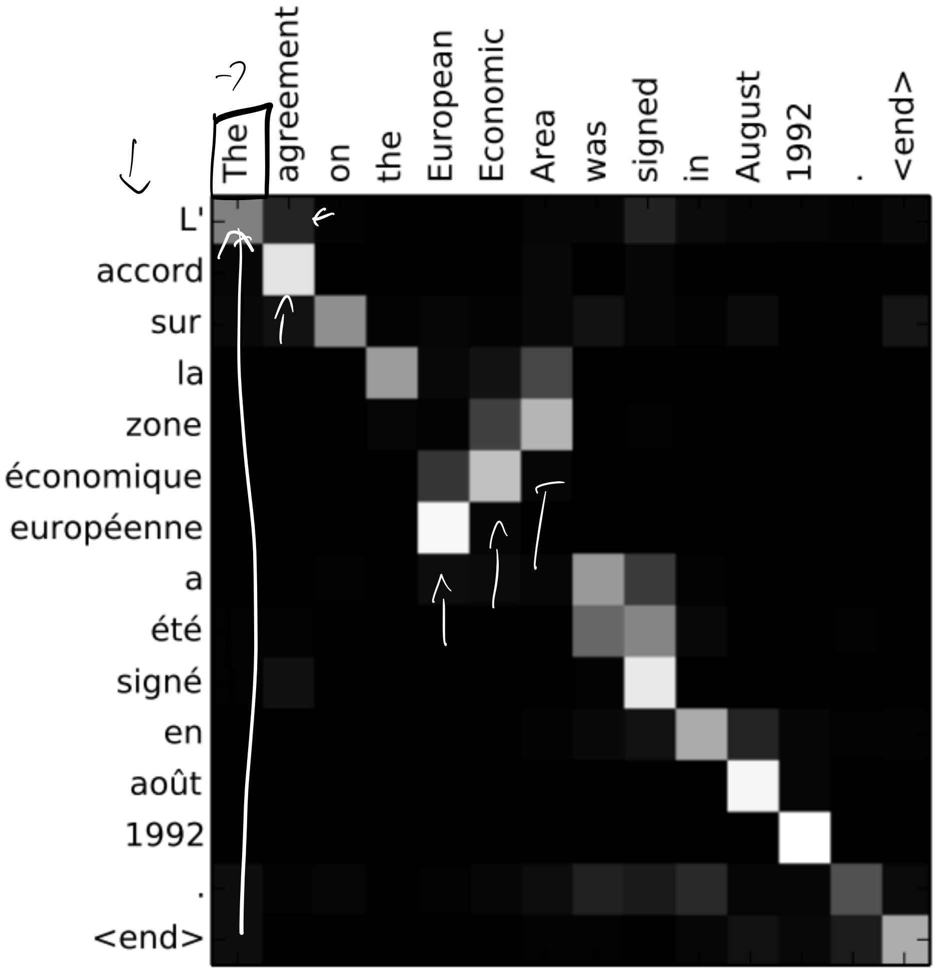
$$Y = Attn(Q, K, V) = \sigma(\frac{QK^T}{\sqrt{d}})V$$
$$= \sigma(\frac{W^Q X (W^K X)^T}{\sqrt{d}}) W^V X$$

Scaled Dot-Product Attention





Attention Visualization



Connection to GPs

Going back to non-parametric kernl based methods (e.g. GPs), we compare the input x to each of the training examples X using a kernel to get a vector of similarity scores $\alpha = |K(x, x_i)|_{i=1}^{m}$, which we then use to retrieve a weighted combination of the corresponding target values y_i as :

Connection to GPs

Going back to non-parametric kernl based methods (e.g. GPs), we compare the input x to each of the training examples X using a kernel to get a vector of similarity scores $\alpha = |K(x, x_i)|_{i=1}^{m}$, which we then use to retrieve a weighted combination of the corresponding target values y_i as :

$$\hat{y} = \sum_{i=1}^{m} \alpha_i y_i$$

Connection to GPs

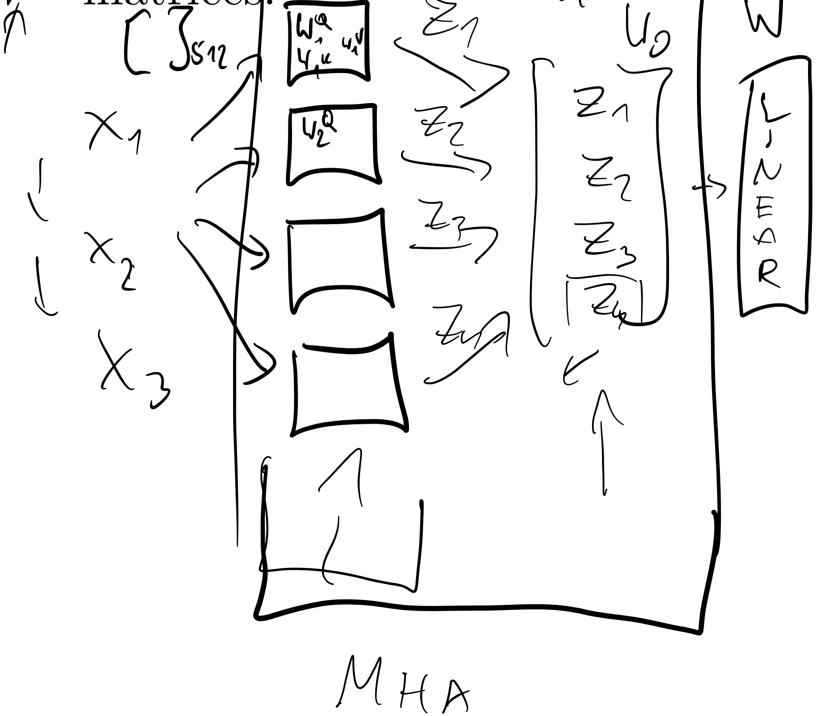
Going back to non-parametric kernl based methods (e.g. GPs), we compare the input x to each of the training examples X using a kernel to get a vector of similarity scores $\alpha = |K(x, x_i)|_{i=1}^{m}$, which we then use to retrieve a weighted combination of the corresponding target values y_i as :

$$\hat{y} = \sum_{i=1}^{m} \alpha_i y_i$$

If we replace the stored examples matrix X with a learned embedding $K = W^K X$, stored outputs with $V = W^V Y$, and create an input embedding $q = W^Q x$, we can arrive at attention!

In practice it is advantageous to have multiple "attention heads" each with a different set of W^Q, W^K, W^V matrices.

- Why do you think that is?



Multi Head Attention and Self Attention

In practice it is advantageous to have multiple "attention heads" each with a different set of W^Q, W^K, W^V matrices.

- Why do you think that is?
- Do we really need all of them?

Multi Head Attention and Self Attention

In practice it is advantageous to have multiple "attention heads" each with a different set of W^Q, W^K, W^V matrices.

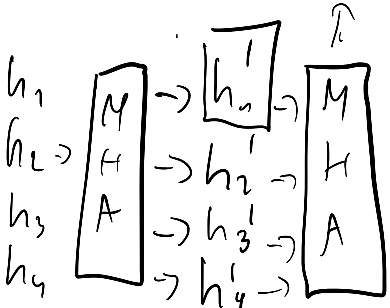
- Why do you think that is?
- Do we really need all of them?

In practice it is advantageous to have multiple "attention heads" each with a different set of W^Q, W^K, W^V matrices.

- Why do you think that is?
- Do we really need all of them?

We then simply concatenate the outputs of all of the attention heads together and multiplied by one final matrix W^O that is learned as well, this is called **Multi Head Attention**.

$$
\begin{aligned}
o = MHA(Q, K, V) &= Concat(h_1, \ldots, h_h)W^O \\
&= Concat(Attn(Q_1, K_1, V_1), \ldots, Attn(Q_h, K_h, V_h))W^O
\end{aligned}
$$



Multi Head Attention and Self Attention

In practice it is advantageous to have multiple "attention heads" each with a different set of W^Q, W^K, W^V matrices.

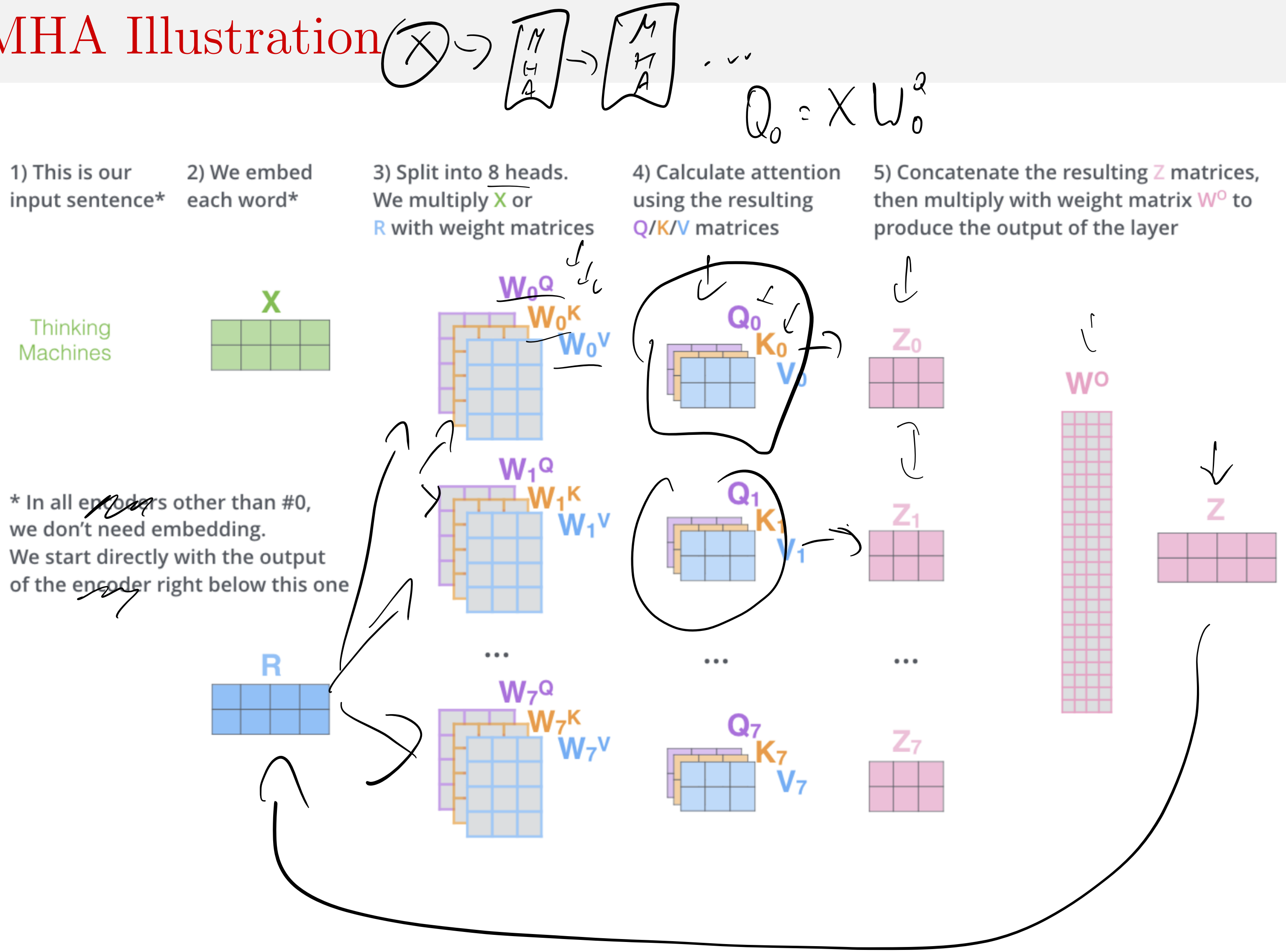
- Why do you think that is?
- Do we really need all of them?

We then simply concatenate the outputs of all of the attention heads together and multiplied by one final matrix W^O that is learned as well, this is called **Multi Head Attention**.

$$
\begin{aligned}
MHA(Q, K, V) &= Concat(h_1, \ldots, h_h)W^O \\
&= Concat(Attn(Q_1, K_1, V_1), \ldots, Attn(Q_h, K_h, V_h))W^O
\end{aligned}
$$

Additionally, we can stack several identical Attention / MHA blocks on top each other. This is called **Self-Attention**

MHA Illustration



Transformer

First proposed in a 2017 paper "Attention is all you need", the **Transformer** architecture consists of two stacks (called **Encoder** and **Decoder**) of blocks:

Transformer

First proposed in a 2017 paper "Attention is all you need", the **Transformer** architecture consists of two stacks (called **Encoder** and **Decoder**) of blocks:

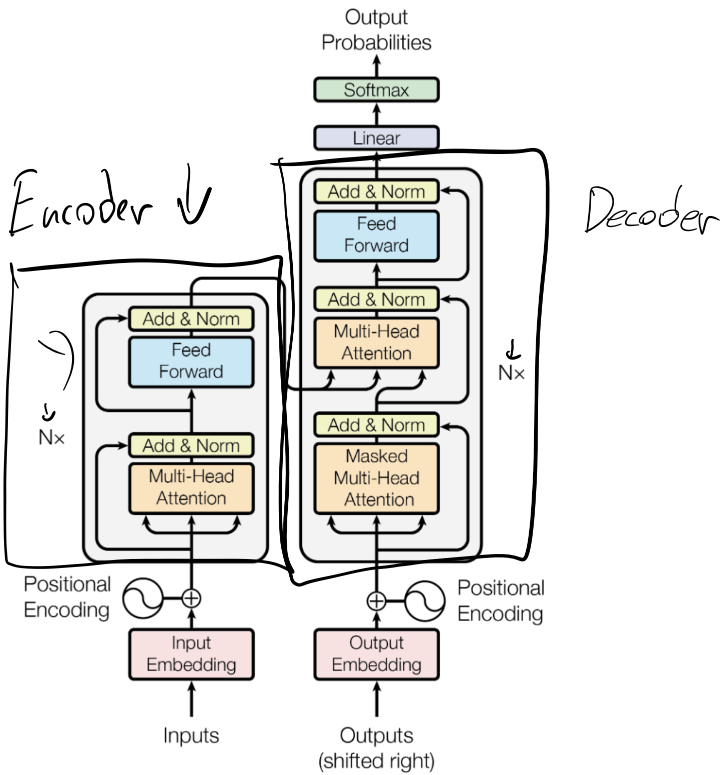
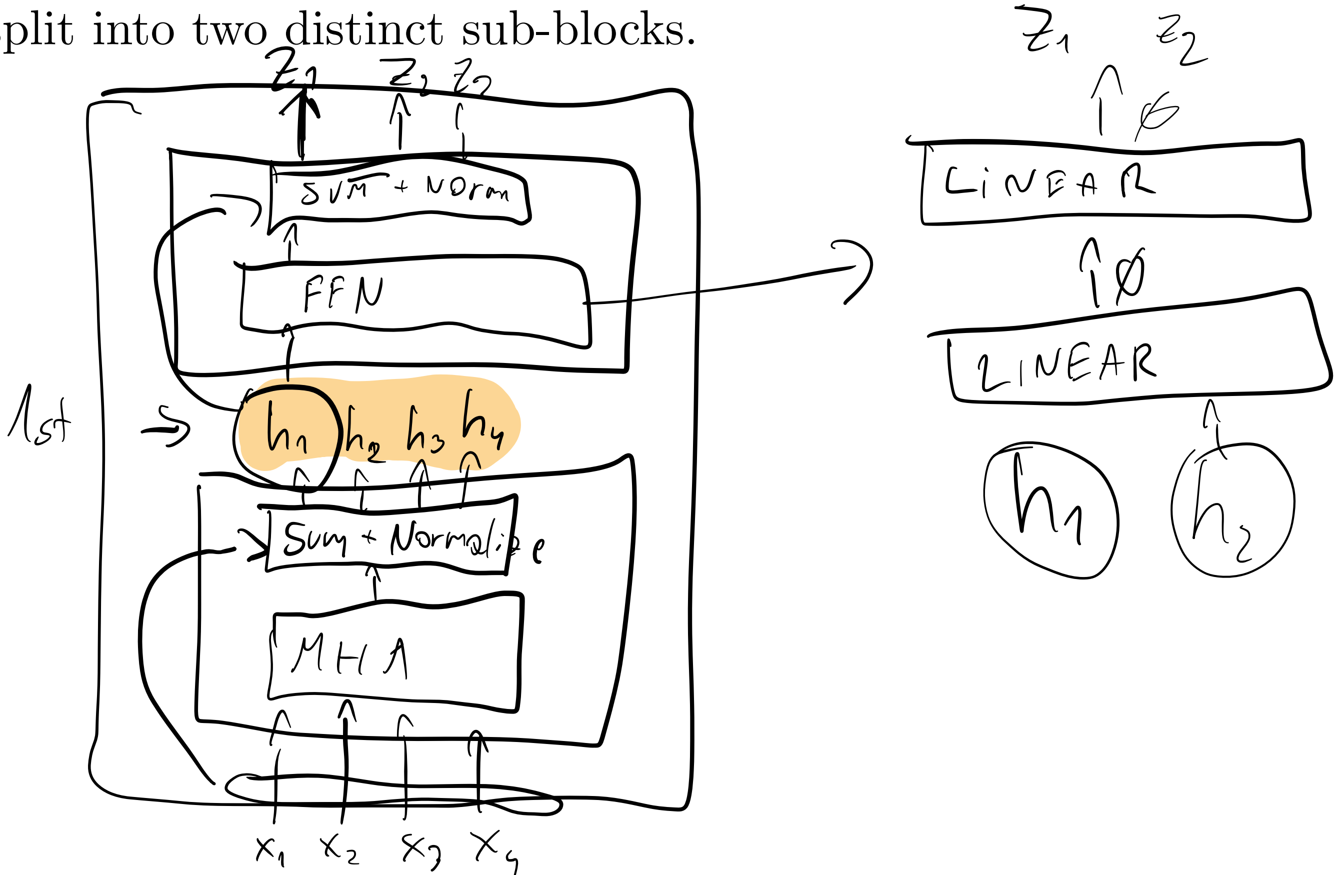
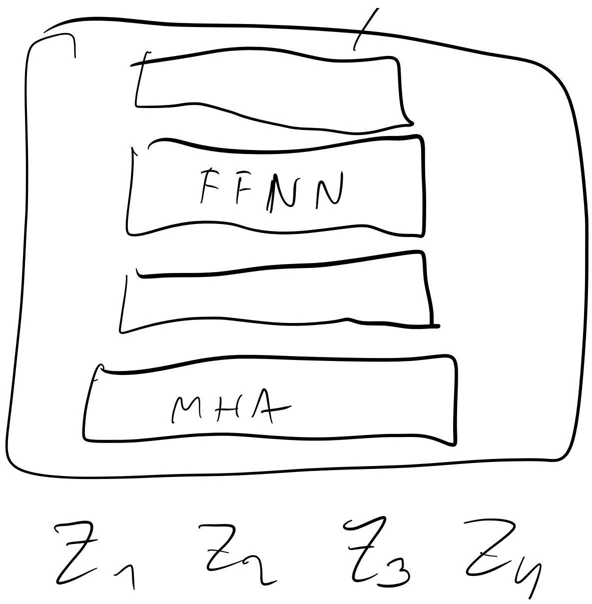


Figure 1: The Transformer - model architecture.

The **Encoder** consists of a stack of 6 blocks. Each block is further split into two distinct sub-blocks.



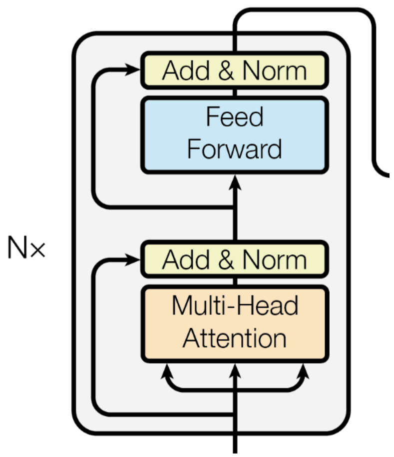

FFNN
MHA
Z_1 Z_2 Z_3 Z_4

Transformer

The **Encoder** consists of a stack of 6 blocks. Each block is further split into two distinct sub-blocks.
The first is a Multi Head Self Attention mechanism, and the second is a simple FFNN. Both of the sub-blocks have a residual connection around them, followed by normalization.

Transformer

The **Encoder** consists of a stack of 6 blocks. Each block is further split into two distinct sub-blocks.
The first is a Multi Head Self Attention mechanism, and the second is a simple FFNN. Both of the sub-blocks have a residual connection around them, followed by normalization.



Transformer

Similarily, the **Decoder** is also a stack of 6 blocks.

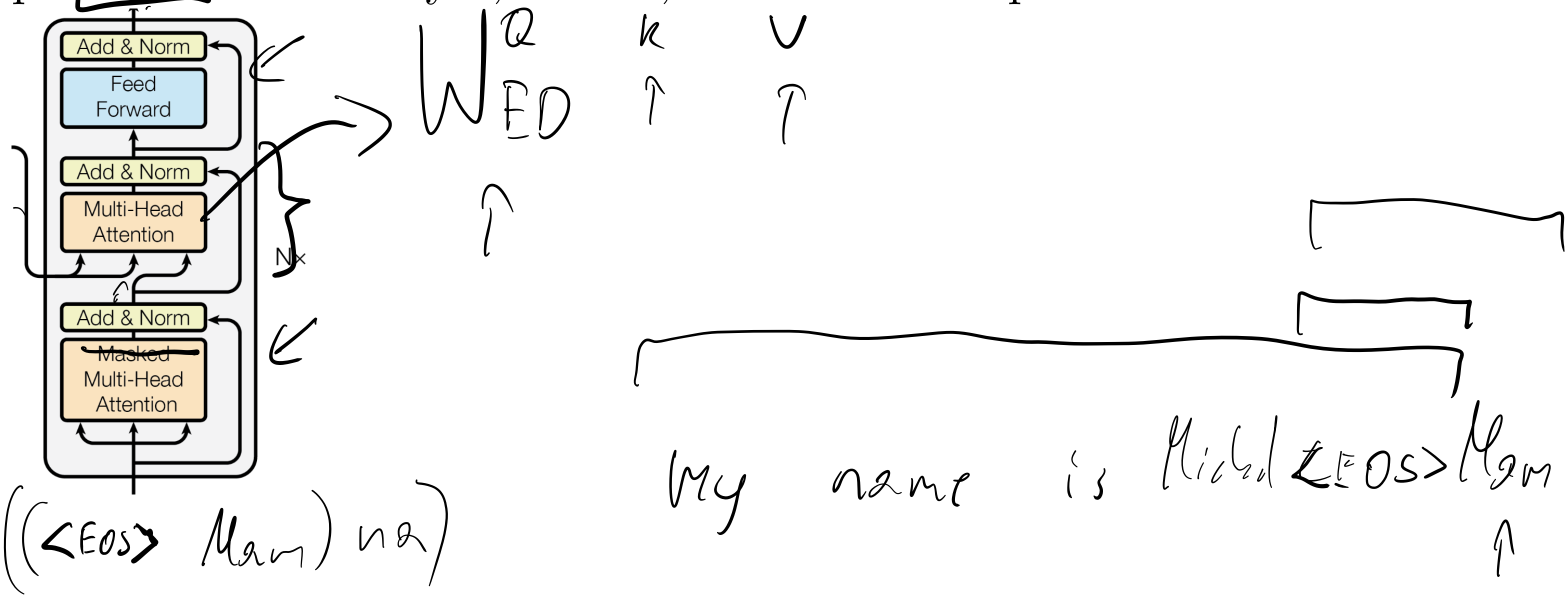
Transformer

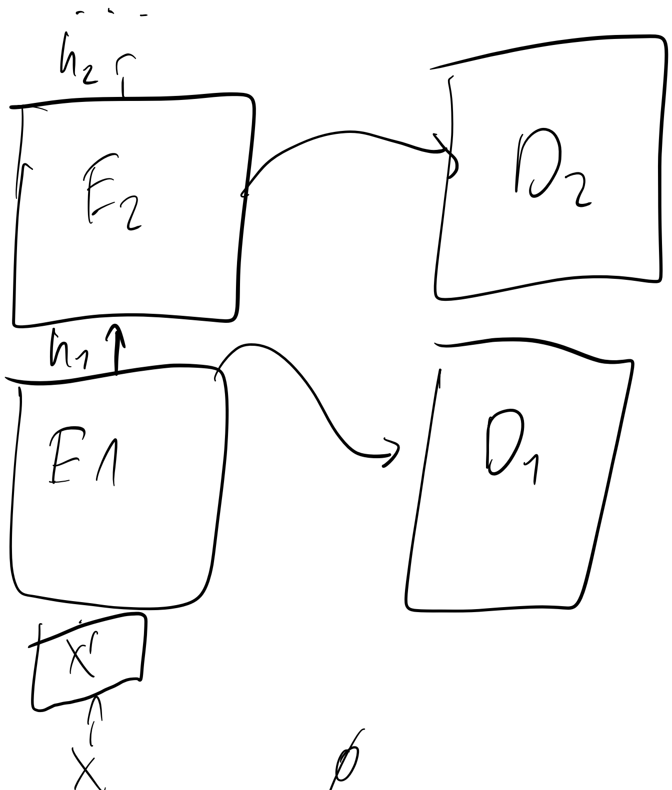
Similarily, the **Decoder** is also a stack of 6 blocks.
However in addition to the two sub-blocks of the encoder, it features a 3rd sub-block.

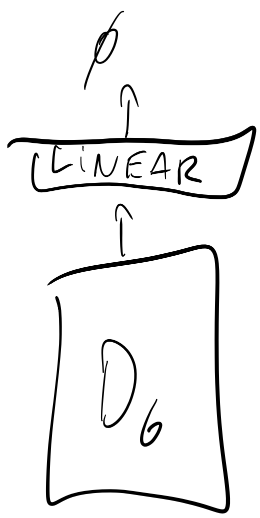
Transformer

Similarily, the **Decoder** is also a stack of 6 blocks.
However in addition to the two sub-blocks of the encoder, it features a 3rd sub-block.
This 3rd sub-block performs multi-head attention over the output of the encoder. This ”encoder-decoder attention” layer uses Q from the previous decoder layer, and K, V from the output of the encoder.

Similarily, the **Decoder** is also a stack of 6 blocks.
However in addition to the two sub-blocks of the encoder, it features a 3rd sub-block.
This 3rd sub-block performs multi-head attention over the output of the encoder. This "encoder-decoder attention" layer uses Q from the previous decoder layer, and K, V from the output of the encoder.




h2
E2
D2
h1
E1
D1
X'
X

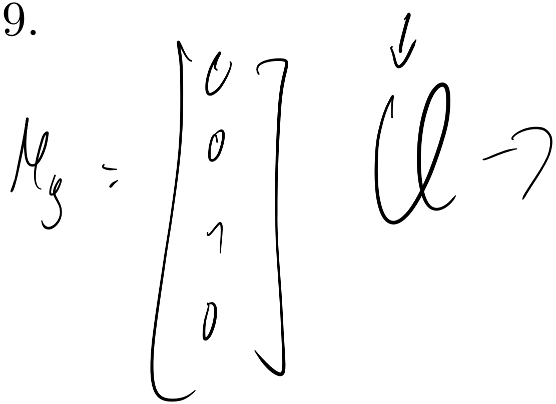

ϕ
LINEAR
D6

Transformer

What about inputs?

Transformer

What about inputs? The input embedding is a learneable "static" **token** embedding similar to the Word2Vec model we have seen in the lecture 9.



pos 1 2 3 4

$x_2 \quad x_1 \quad x_3 \quad x_4$

$q_2 \quad q_1 \quad q_3 \quad q_4$

$k_2 \quad k_1 \quad k_3 \quad k_4$

$v_2 \quad v_1 \quad v_3 \quad v_4$

[ ]

$$\phi\left(\frac{q^T k}{\sqrt{d}}\right) v$$

$$\begin{bmatrix}1\\0\\0\\0\end{bmatrix} \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \cdots \quad W_2$$

$$x' = x \cdot W + pos_x W_2$$

Transformer

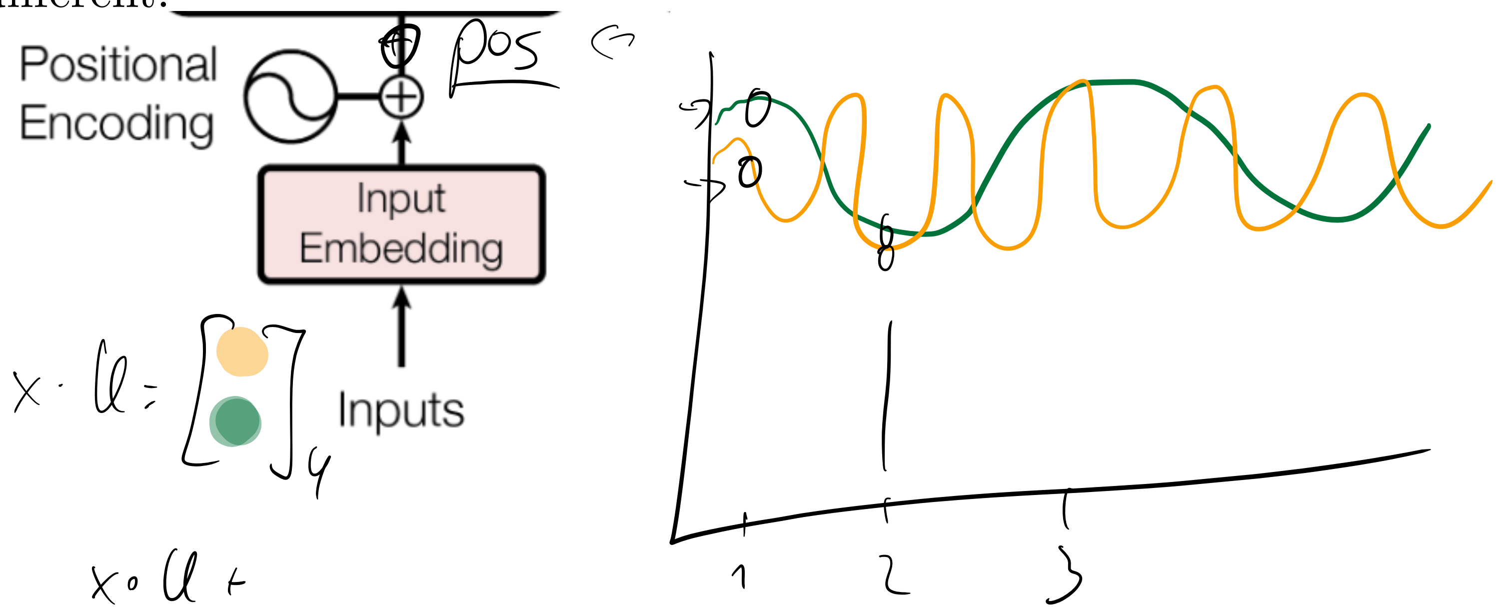
What about inputs? The input embedding is a learneable "static" **token** embedding similar to the Word2Vec model we have seen in the lecture 9.
What is "Positional Encoding?"

Transformer

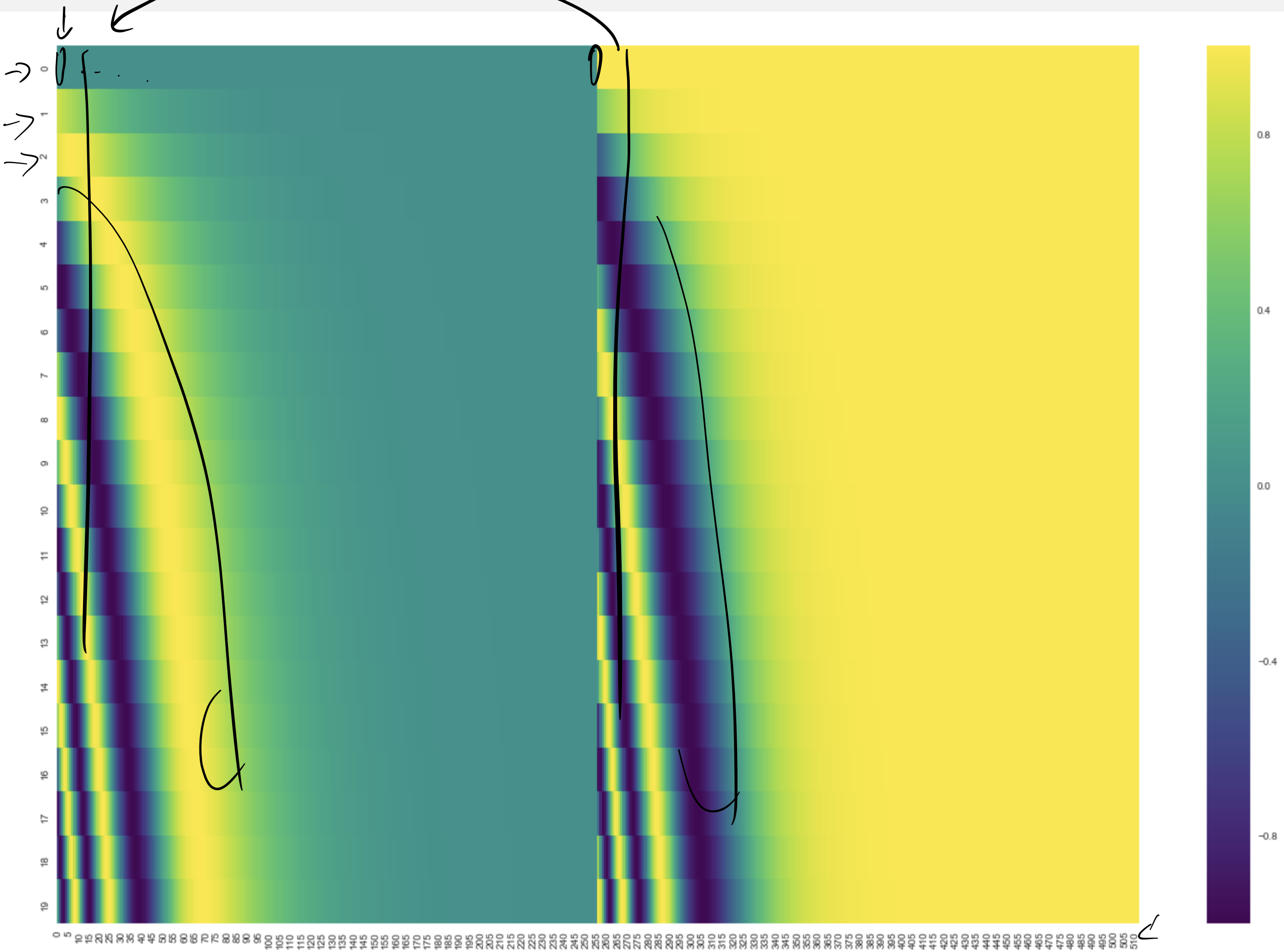
What about inputs? The input embedding is a learneable "static" **token** embedding similar to the Word2Vec model we have seen in the lecture 9.
What is "Positional Encoding?" It's either a learneable (representing position in a sequence) embedding, or a predefined embedding of different.

Transformer

What about inputs? The input embedding is a learneable "static" **token** embedding similar to the Word2Vec model we have seen in the lecture 9.
What is "Positional Encoding?" It's either a learneable (representing position in a sequence) embedding, or a predefined embedding of different.

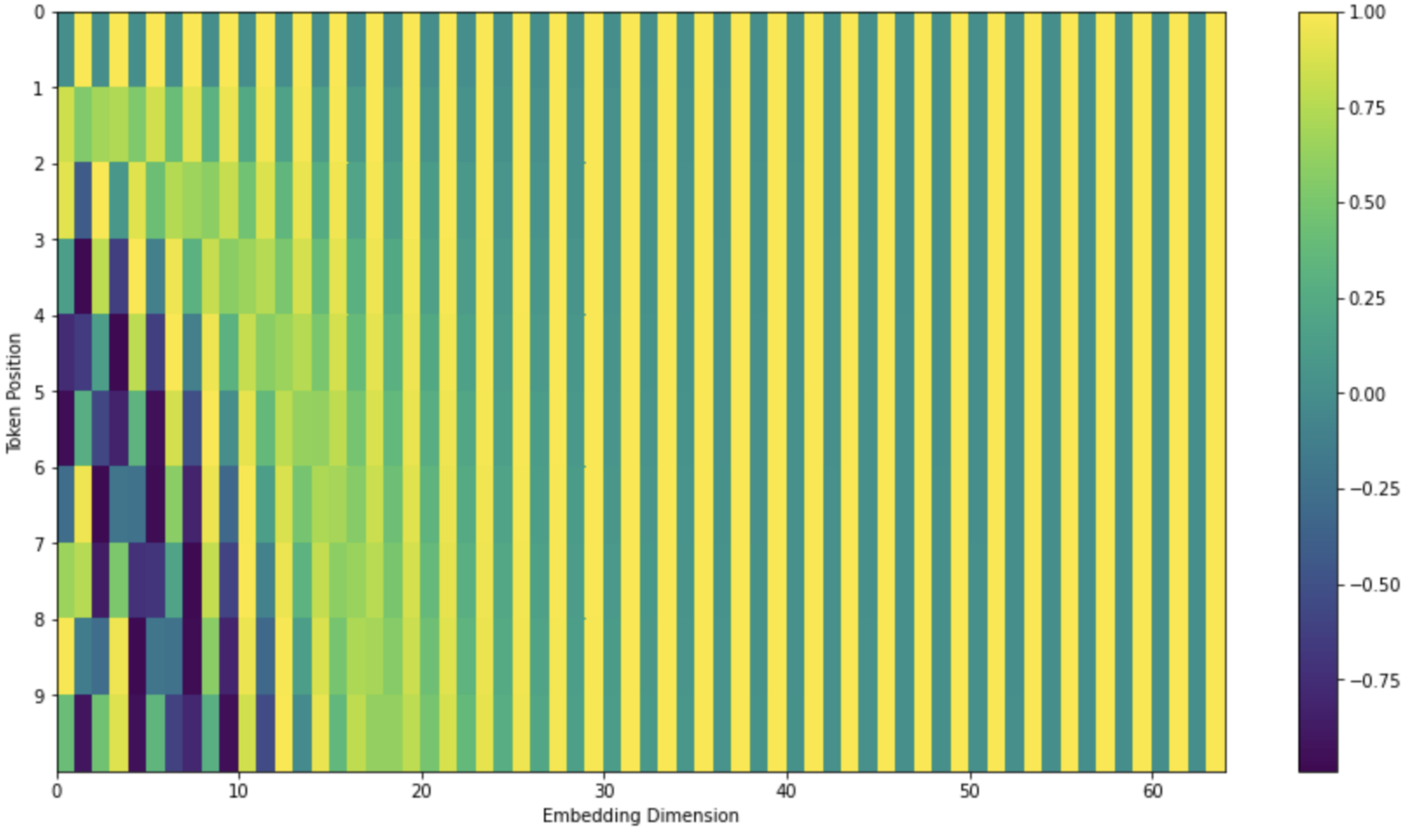


Positional Encoding



A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in

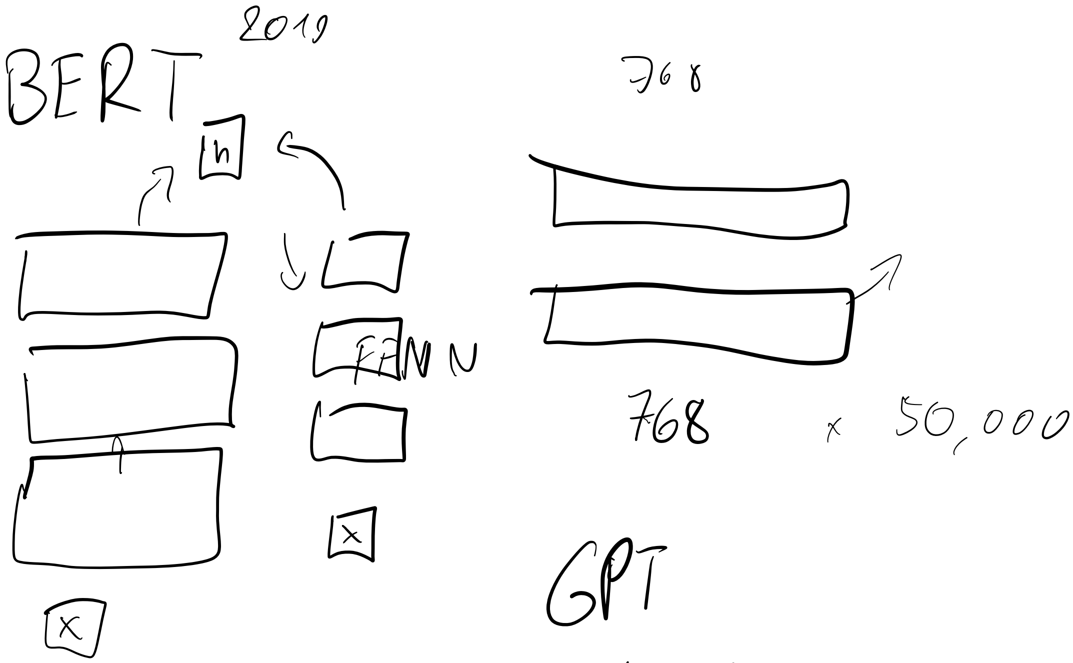
Positional Encoding



How to train a Transformer

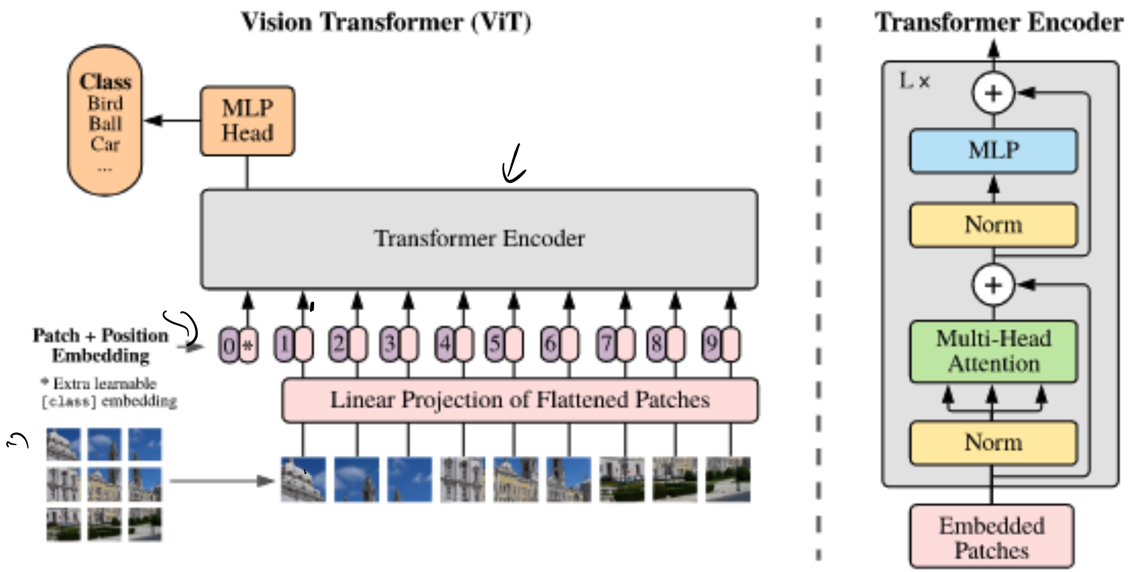
The original Transformer model was trained on an English-German translations, where at each step the final decoder state was fed into a simple Linear Layer followed by a softmax to produce probabilities over next tokens.

Currently there are a large number of pre-training tasks (similar in idea to W2V). One of the most common ones is **Masked Language Modelling**, where we randomly replace 15% of tokens with "[MASK]", and the goal of the model is to predict back the original token.

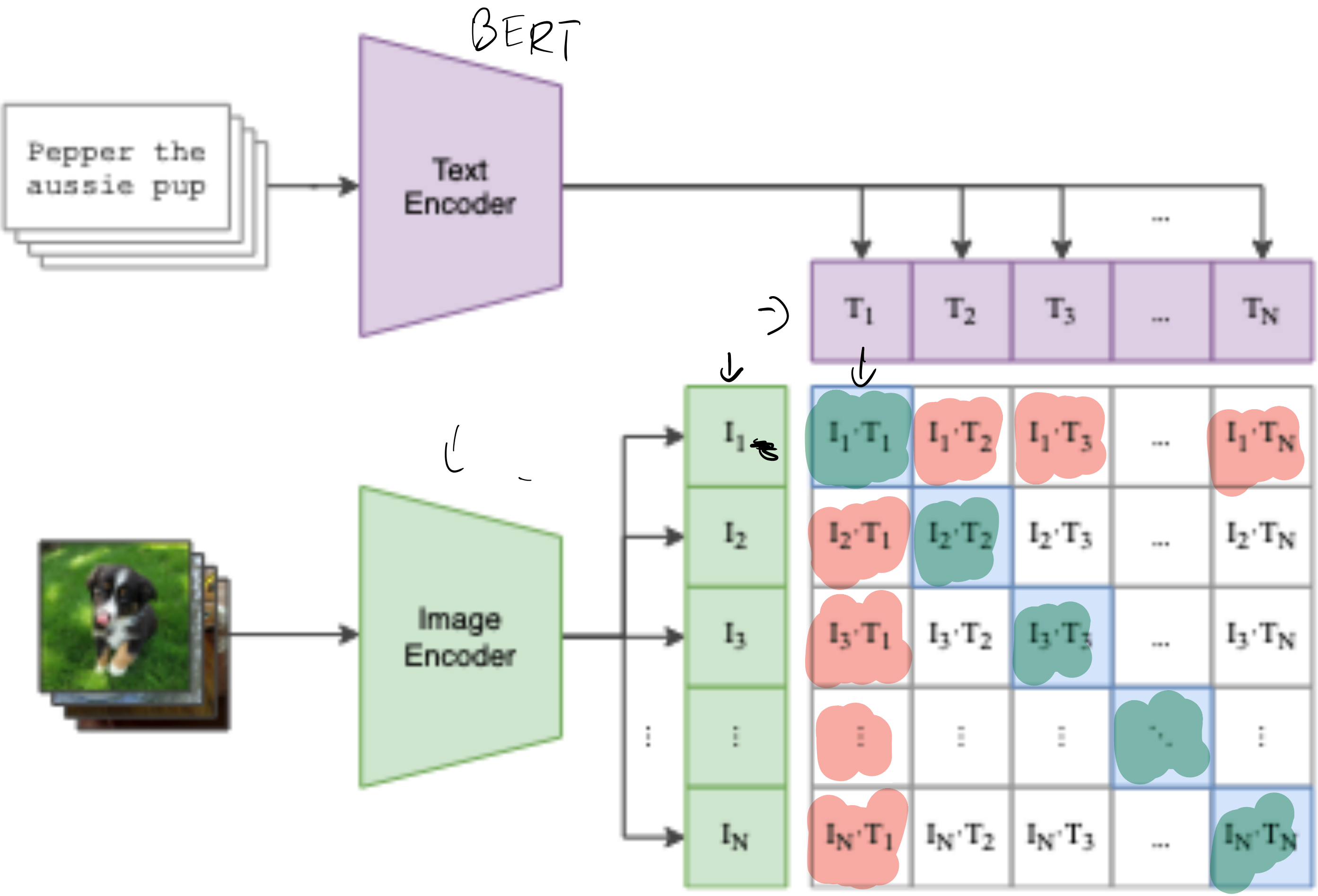


huggingface transformers

Vision Transformers



CLIP


BERT
Pepper the aussie pup
Text Encoder
Image Encoder
T_1 T_2 T_3 ... T_N
I_1 I_2 I_3 I_N
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
$I_2 \cdot T_1$ $I_2 \cdot T_2$ $I_2 \cdot T_3$ ... $I_2 \cdot T_N$
$I_3 \cdot T_1$ $I_3 \cdot T_2$ $I_3 \cdot T_3$ ... $I_3 \cdot T_N$
$I_N \cdot T_1$ $I_N \cdot T_2$ $I_N \cdot T_3$ ... $I_N \cdot T_N$

Neural Net Demo in Jax

Demo