UofT Computer Vision Club

Paper Breakdown Series

Attention Is All You Need

youtube.com/@uoftcv

github.com/ut-cvdg/paper-breakdown

Follow us on social media

UofT CV is sponsored by GitHub

Sign up for the GitHub Student Developer Pack:
https://education.github.com/benefits

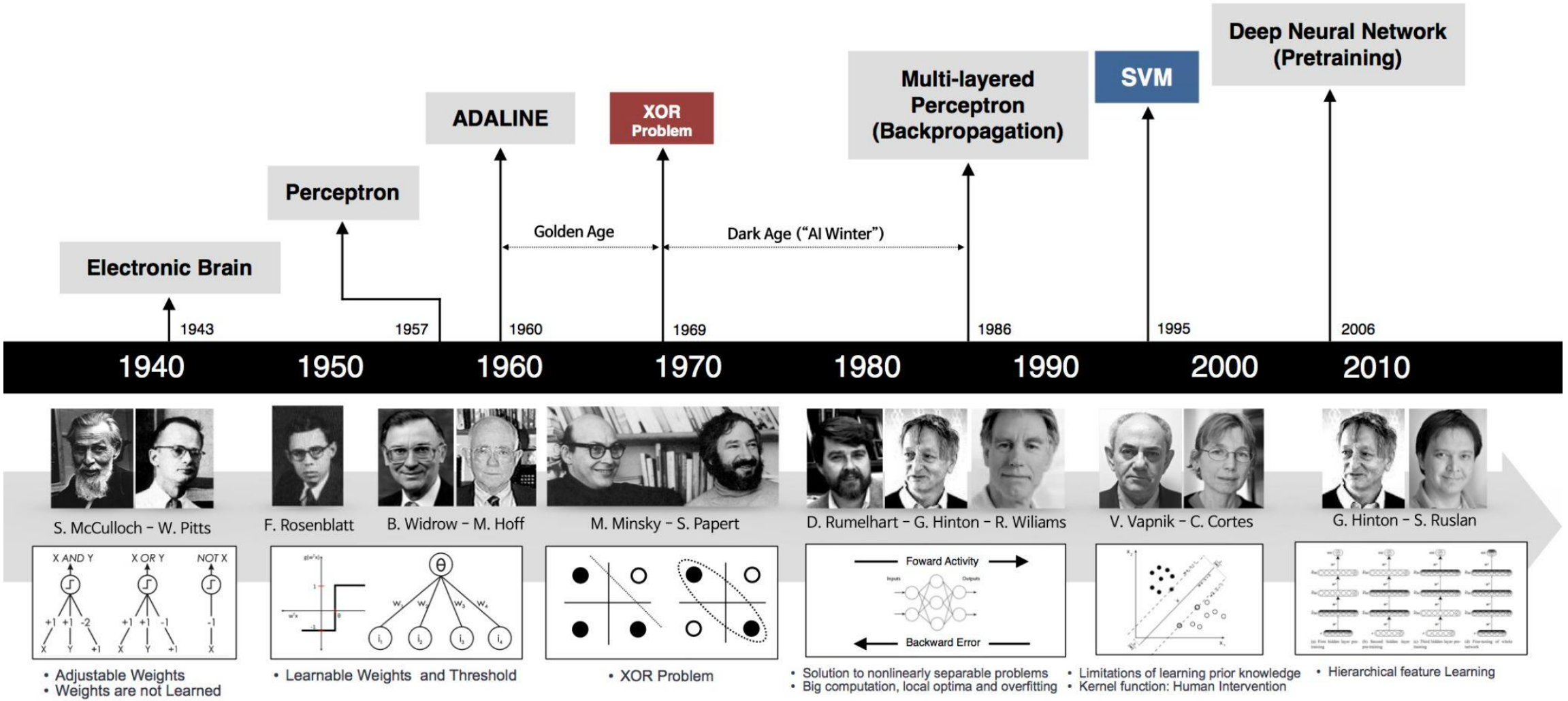
UNIVERSITY OF TORONTO | CVDG

About the Club

- We are a student club for Computer Vision and Machine Learning affiliated with the University of Toronto.
- We run:
  - paper “discussions” for paper in the past 1 week
  - conference-style research orals for pre-prints
  - Now, paper breakdown series

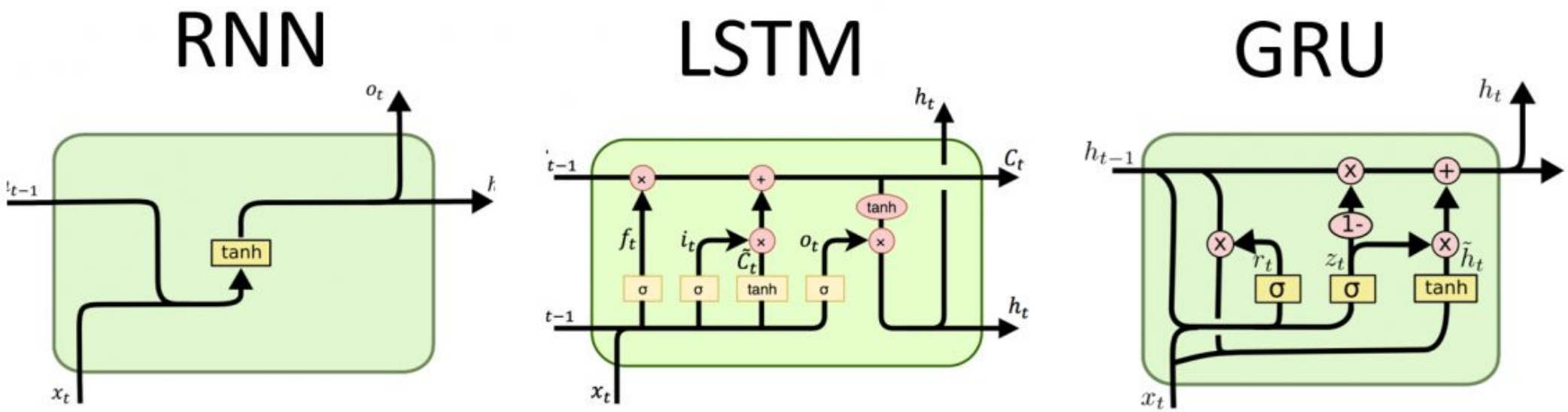
If you want to help run our club, talk to any of us or drop an email contact@cvdg.tech.

Timeline


Deep Neural Network (Pretraining)
SVM
Multi-layered Perceptron (Backpropagation)
XOR Problem
ADALINE
Perceptron
Electronic Brain
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch - W. Pitts
F. Rosenblatt
B. Widrow - M. Hoff
M. Minsky - S. Papert
D. Rumelhart - G. Hinton - R. Wiliams
V. Vapnik - C. Cortes
G. Hinton - S. Ruslan
X AND Y
X OR Y
NOT X
Foward Activity
Backward Error
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

Previous Models

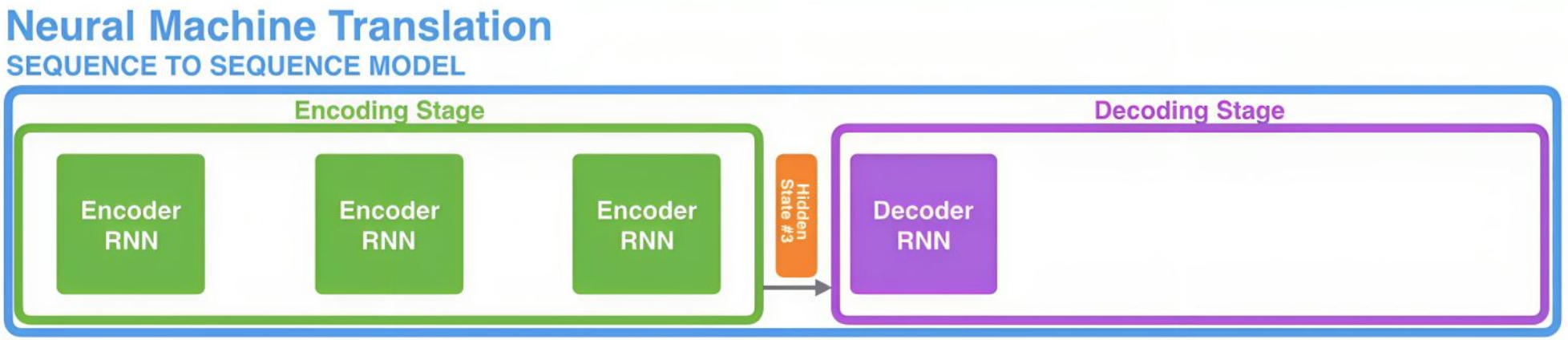
The main models for sequence-to-sequence models (machine translation)



Source: Jay Alammar's Blog

Limitations of Previous Models

- Sequential processing
- Fixed source representation
- Context vector bottleneck



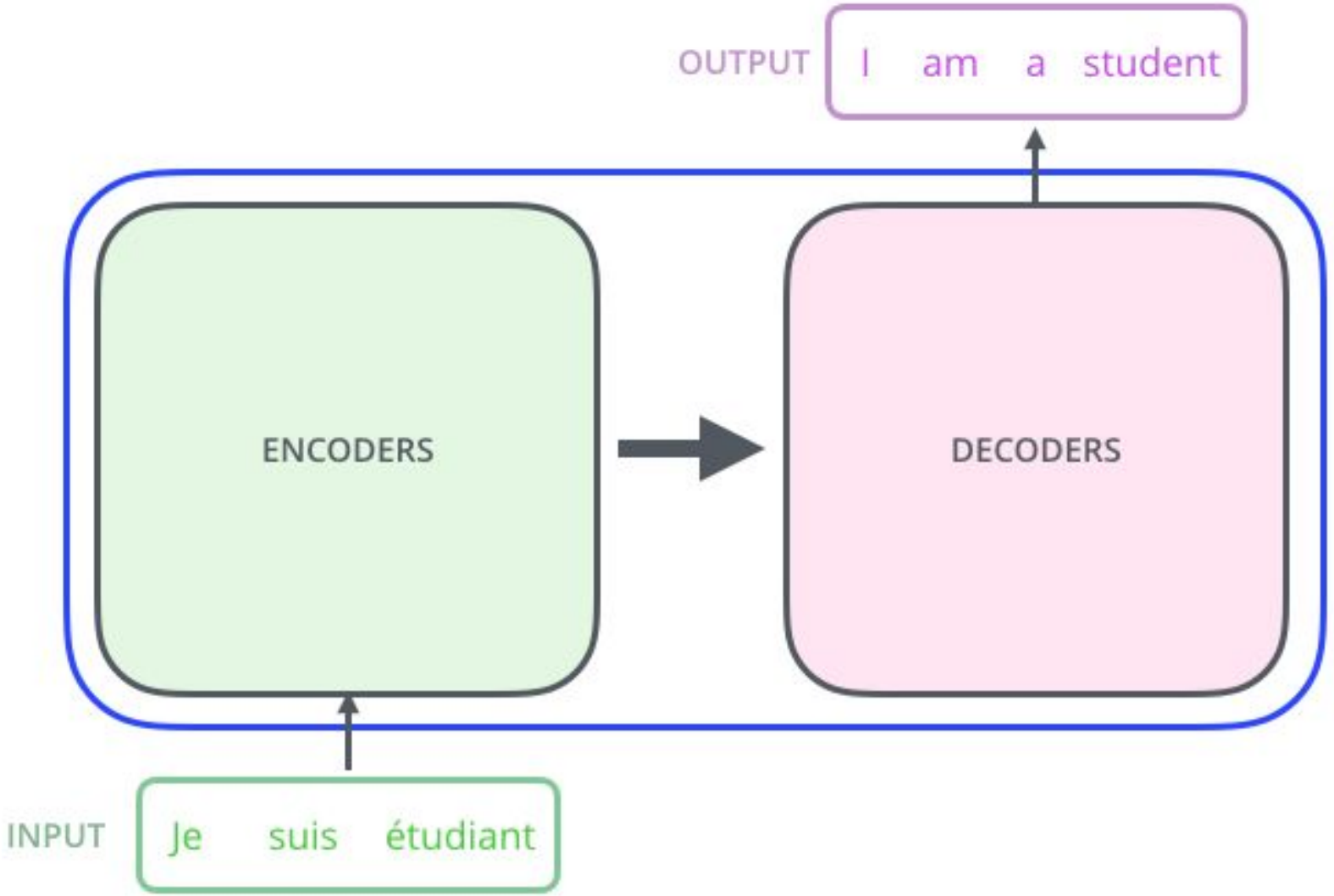
Source: Jay Alammar's Blog

Big Picture Walkthrough

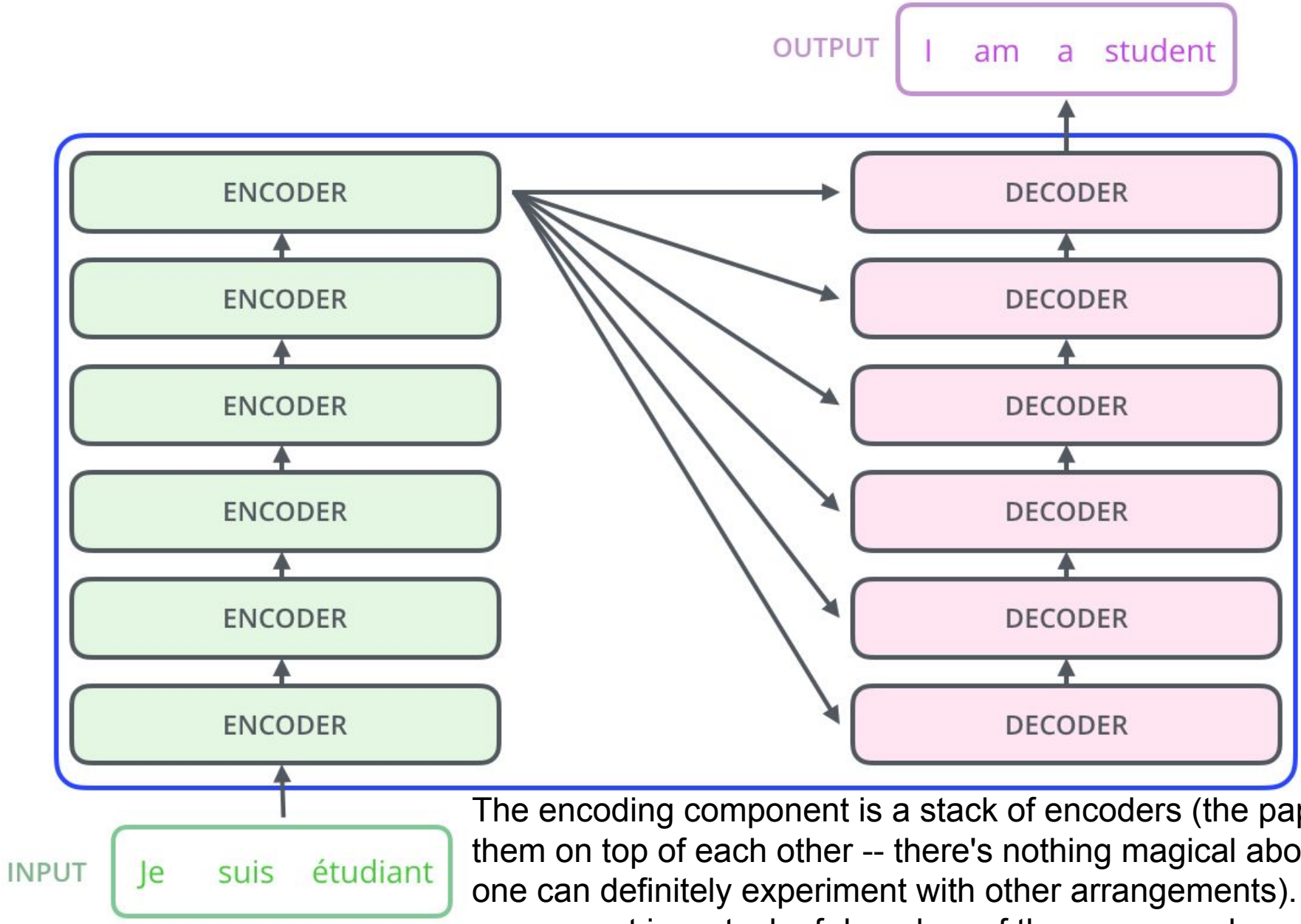
In a machine translation application, it would take a sentence in one language, and output its translation in another.

Source: Jay Alammar's Blog

Popping open that box, we see an encoding component, a decoding component, and connections between them.



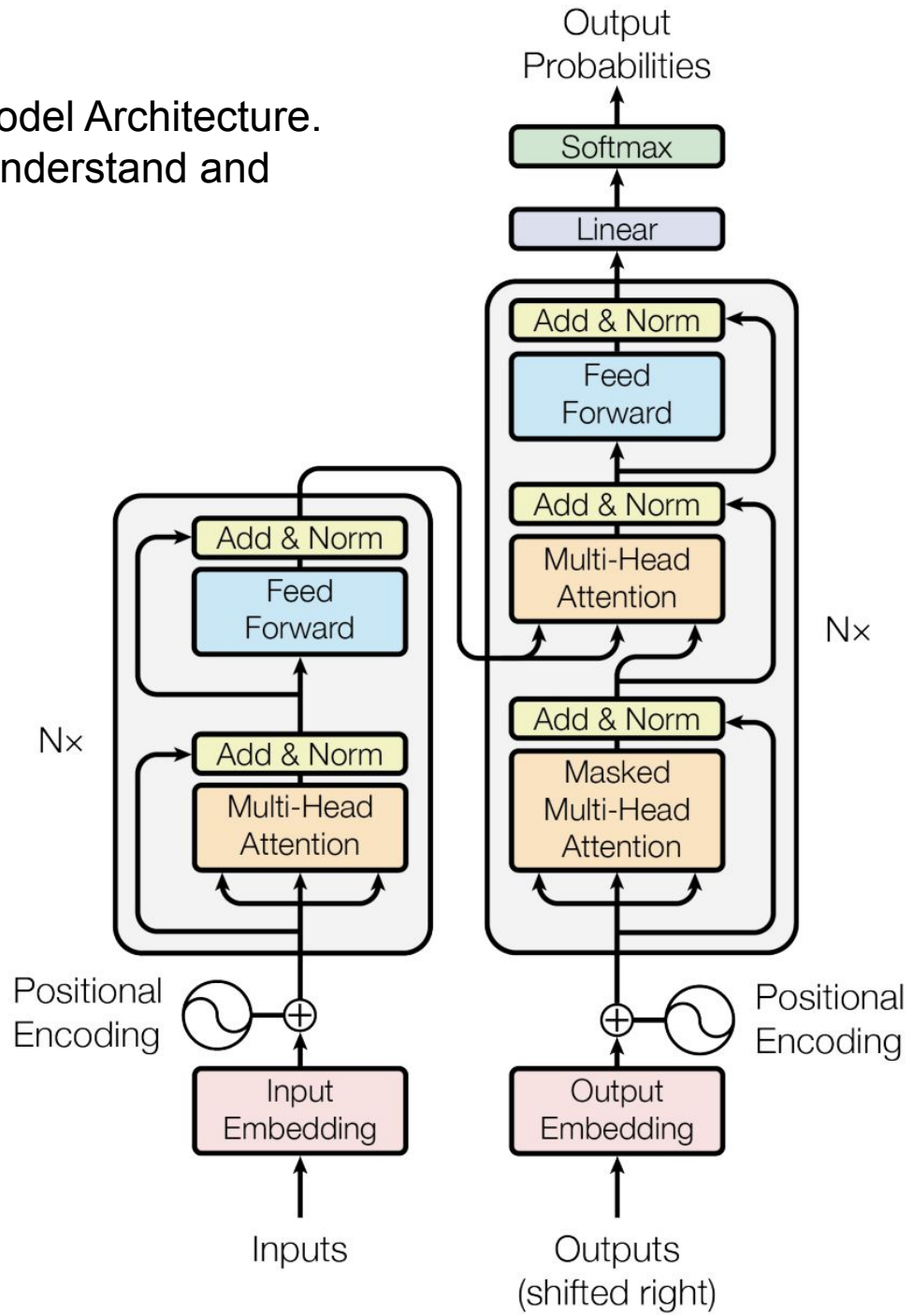
Source: Jay Alammar's Blog



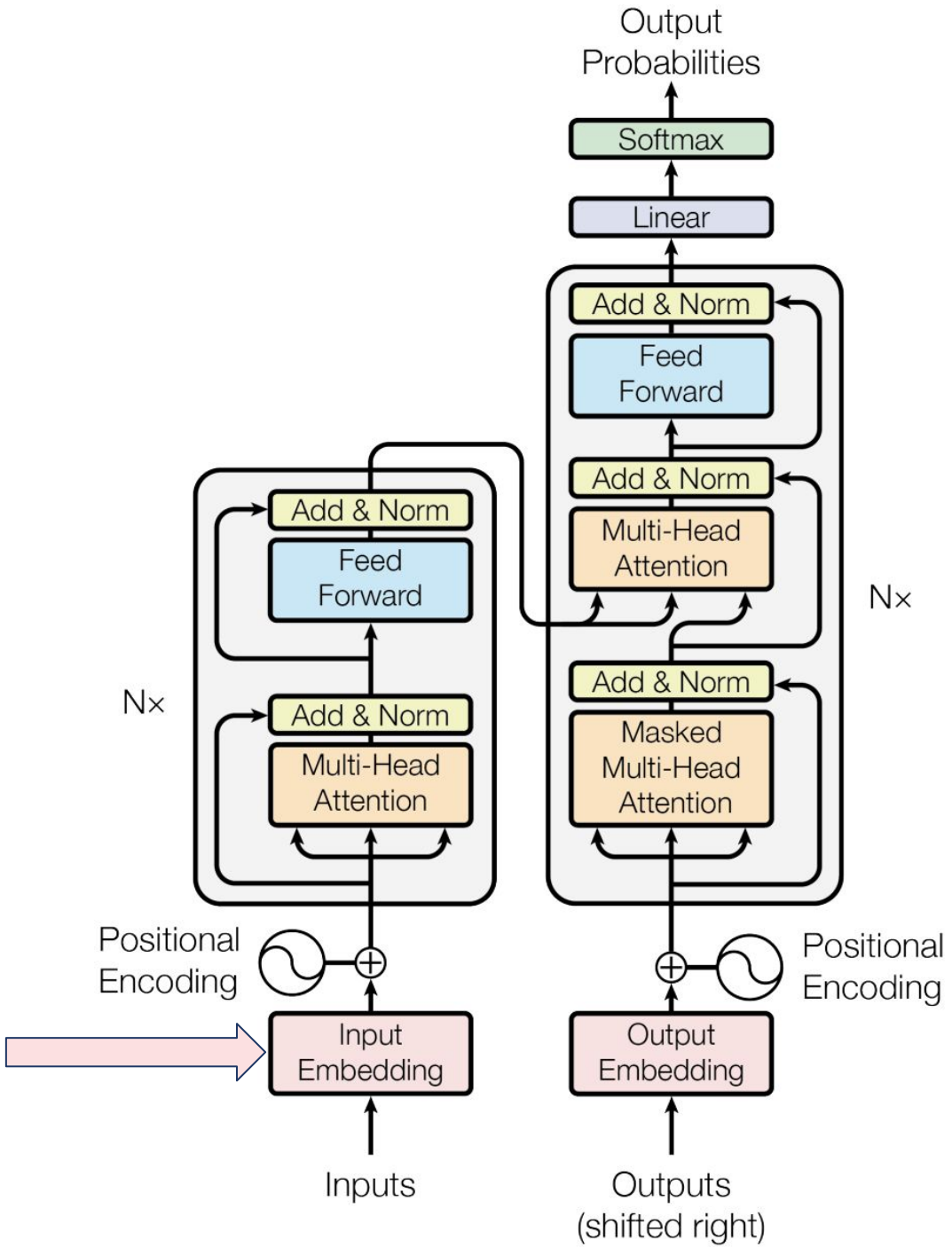
The encoding component is a stack of encoders (the paper stacks six of them on top of each other -- there's nothing magical about the number six, one can definitely experiment with other arrangements). The decoding component is a stack of decoders of the same number.

Source: Jay Alammar's Blog

Here is the complete Transformer model Architecture. By the end of this session, you will understand and know how to implement every block.



Source: Vaswani et al.



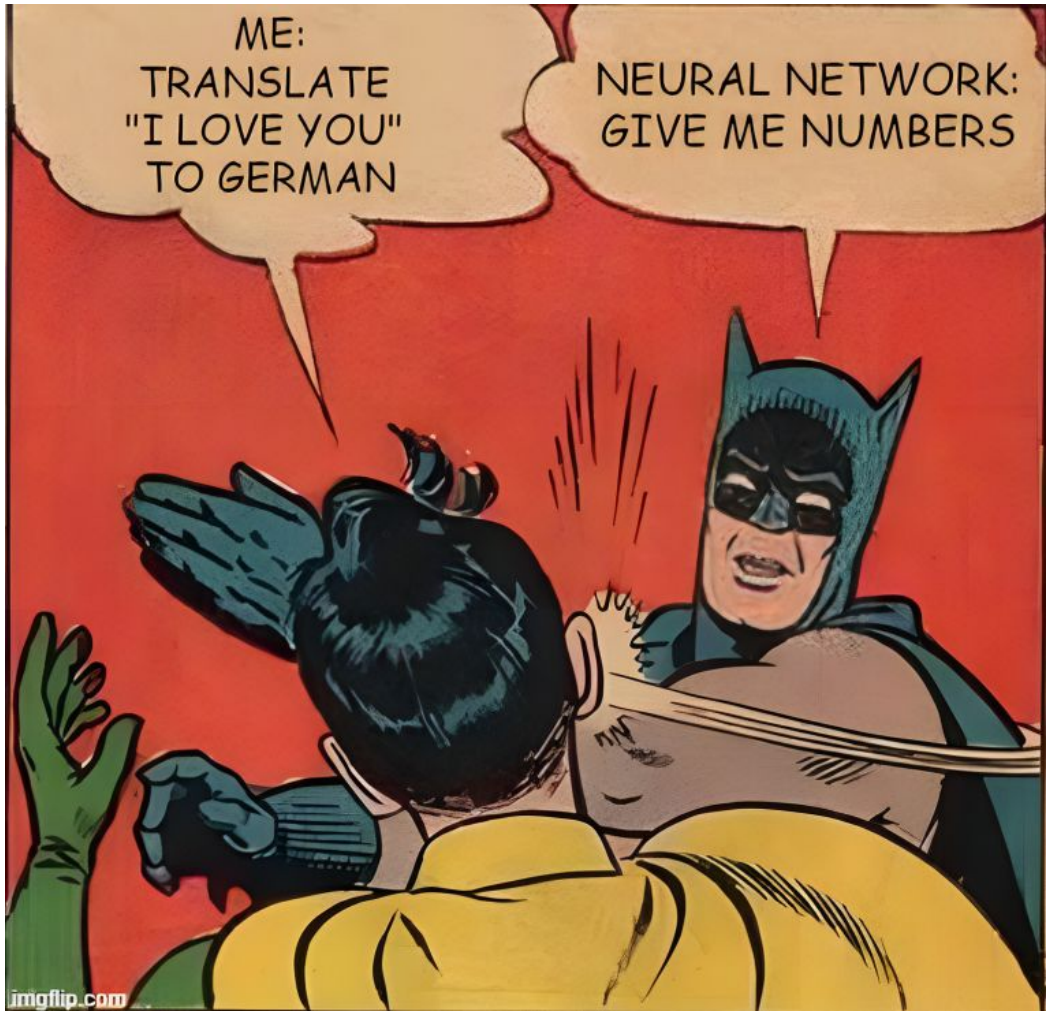
Source: Vaswani et al.

Data Preprocessing

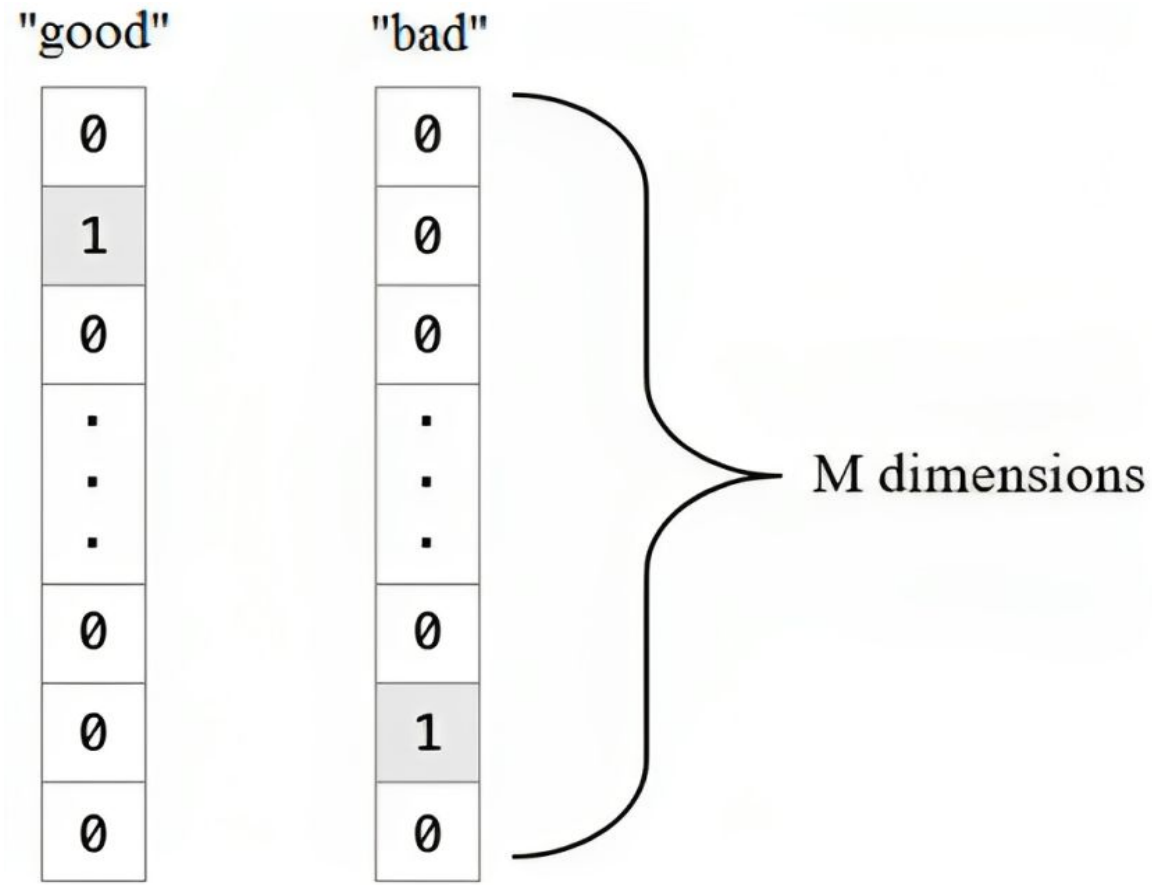
A bit about neural networks:

- Process lists of numbers
- Matrix multiplications on steroids

We usually use one vector per word.

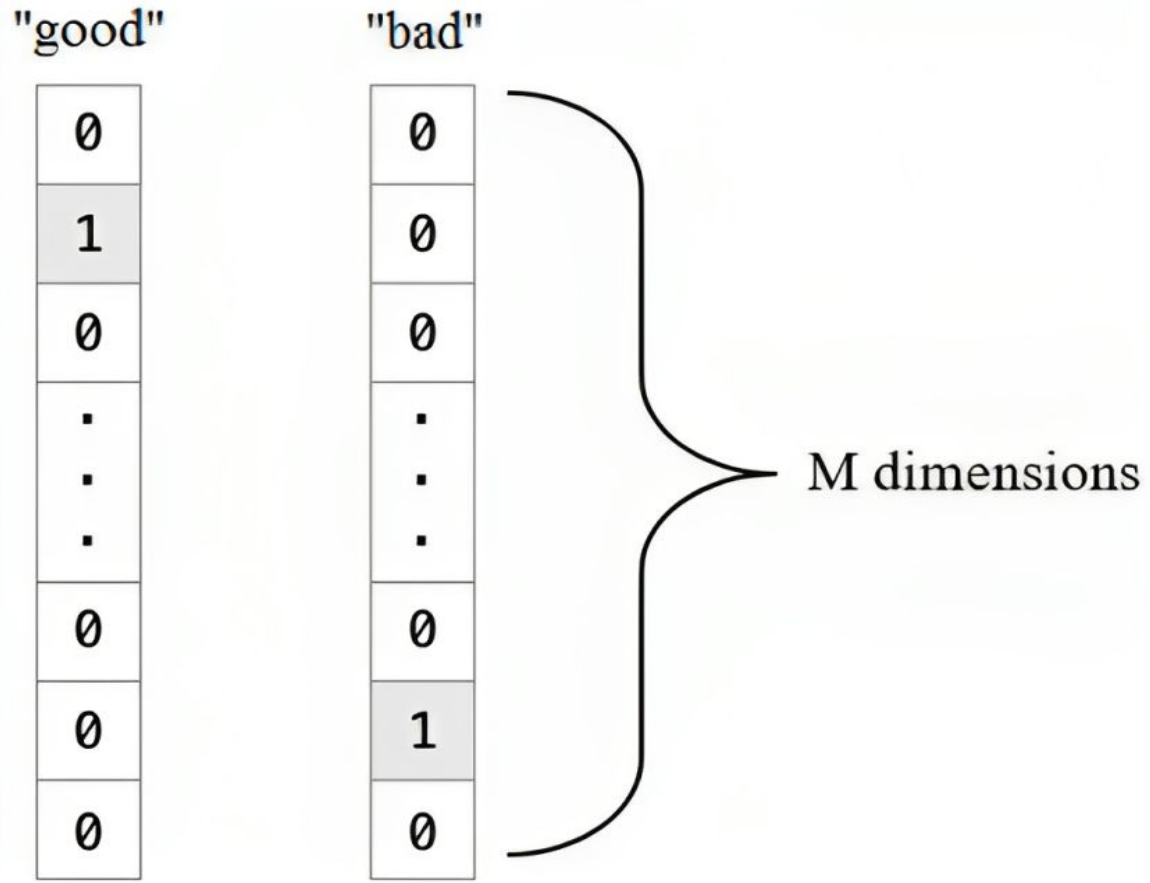


One-Hot Vectors

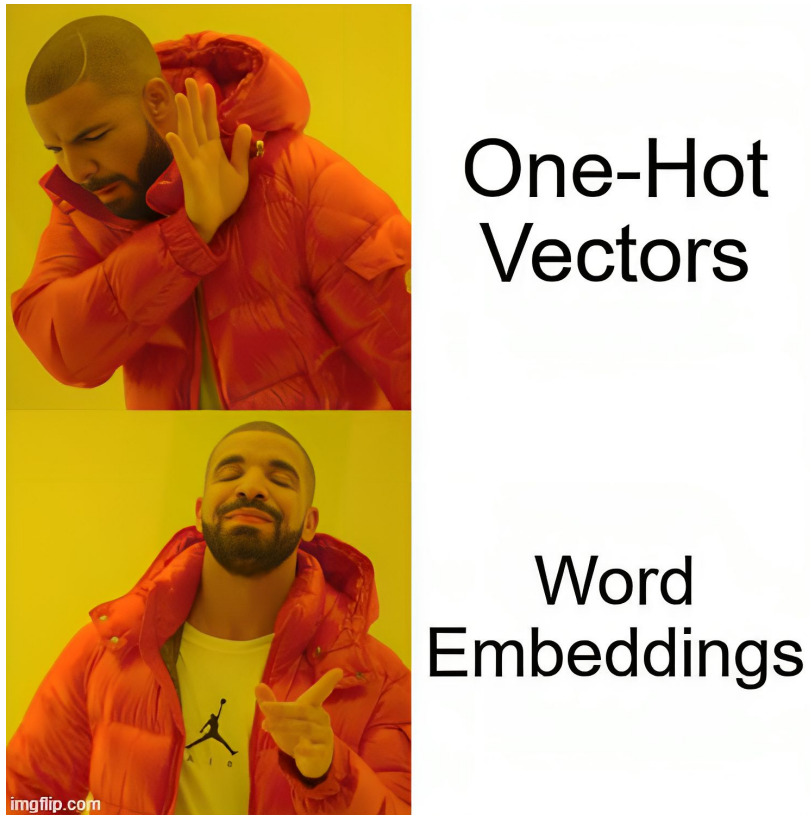


[Source: Vo, et al.]

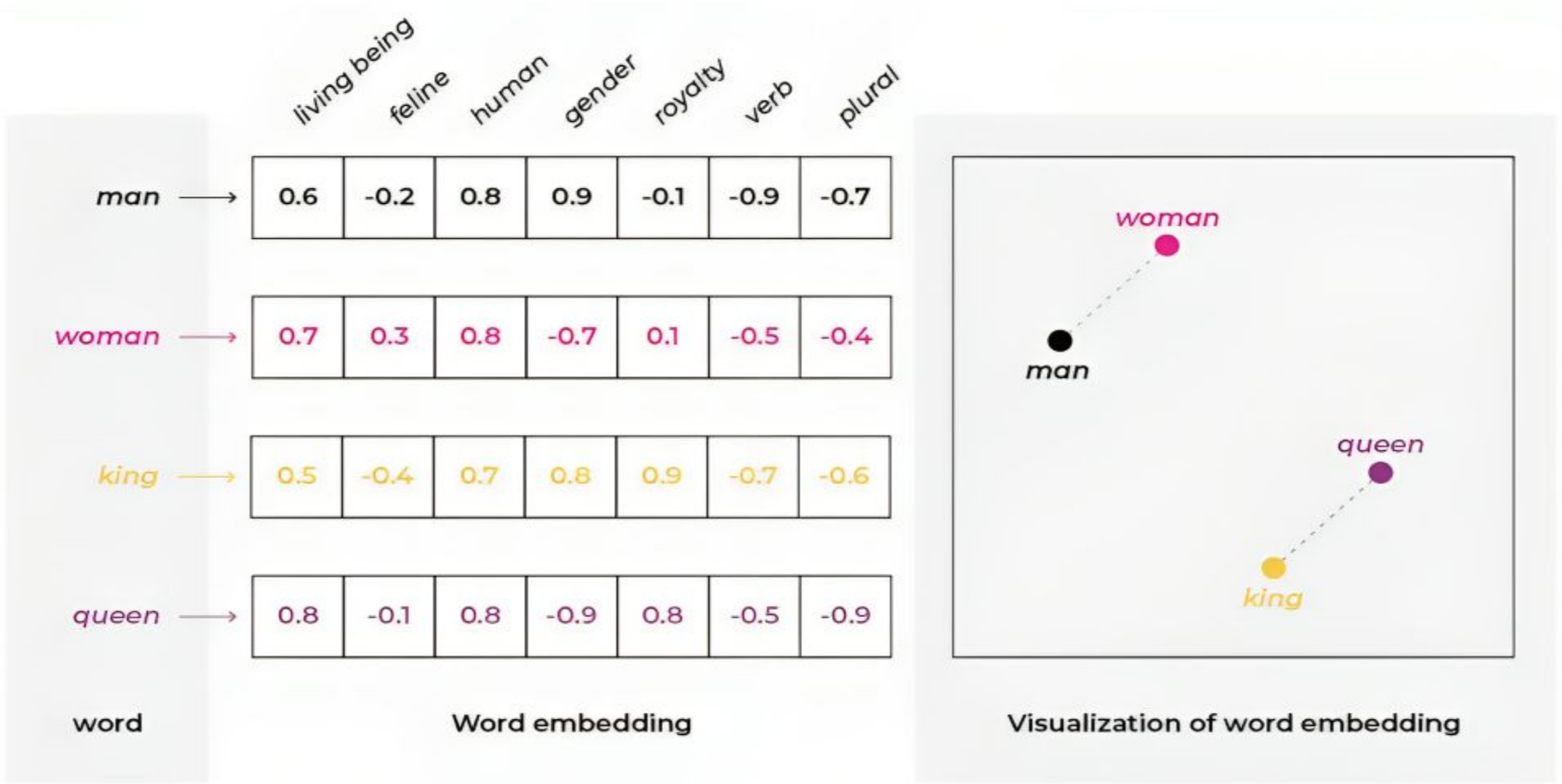
One-Hot Vectors



[Source: Vo, et al.]



Word Embeddings



[Source: Francisco Castillo]

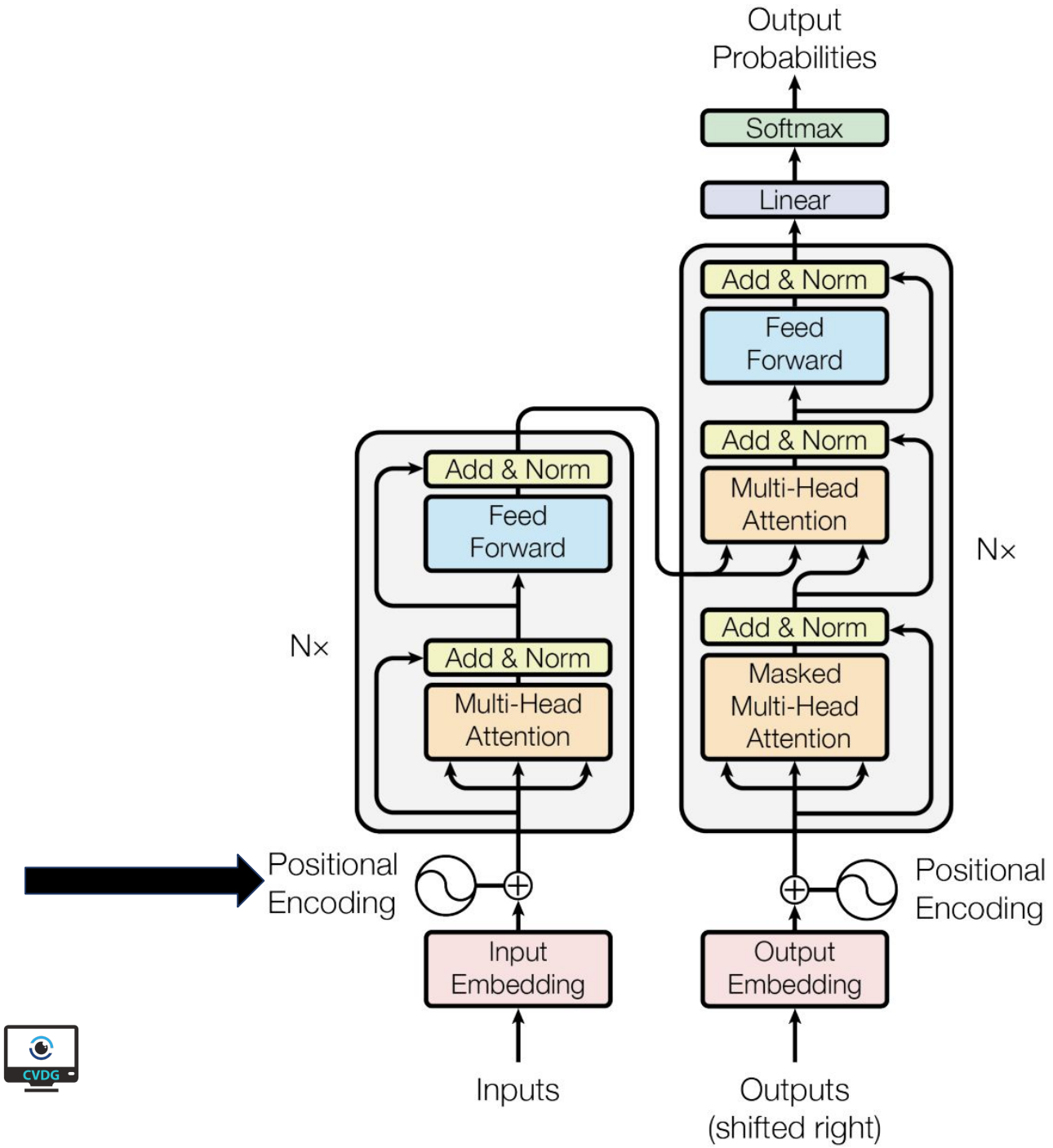
PyTorch

```python
class InputEmbedding(nn.Module):
    def __init__(self, d_model: int, vocab_size: int) -> None:
        super(InputEmbedding, self).__init__()
        # you can also do this:
        # super().__init__()
        self.d_model = d_model  # in this paper, it 512
        self.vocab_size = vocab_size
        self.embedding = nn.Embedding(vocab_size, d_model)

    def forward(self, x):
        return self.embedding(x) * math.sqrt(self.d_model)
        # check the last line on page 5:
        # "In the embedding layers, we multiply those weights by d
model."
```

TensorFlow

```python
def prepare_batch(pt, en):
    """
    Preprocess a batch of Portuguese and English sentences for training a machine translation model.

    Args:
        pt: A tensor of Portuguese sentences of shape (batch_size,) and dtype tf.string.
        en: A tensor of English sentences of shape (batch_size,) and dtype tf.string.

    Returns:
        A tuple of two tensors representing the input and output sequences for the model, and a tensor
of shape
        (batch_size, max_length) representing the ground truth output sequences. The input sequence
tensor has shape
        (batch_size, max_length) and dtype tf.int64, and the output sequence tensor has shape
(batch_size, max_length)
        and dtype tf.int64.
    """
    pt = tokenizers.pt.tokenize(pt)      # Output is ragged.
    pt = pt[:, :MAX_TOKENS]    # Trim to MAX_TOKENS.
    pt = pt.to_tensor()  # Convert to 0-padded dense Tensor

    en = tokenizers.en.tokenize(en)
    en = en[:, :(MAX_TOKENS+1)]
    en_inputs = en[:, :-1].to_tensor()  # Drop the [END] tokens
    en_labels = en[:, 1:].to_tensor()   # Drop the [START] tokens

    return (pt, en_inputs), en_labels
```

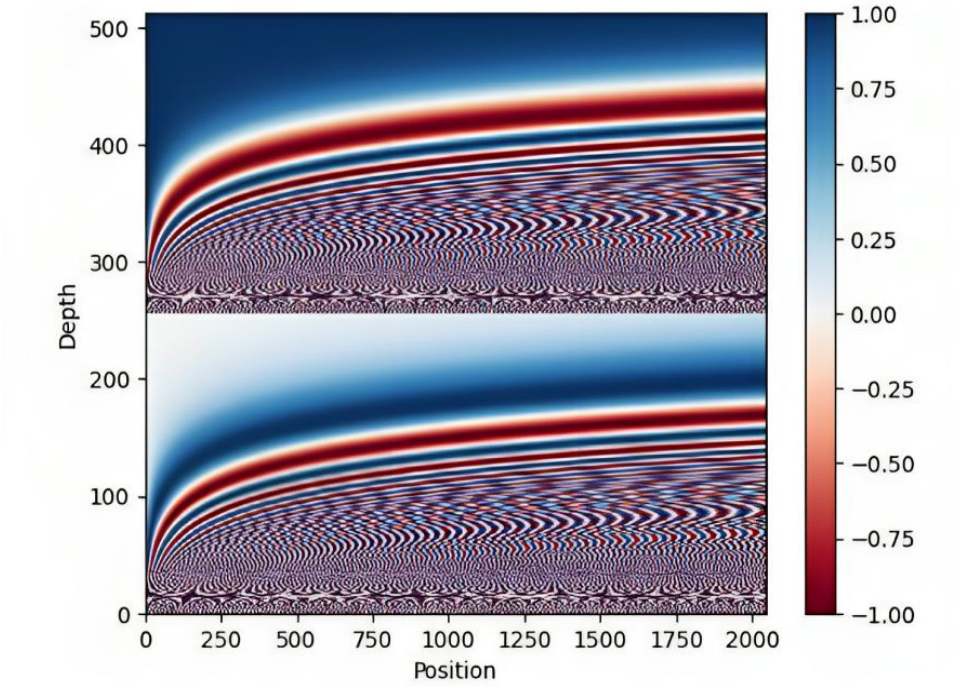
UNIVERSITY OF TORONTO | CVDG



Source: Vaswani et al.

Positional Encoding

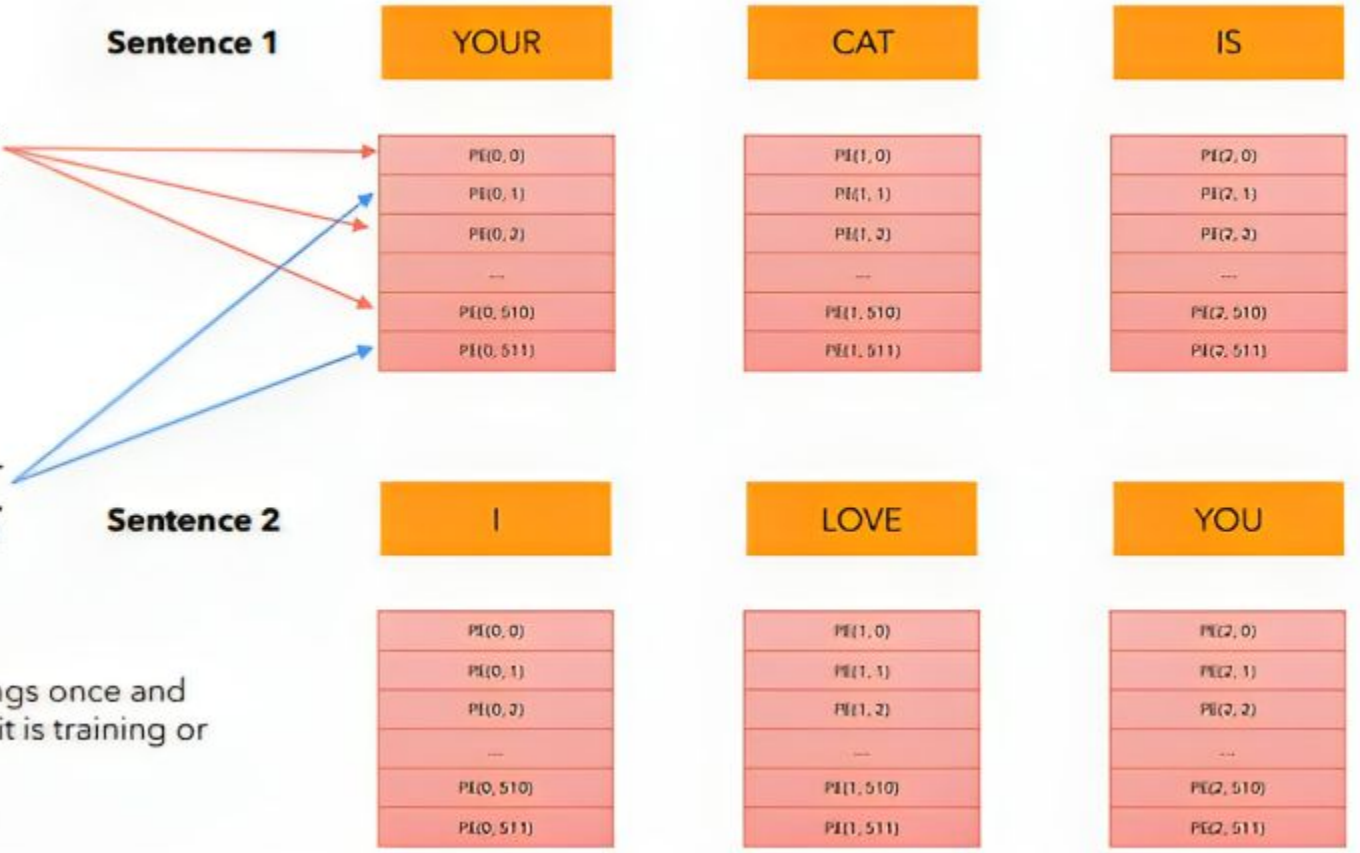
- We want each word to carry some information about its position in the sentence.
- Positional encoding represents a pattern that can be learned by the model.



[Source: Umar Jamil]

$$PE(pos, 2i) = \sin \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$

$$PE(pos, 2i+1) = \cos \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$



We only need to compute the positional encodings once and then reuse them for every sentence, no matter if it is training or inference.

[Source: Umar Jamil]

PyTorch

```python
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, seq_len: int, dropout: float) -> None:
        super(PositionalEncoding, self).__init__()
        self.d_model = d_model  # in this paper, it 512
        self.seq_len = seq_len  # maximum length of the sequence
        self.dropout = nn.Dropout(p=dropout)
        # create a matrix of shape (seq_len, d_model)
        # pe stands for positional encoding
        pe = torch.zeros(seq_len, d_model)
        # create a vector of shape (seq_len, 1)
        position = torch.arange(0, seq_len, dtype=torch.float32).unsqueeze(1)
        # now, we will create the denominator of the positional encoding formulae
        # since it is a bit long, we will break it into a few lines
        # first, we need a vector containing multiples of 2 from 0 to d_model (here, 512)
        # this line is because of the 2i term which is the power of 10000
        # thus, this vector provides for the numbers we need for 2i
        vector = torch.arange(0, d_model, 2, dtype=torch.float32)
```

PyTorch

```
        # now, we raise 10,000 to the power of 2i/d_model
        denominator_original = torch.pow(10000, vector/d_model)
        # this is the one used by Harvard Transformer article
        denominator_harvard = torch.exp(vector * (-math.log(10000.0)/d_model))
        # we apply sin for even dimension and cos for odd dimenion
        # apply sin and store it in even indices of pe
        pe[:, 0::2] = torch.sin(position * denominator_original)
        # apply cos and store it in odd indices of pe
        pe[:, 1::2] = torch.cos(position * denominator_original)
        # we need to add the batch dimension so that we can apply it to
        # batches of sentences
        pe = pe.unsqueeze(0)  # new shape: (1, seq_len, d_model)
        # register the pe tensor as a buffer so that it can be saved along with the
        # state of the model
        self.register_buffer("pe", pe)

    def forward(self, x):
        # we don't want to train the positional encoding, ie, we don't want to make it
        # a learnable parameter, so we set its requires_grad to False
        x = x + self.pe[:, :x.size(1)].requires_grad_(False)  # (batch, seq_len, d_model)
        return self.dropout(x)
```

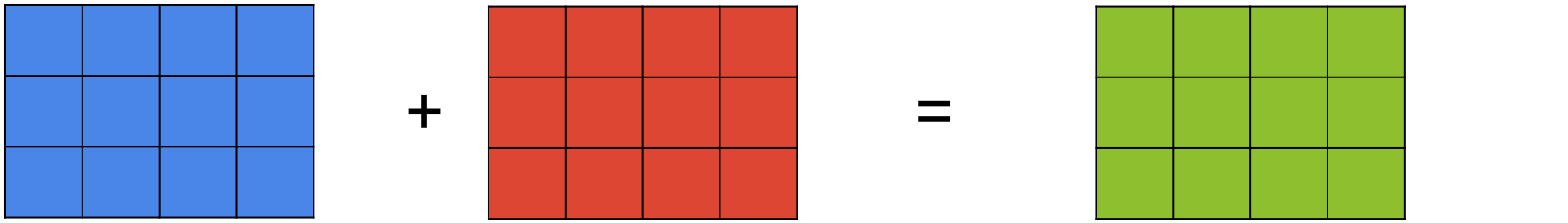
TensorFlow

```python
def positional_encoding(length, depth):
  """
  Generates a matrix of position encodings for an input sequence.

  Args:
      length: An integer representing the length of the input sequence.
      depth: An integer representing the dimensionality of the encoding.

  Returns:
      A `tf.Tensor` of shape `(length, depth)` representing the position encoding matrix.
  """
  depth = depth/2

  positions = np.arange(length)[:, np.newaxis]     # (seq, 1)
  depths = np.arange(depth)[np.newaxis, :]/depth   # (1, depth)

  angle_rates = 1 / (10000**depths)         # (1, depth)
  angle_rads = positions * angle_rates      # (pos, depth)

  pos_encoding = np.concatenate(
      [np.sin(angle_rads), np.cos(angle_rads)],
      axis=-1)

  return tf.cast(pos_encoding, dtype=tf.float32)
```

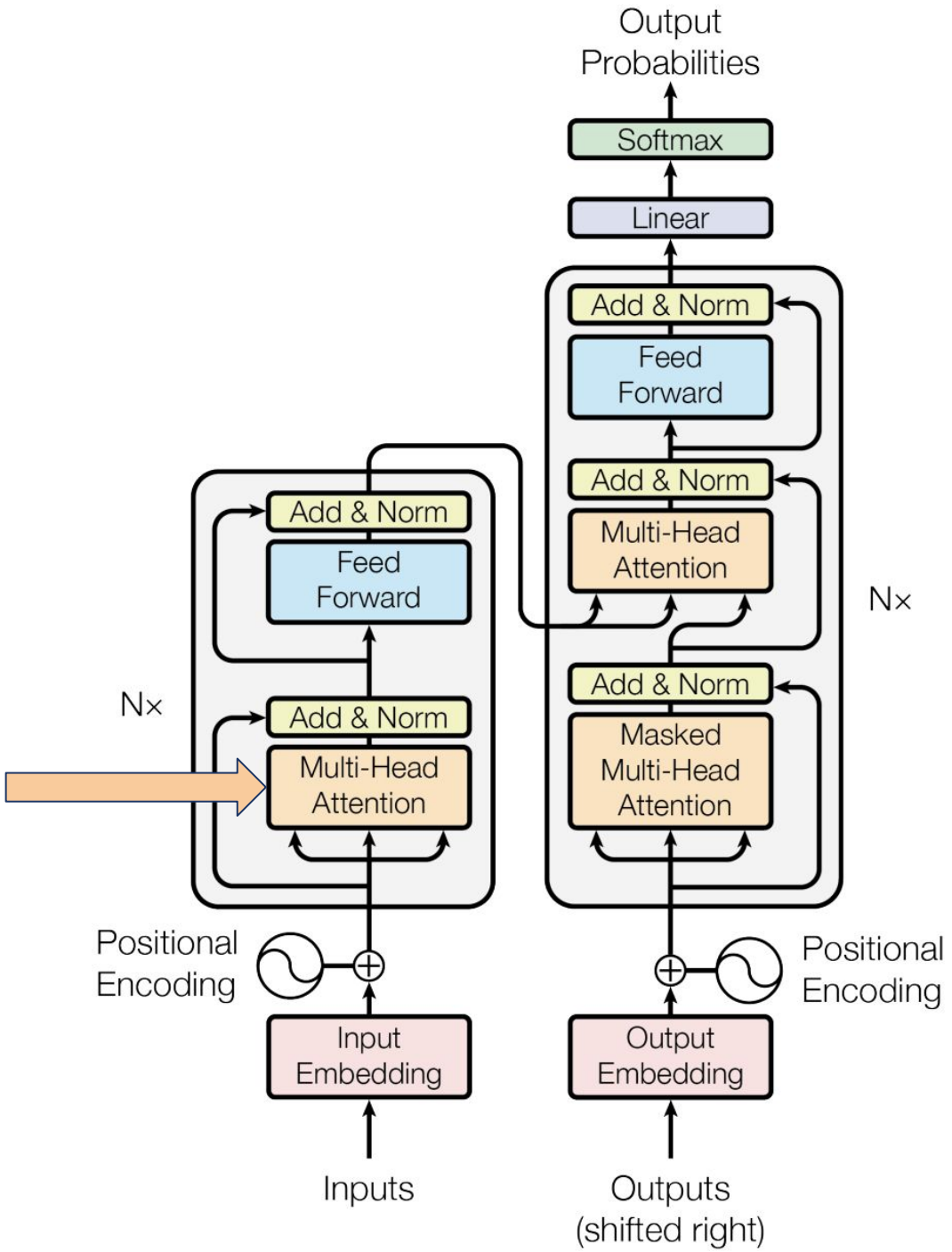
UNIVERSITY OF TORONTO

CVDG

Embedding + Positional Encoding

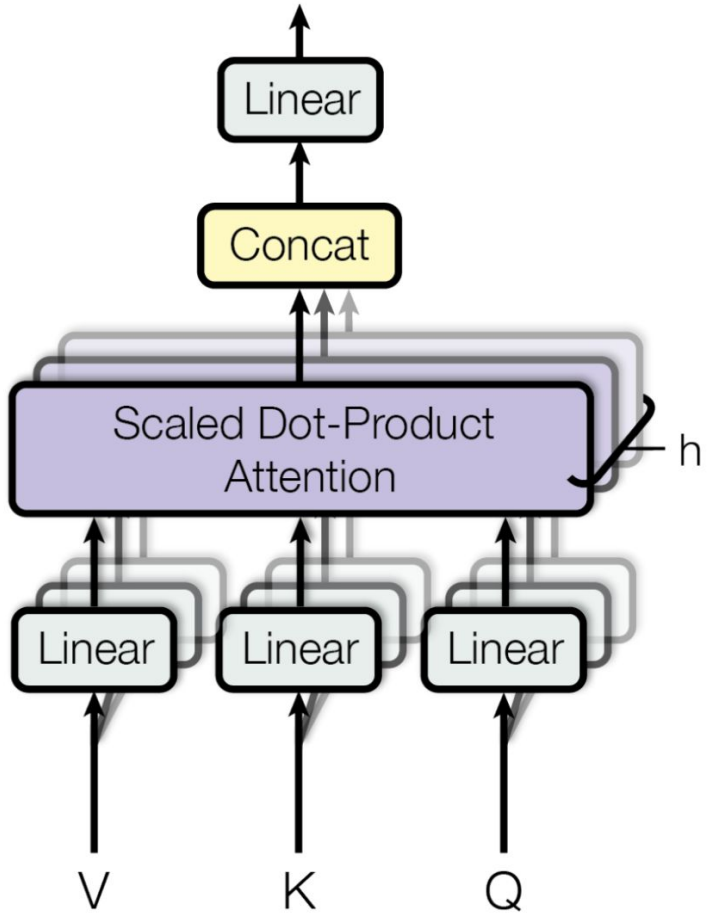


Input Embedding Positional Encoding Position-Aware Embedding


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Multi-Head Attention

=



Source: Vaswani et al.

First, What is Self-Attention?

Attending to the most important parts of the input.

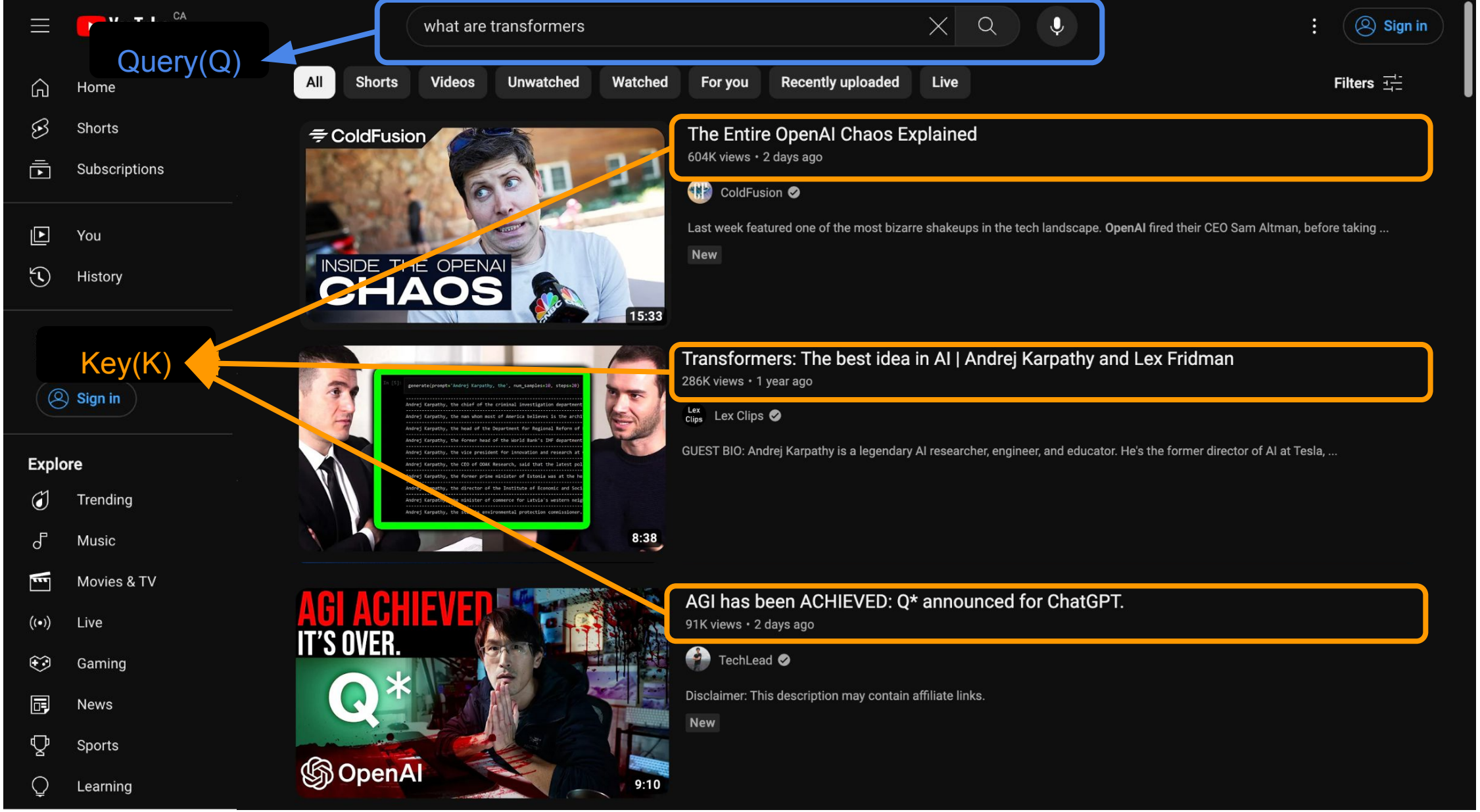
Source: The Brave of Sun Fighbird - Hajime Yatate

First, What is Self-Attention?

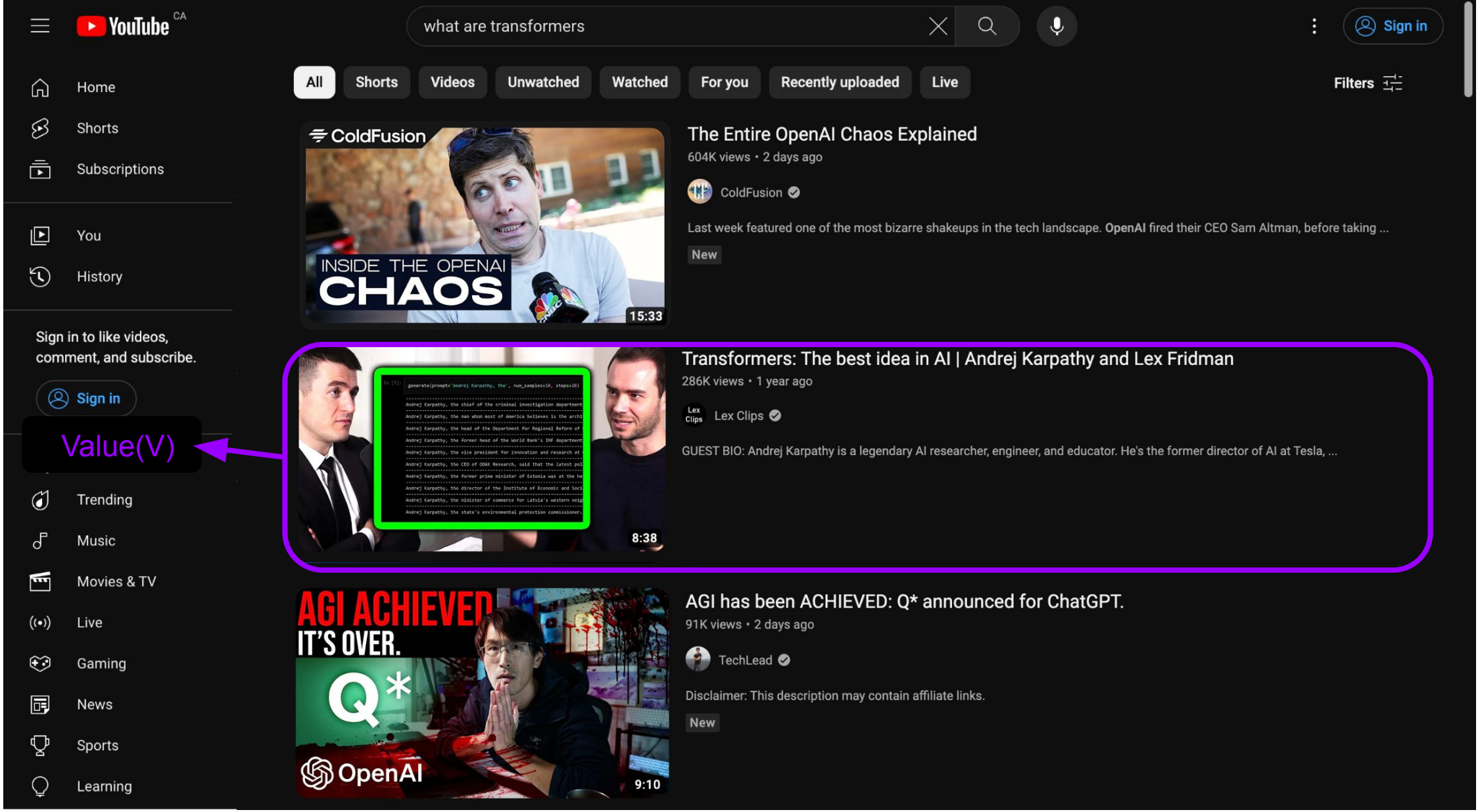
Attending to the most important parts of the input.

Analogous to a search problem!

Source: The Brave of Sun Fighbird - Hajime Yatate



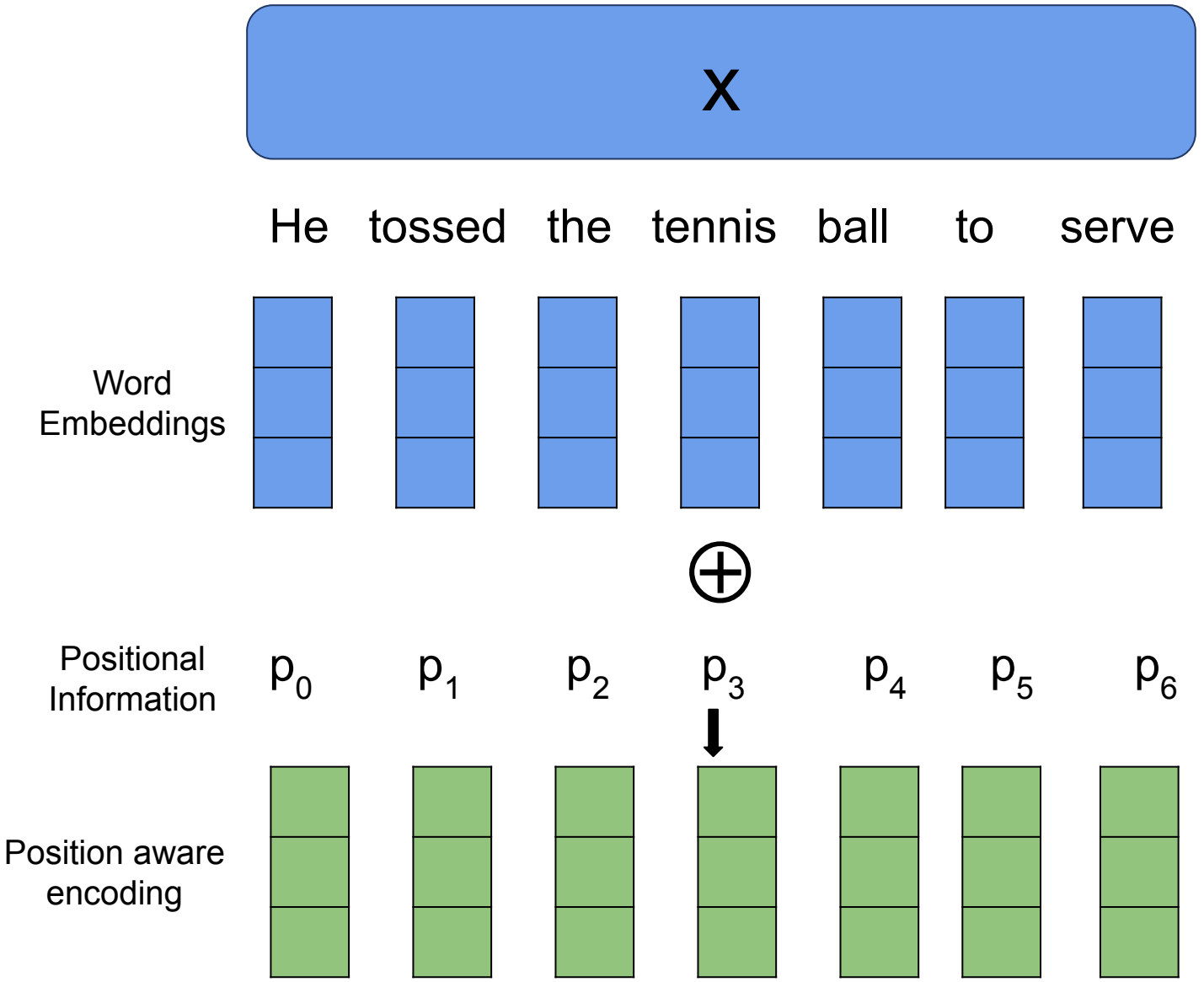
Source: youtube.com



Source: youtube.com

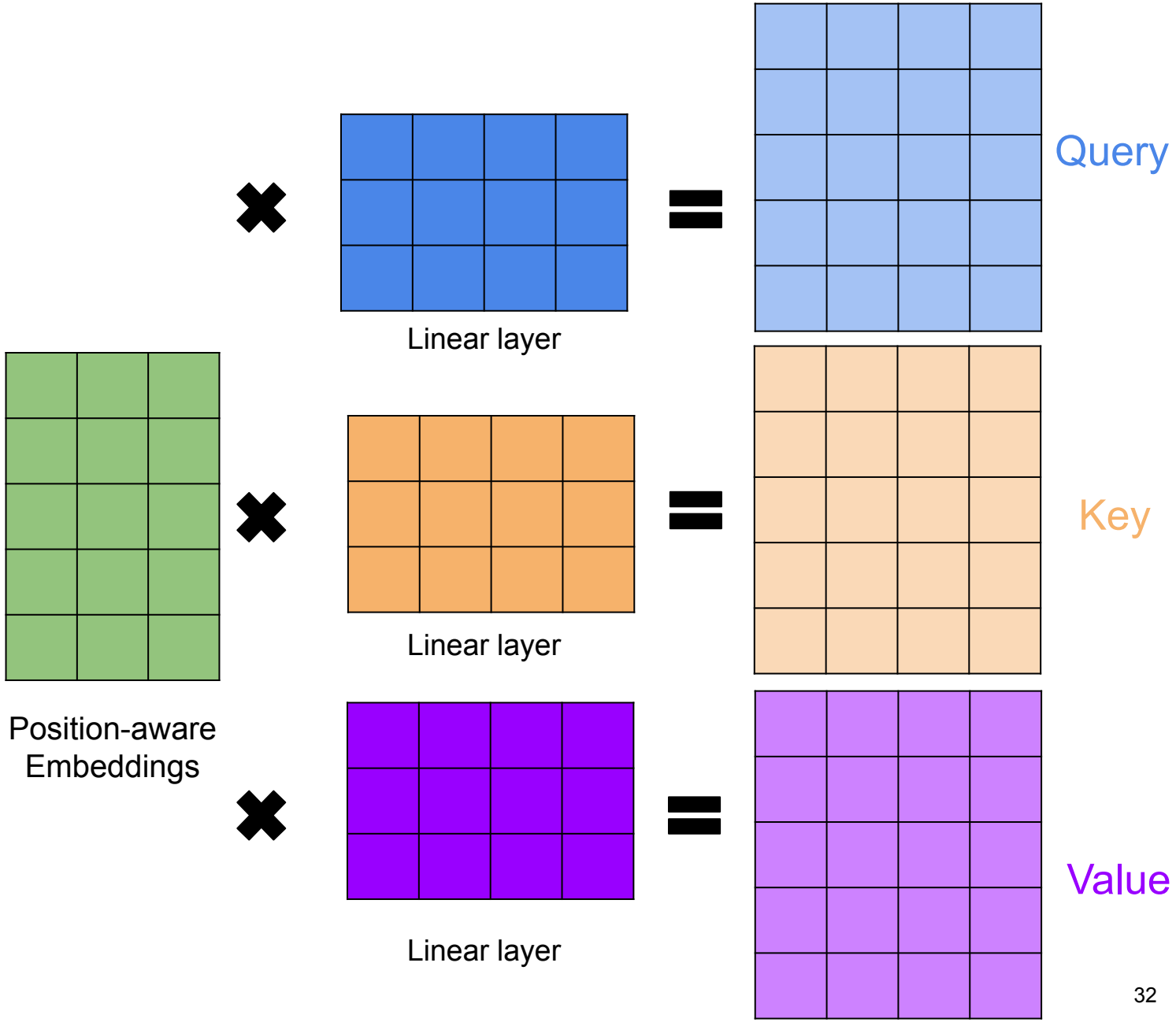
Self-attention with Words

1. Encode **position** information
2. Extract **query, key, value**
3. Compute **attention weighting**
4. Extract **features** with high attention



Self-attention with Words

1. Encode **position** information
2. Extract **query, key, value**
3. Compute **attention weighting**
4. Extract **features** with high attention

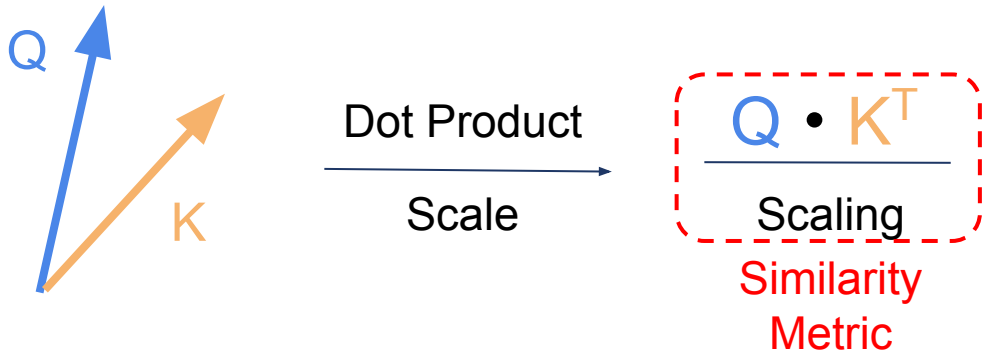


Self-attention with Words

1. Encode **position** information
2. Extract **query, key, value**
3. Compute **attention weighting**
4. Extract **features** with high attention

Attention Score: compute pairwise similarity between each query and key

How to compute similarity between two sets of features?



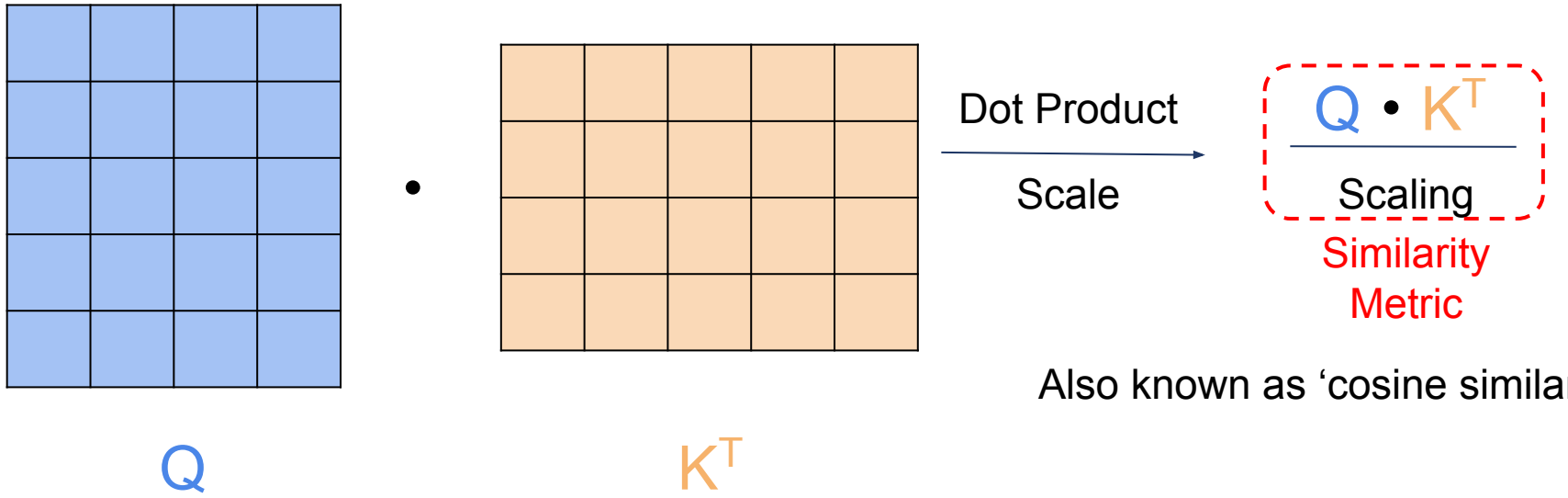
Also known as 'cosine similarity'

Self-attention with Words

1. Encode **position** information
2. Extract **query, key, value**
3. Compute **attention weighting**
4. Extract **features** with high attention

Attention Score: compute pairwise similarity between each query and key

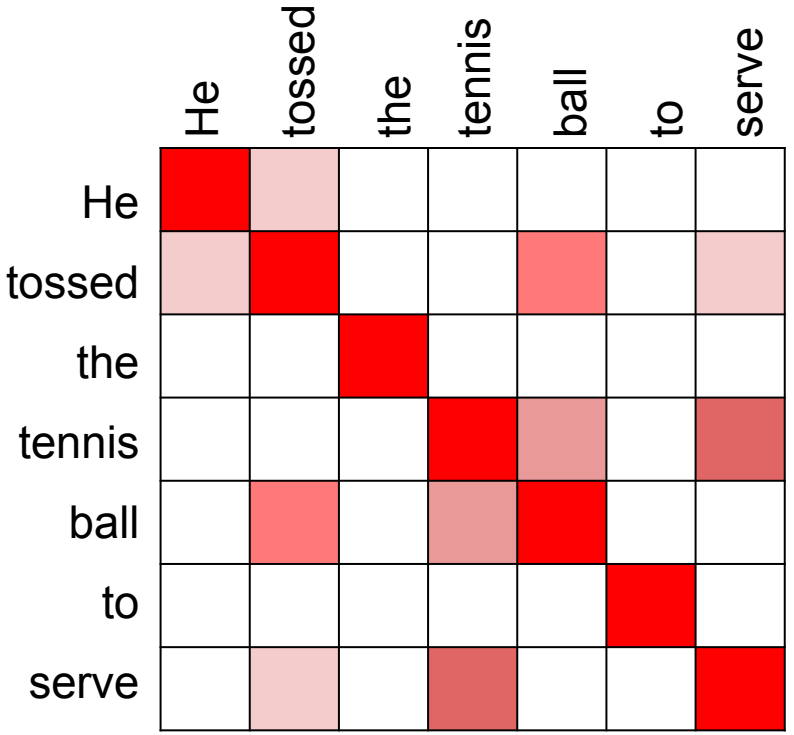
How to compute similarity between two sets of features?



Also known as 'cosine similarity'

Self-attention with Words

1. Encode **position** information
2. Extract **query, key, value**
3. Compute **attention weighting**
4. Extract **features** with high attention

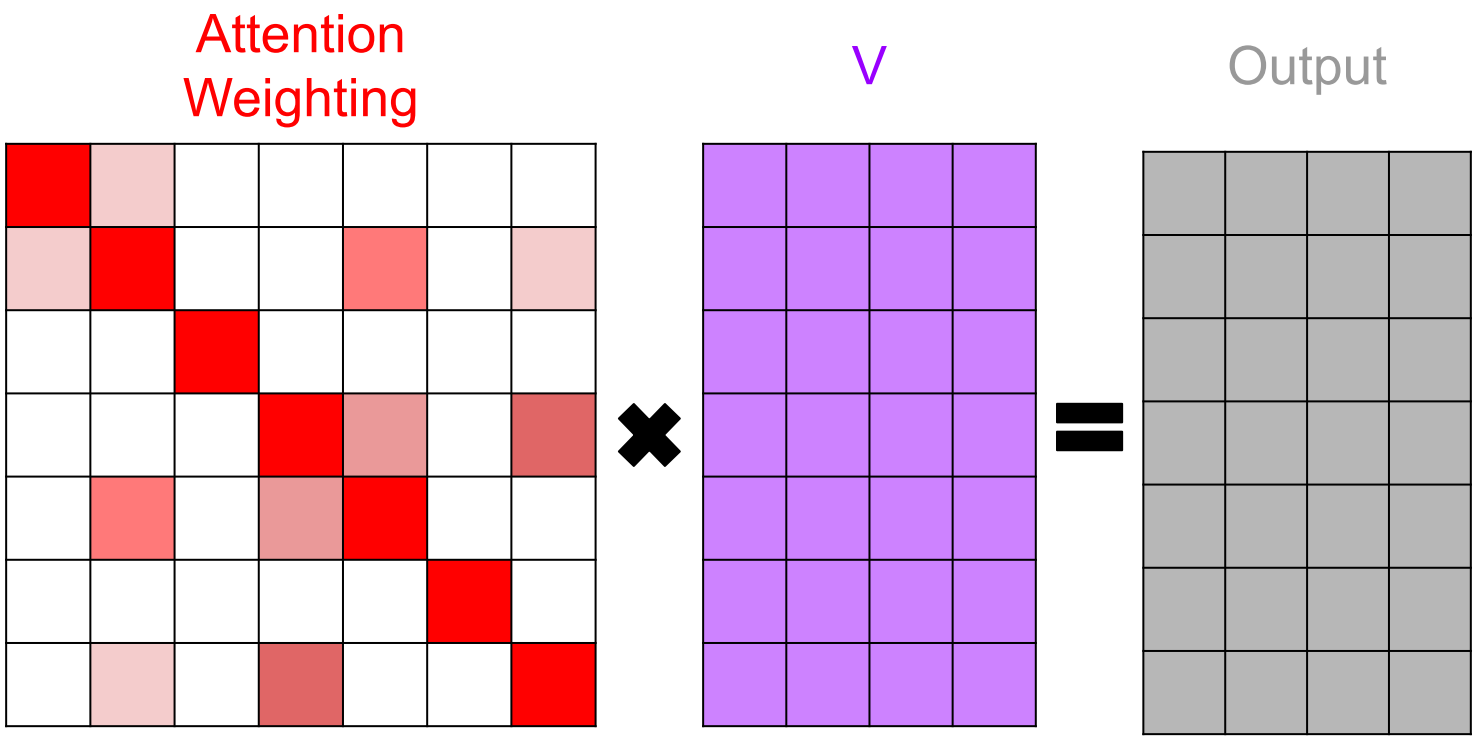


$$\text{softmax}\left[\frac{Q \cdot K^T}{\text{Scaling}}\right]$$

Attention Weighting

Self-attention with Words

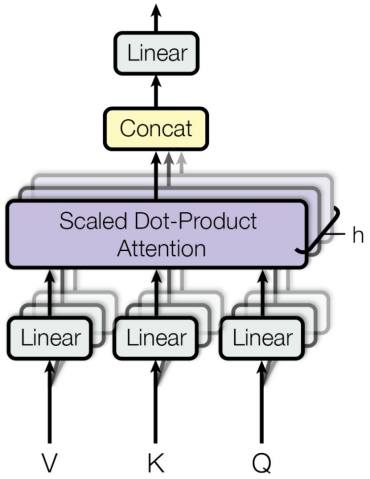
1. Encode **position** information
2. Extract **query, key, value**
3. Compute **attention weighting**
4. Extract **features** with high attention



$$\text{softmax}\left[\frac{Q \cdot K^T}{\text{Scaling}}\right] \cdot V = A(Q, K, V)$$

Multi-Head Attention

=



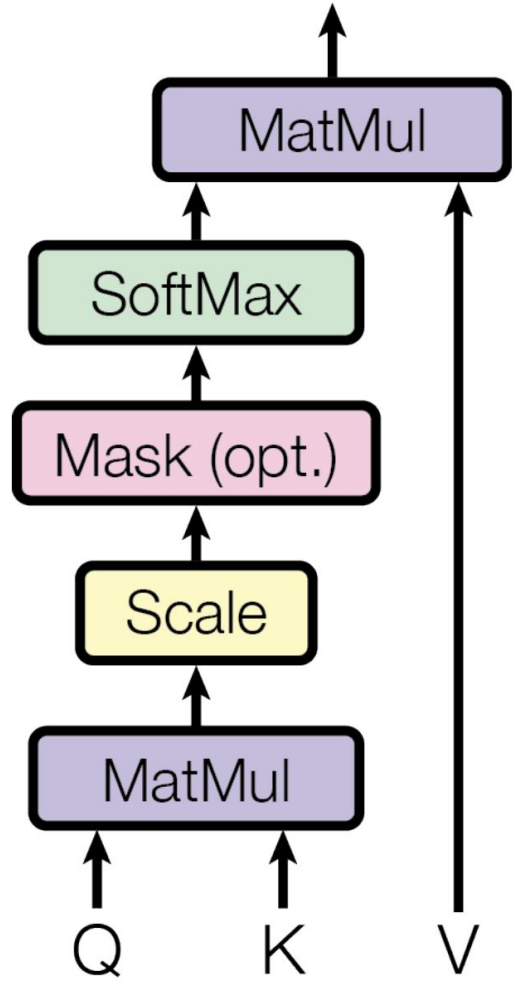
$$\text{softmax}\left[\frac{Q \cdot K^T}{\text{Scaling}}\right] \cdot V$$

Scaled Dot-Product Attention

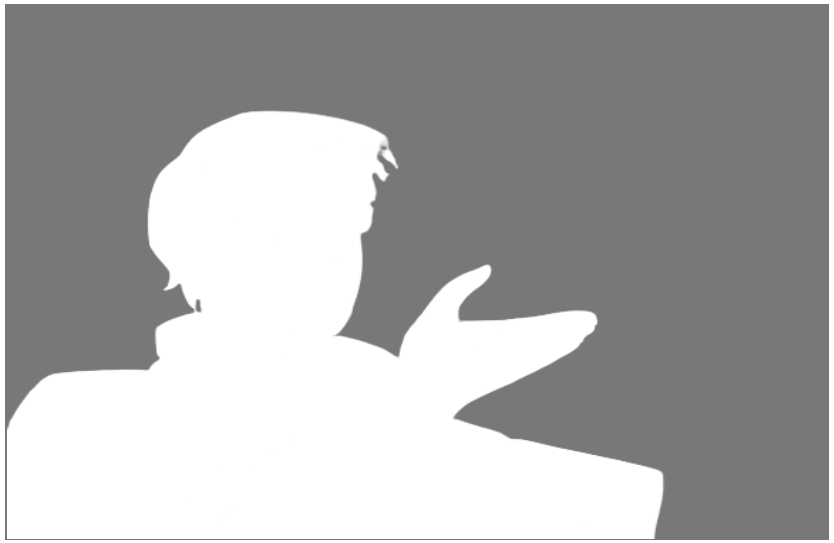
=

Self-Attention

=



Once again, Self-Attention with Images



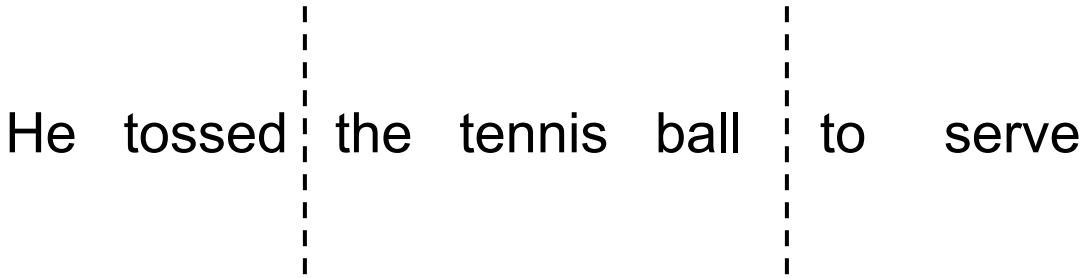
Attention Weighting

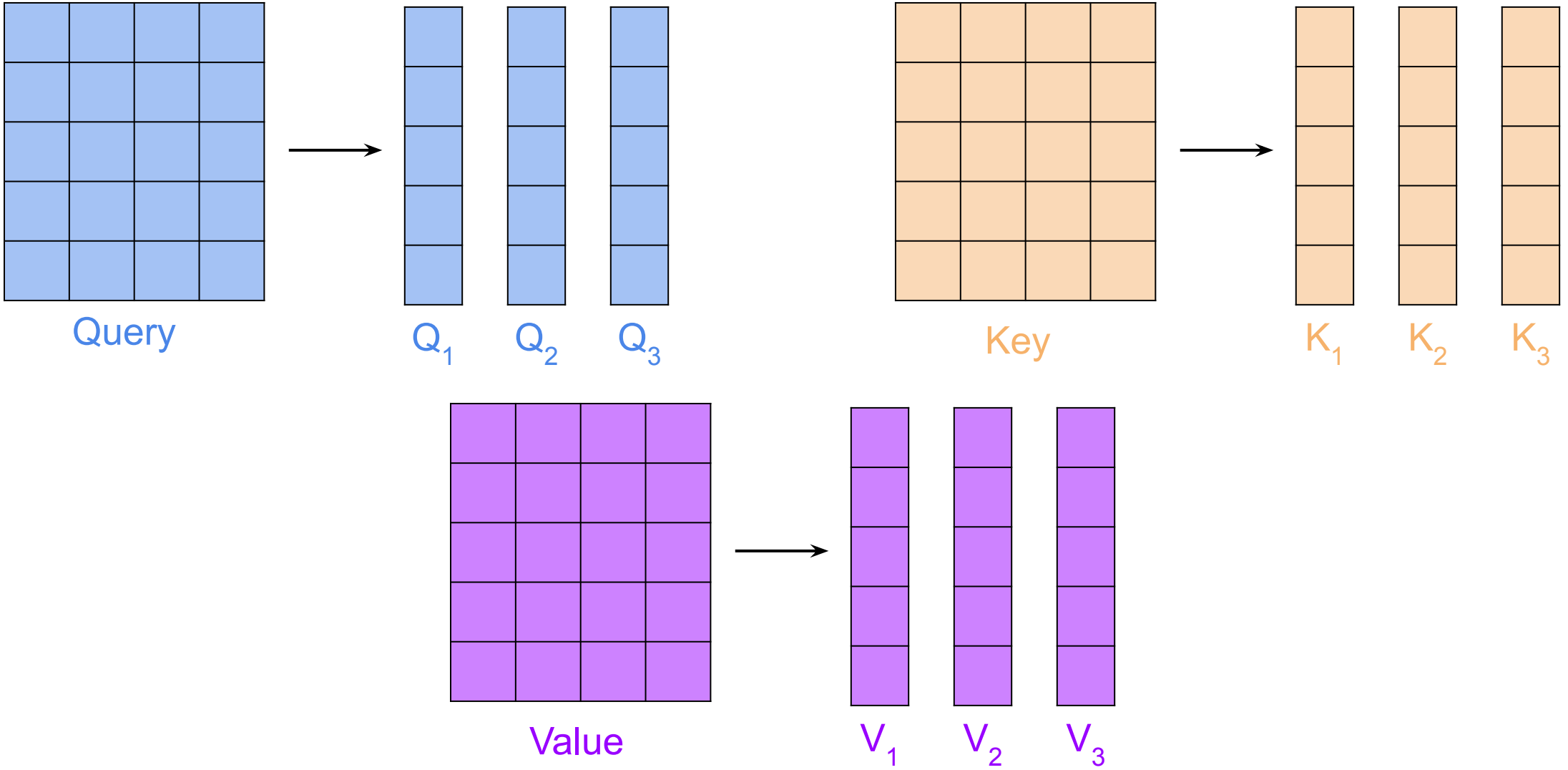
Value

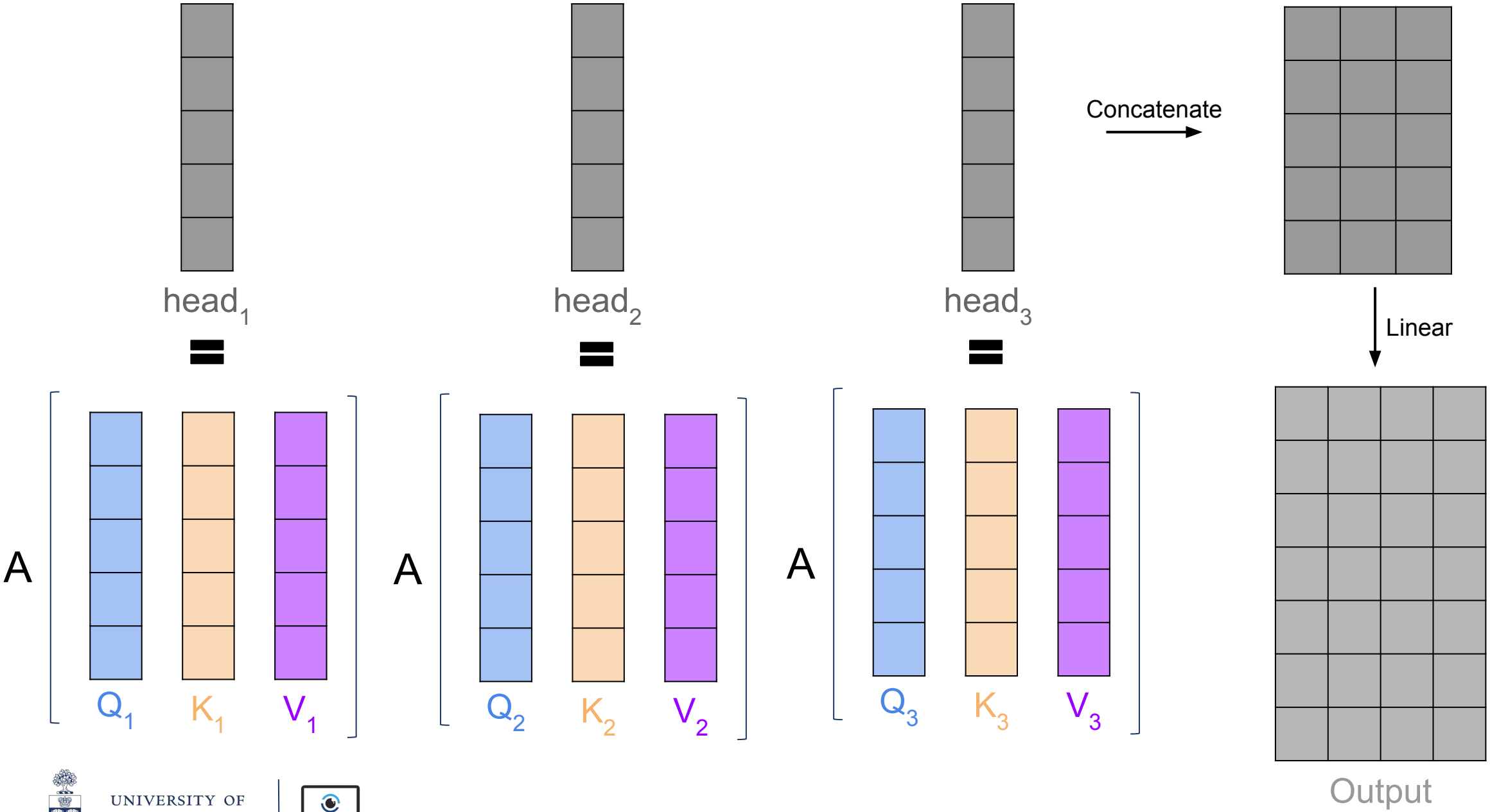
Output

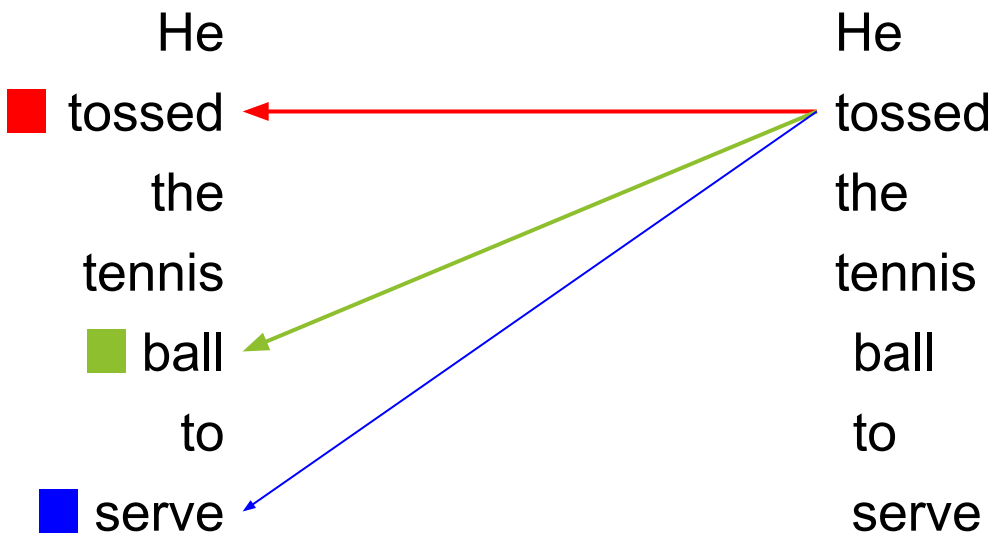
Now, onto Multi-Head Attention

- To get more information, let's split the input into h different parts.
- So we run h self-attention blocks on different parts of the input, simultaneously.
- For the machine translation task in the paper, h = 8.
- For our example, let us consider h = 3.




Query
Q_1
Q_2
Q_3
Key
K_1
K_2
K_3
Value
V_1
V_2
V_3


head1
head2
head3
Concatenate
Linear
A
A
A
Q1
K1
V1
Q2
K2
V2
Q3
K3
V3
Output


He
tossed
the
tennis
ball
to
serve
He
tossed
the
tennis
ball
to
serve

Multi-Head Attention with Images

PyTorch

```python
class MultiHeadAttentionBlock(nn.Module):
    def __init__(self, d_model: int, h: int, dropout: float) -> None:
        """Take in model size and number of heads."""
        super(MultiHeadAttention, self).__init__()
        self.d_model = d_model  # embedding vector size
        self.h = h   # number of heads
        # make sure d_model is divisible by h
        assert d_model % h == 0, "d_model is not divisible by h"
        # we assume d_v always equals d_k
        self.d_k = d_model // h  # dimension of vector seen by each
head    self.wq = nn.Linear(d_model, d_model, bias=False)  # Wq
        self.wk = nn.Linear(d_model, d_model, bias=False)  # Wk
        self.wv = nn.Linear(d_model, d_model, bias=False)  # Wv
        self.wo = nn.Linear(d_model, d_model, bias=False)  # Wo
        self.dropout = nn.Dropout(dropout)
```

PyTorch

```python
@staticmethod
    def attention(key, query, value, mask=None, dropout=None):
        """Compute scaled dot-product attention"""
        d_k = query.size(-1)
        # calculate the attention scores by applying scaled dot-product attention
        # (batch, h, seq_len, seq_len) --> (batch, h, seq_len, d_k)
        scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
        if mask is not None:
            # write a very low value (indicating -infinity) to the positions
            # where mask == 0, this will tell softmax to replace those values
            # with zero
            scores = scores.masked_fill(mask==0, -1e9)
            # alternatively, we can use the inplace operation masked_fill_
            # , where '_' after 'masked_fill' indicates inplace operation
            # scores.masked_fill_(mask==0, -1e9)
        # now, we convert the attention scores to probability scores by
        # applying softmax
        # note: all the probability scores of a particular datapoint must sum upto 1
        prob_scores = scores.softmax(dim=-1)  # (batch, h, seq_len, seq_len)
        if dropout is not None:
            prob_scores = dropout(prob_scores)
        # now, we (matrix) multiply prob_scores with value
        # so the shape changes from (batch, h, seq_len, seq_len)
        # to (batch, h, seq_len, d_k)
        # we also return the prob_scores, which can be used for visualization
        return torch.matmul(prob_scores, value), prob_scores
```

PyTorch

```
def forward(self, q, k, v, mask=None):
        # multiply Wq matrix by q
        # this matrix multiplication does not change the shape of q
        query = self.wq(q)  # (batch, h, seq_len)
        # similarly for key and value
        key = self.wk(k)    # (batch, h, seq_len)
        value = self.wv(v)  # (batch, h, seq_len)
        # (batch, seq_len, d_model) --> (batch, seq_len, h, d_k)
        query = query.view(query.shape[0], query.shape[1], self.h, self.d_k)
        # (batch, seq_len, h, d_k) --> (batch, h, seq_len, d_k)
        query = query.transpose(1,2) # interchange the indices 1 and 2 with each other
        # similarly the dimensions of key and value will also change
        key = key.view(key.shape[0], key.shape[1], self.h, self.d_k)
        key = key.transpose(1,2)
        value = value.view(value.shape[0], value.shape[1], self.h, self.d_k)
        value = value.transpose(1,2)
        # calculate attention
        x, self.attn_scores =  MultiHeadAttentionBlock.attention(query, key, value,
                                                                 mask, self.dropout)
```

PyTorch

```
        # combine all the heads together
        # (batch, h, seq_len, d_k) --> (batch, seq_len, h, d_k)
        x = x.transpose(1,2)
        # (batch, seq_len, h, d_k) --> (batch, seq_len, d_model)
        x = x.contiguous().view(x.shape[0], -1, self.h * self.d_k)
        # alternative code to the above line:
        # x = x.reshape(x.shape[0], -1, self.h * self.d_k)
        # now, multiply by Wo
        # this matrix multiplication does not change the shape of x
        # (batch, seq_len, d_model) --> (batch, seq_len, d_model)
        return self.wo(x)
```

TensorFlow

```python
class MultiHeadAttention(tf.keras.layers.Layer):
    def __init__(self, num_heads, d_model, dropout_rate=0.1):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.d_model = d_model
        self.dropout_rate = dropout_rate

        assert d_model % num_heads == 0

        self.depth = d_model // num_heads

        self.query_dense = tf.keras.layers.Dense(d_model)
        self.key_dense = tf.keras.layers.Dense(d_model)
        self.value_dense = tf.keras.layers.Dense(d_model)

        self.dense = tf.keras.layers.Dense(d_model)

    def split_heads(self, inputs, batch_size):
        inputs = tf.reshape(inputs, (batch_size, -1, self.num_heads,
self.depth))
        return tf.transpose(inputs, perm=[0, 2, 1, 3])
```

UNIVERSITY OF TORONTO | CVDG

TensorFlow

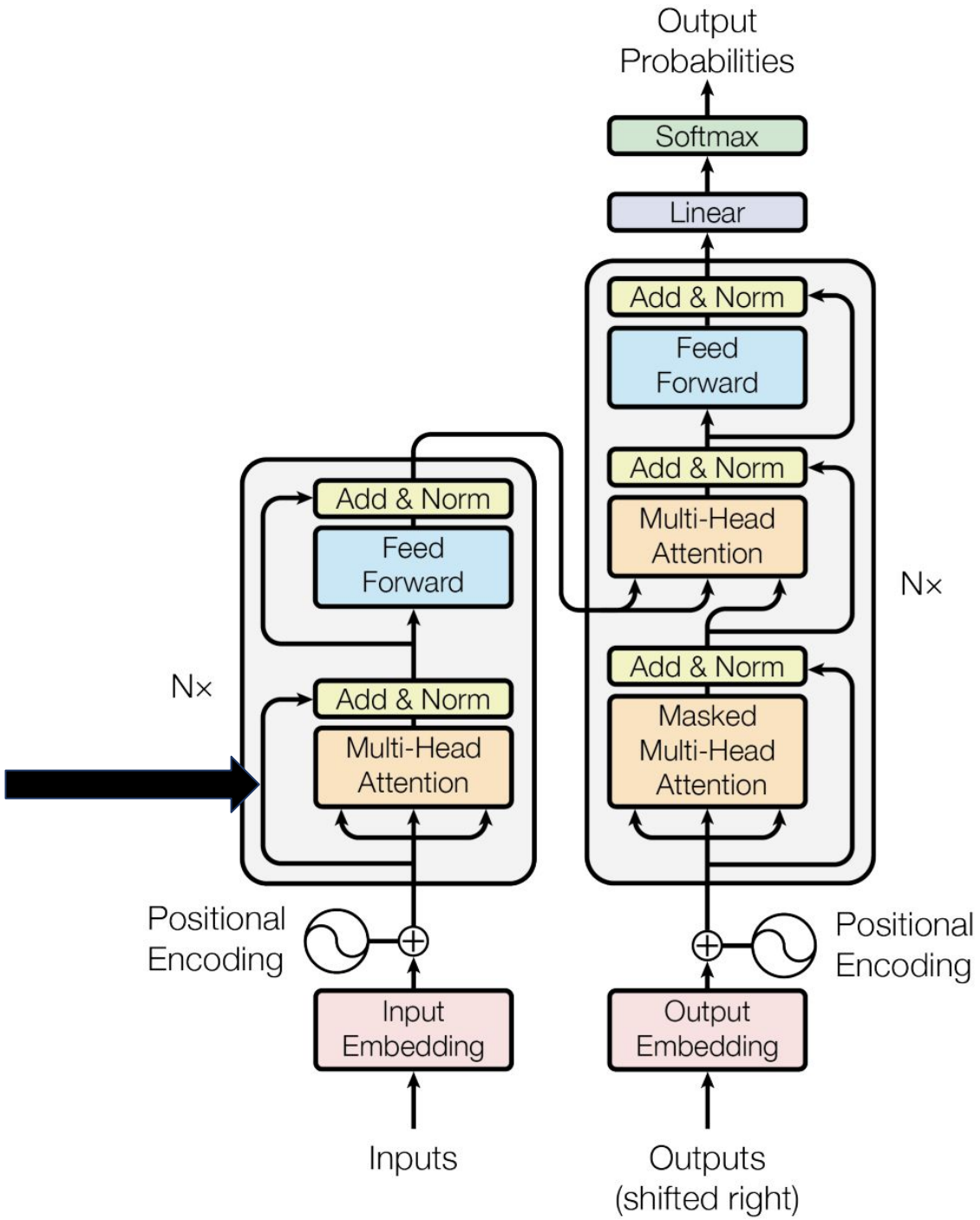
```python
  def scaled_dot_product_attention(self, q, k, v, mask=None):
      matmul_qk = tf.matmul(q, k, transpose_b=True)
      dk = tf.cast(tf.shape(k)[-1], tf.float32)
      scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

      if mask is not None:
          scaled_attention_logits += (mask * -1e9)

      attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
      output = tf.matmul(attention_weights, v)

      return output, attention_weights

  def call(self, inputs):
      query, key, value, mask = inputs
      batch_size = tf.shape(query)[0]

      query = self.query_dense(query)
      key = self.key_dense(key)
      value = self.value_dense(value)

      query = self.split_heads(query, batch_size)
      key = self.split_heads(key, batch_size)
      value = self.split_heads(value, batch_size)

      scaled_attention, attention_weights = self.scaled_dot_product_attention(query, key, value,
mask)
      scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])
      concat_attention = tf.reshape(scaled_attention, (batch_size, -1, self.d_model))
      output = self.dense(concat_attention)

      return output, attention_weights
```

TensorFlow

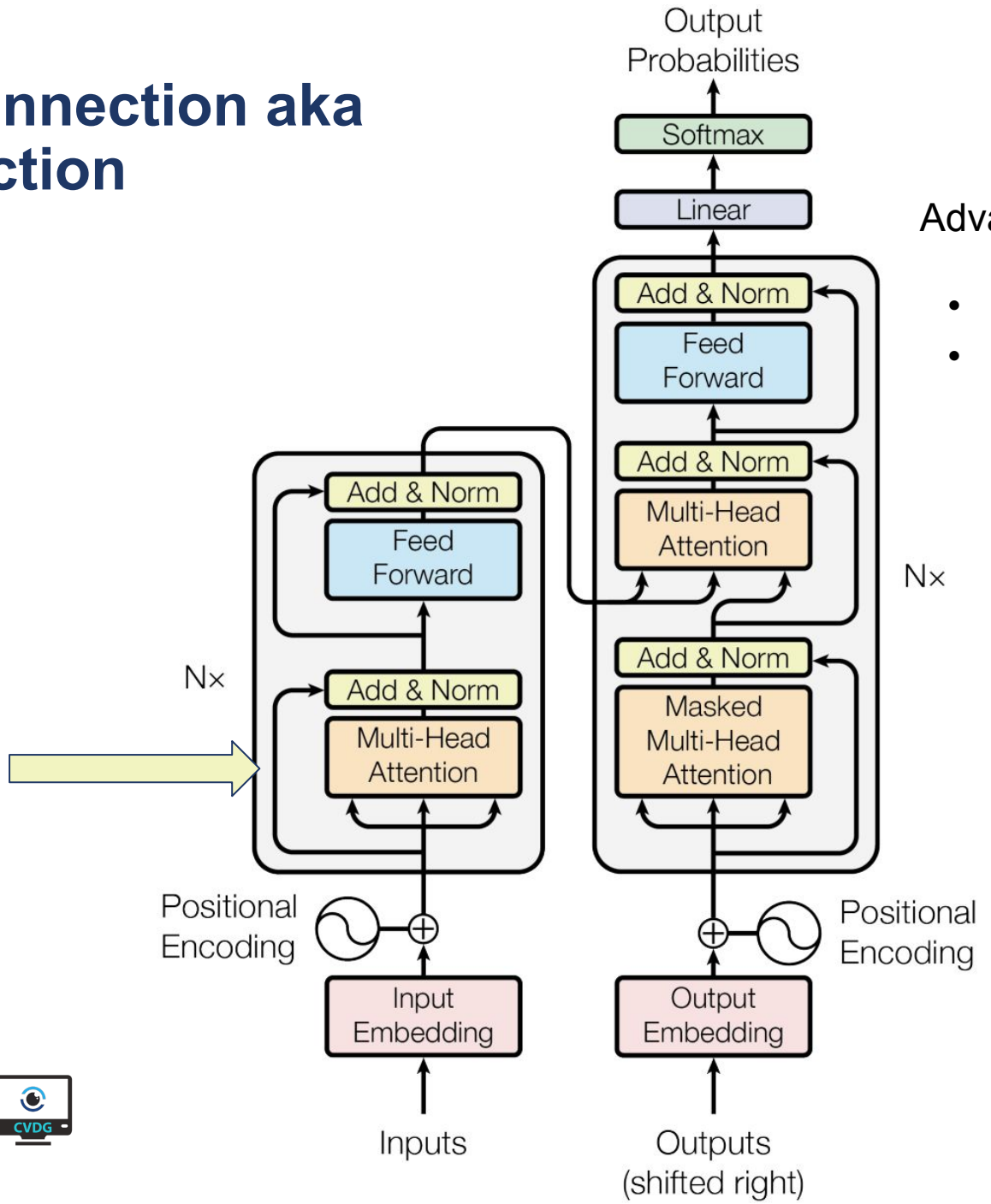
```python
class BaseAttention(tf.keras.layers.Layer):
  """
  Base Attention layer class that contains a MultiHeadAttention, LayerNormalization and Add layer.

  Attributes:
  -----------
  kwargs: dict
      keyword arguments that will be passed to the MultiHeadAttention layer during initialization.

  Methods:
  --------
  call(inputs, mask=None, training=None):
      Performs a forward pass on the input and returns the output.

  """
  def __init__(self, **kwargs):
    """
    Initializes a new instance of the BaseAttention layer class.

    Parameters:
    -----------
    kwargs: dict
        keyword arguments that will be passed to the MultiHeadAttention layer during initialization.
    """
    super().__init__()
    self.mha = tf.keras.layers.MultiHeadAttention(**kwargs)
    self.layernorm = tf.keras.layers.LayerNormalization()
    self.add = tf.keras.layers.Add()
```

TensorFlow

```python
class GlobalSelfAttention(BaseAttention):
  def call(self, x):
    """
    Apply the global self-attention mechanism to the input sequence.

    Args:
        x: A tensor of shape `(batch_size, seq_len, embedding_dim)`
        representing the input sequence.

    Returns:
        A tensor of the same shape as the input, representing the sequence
        after being transformed by the self-attention mechanism.
    """
    attn_output = self.mha(
        query=x,
        value=x,
        key=x)
    x = self.add([x, attn_output])
    x = self.layernorm(x)
    return x
```


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Residual Connection aka Skip Connection

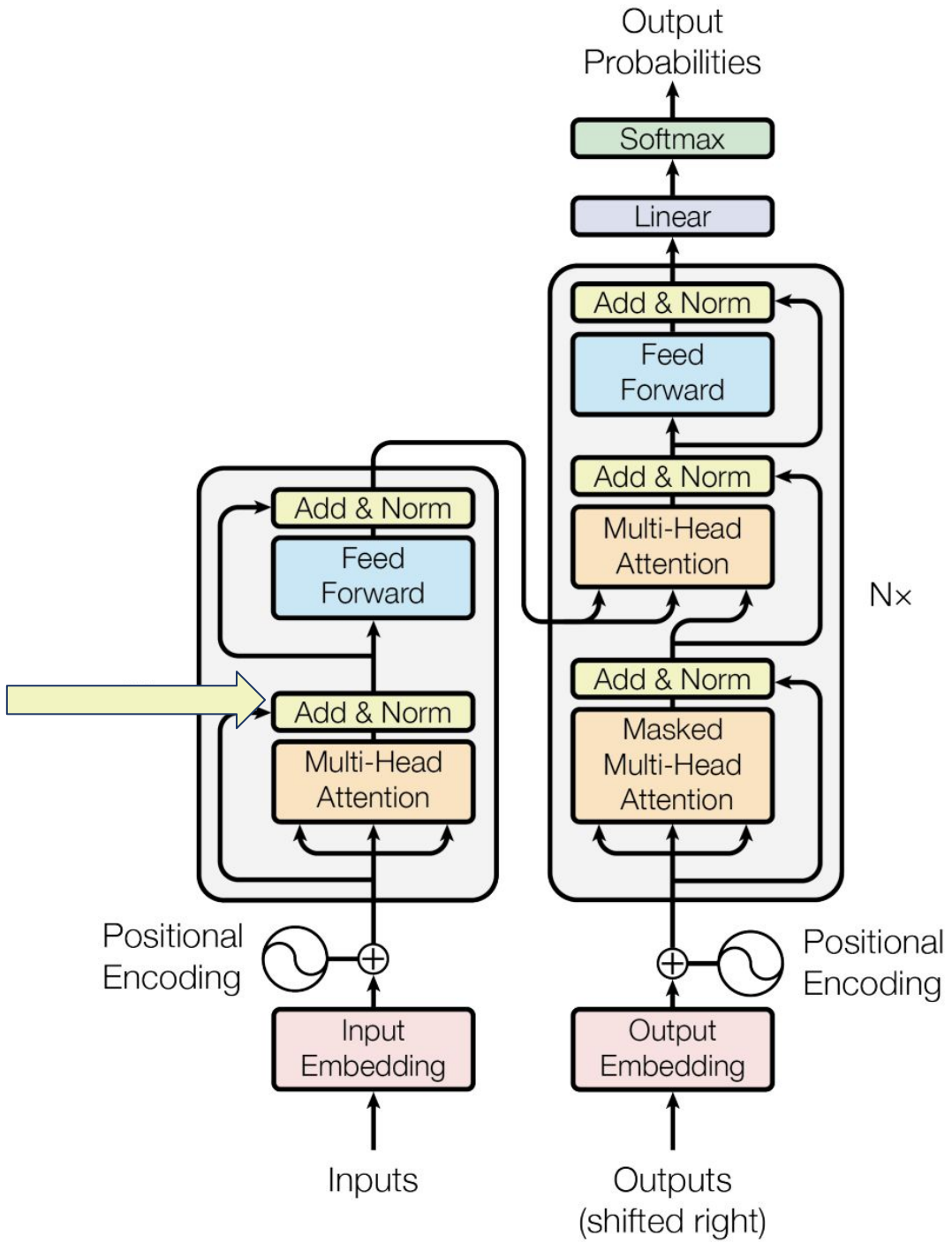


Advantages:

- Helps fight gradient vanishing problem
- Helps in building deep layered (lots of layers) neural networks

PyTorch

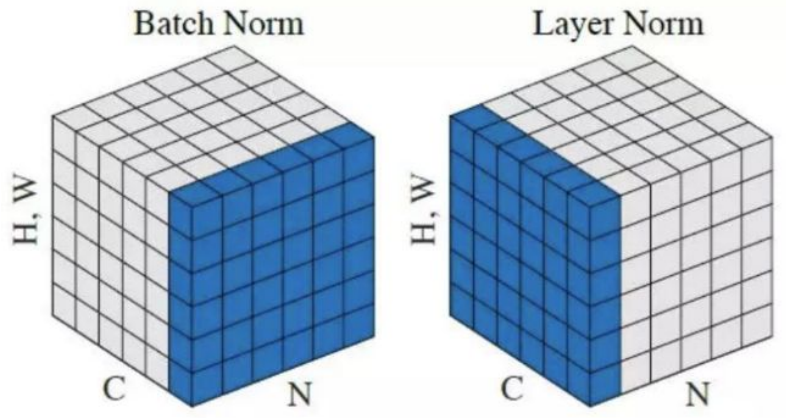
```python
class ResidualConnection(nn.Module):
    """This is the 'add' part in the 'add and norm' block."""
    def __init__(self, features: int, dropout: float) -> None:
        super(ResidualConnection, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        self.norm = LayerNormalization(features)

    def forward(self, x, sublayer):
        """
        x: input
        sublayer: different layers of the transformer architecture (eg: multi-head
        attention, feed-forward network, etc.)

        Returns the skip or residual connection.
        """
        # most implementations first do normalization and then pass x to the sublayer
        # we will also do this way
        return x + self.dropout(sublayer(self.norm(x)))
        # however, the paper first passes x to the sublayer and then does the norm
        # return x + self.dropout(self.norm(sublayer(x)))
```


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Layer Normalization

Batch of 3 items

ITEM 1	ITEM 2	ITEM 3
50.147	1242.223	9.370
3314.825	688.123	4606.674
...	...	...
...	...	...
8463.361	434.944	944.705
8.021	149.442	21189.444
μ_1	μ_2	μ_3
σ_1^2	σ_2^2	σ_3^2



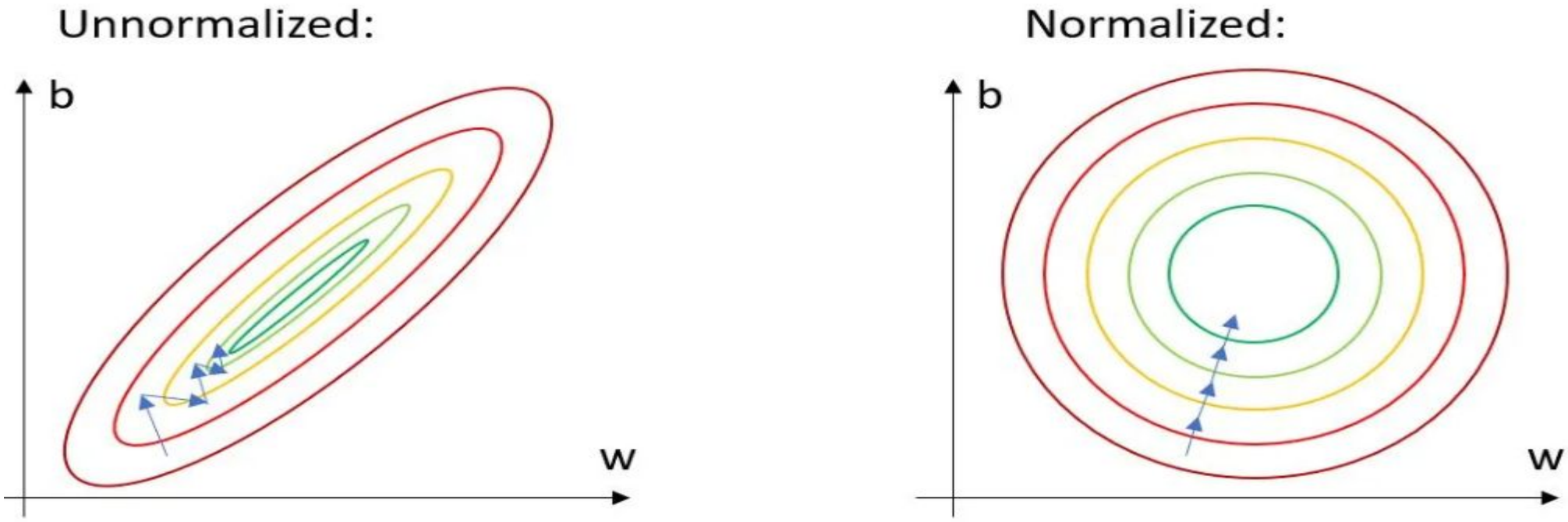
$$\hat{x}_j = \frac{x_j - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

We also introduce two parameters, usually called **gamma** (multiplicative) and **beta** (additive) that introduce some fluctuations in the data, because maybe having all values between 0 and 1 may be too restrictive for the network. The network will learn to tune these two parameters to introduce fluctuations when necessary.

Source: Umar Jamil's Transformers from Scratch

Advantages:

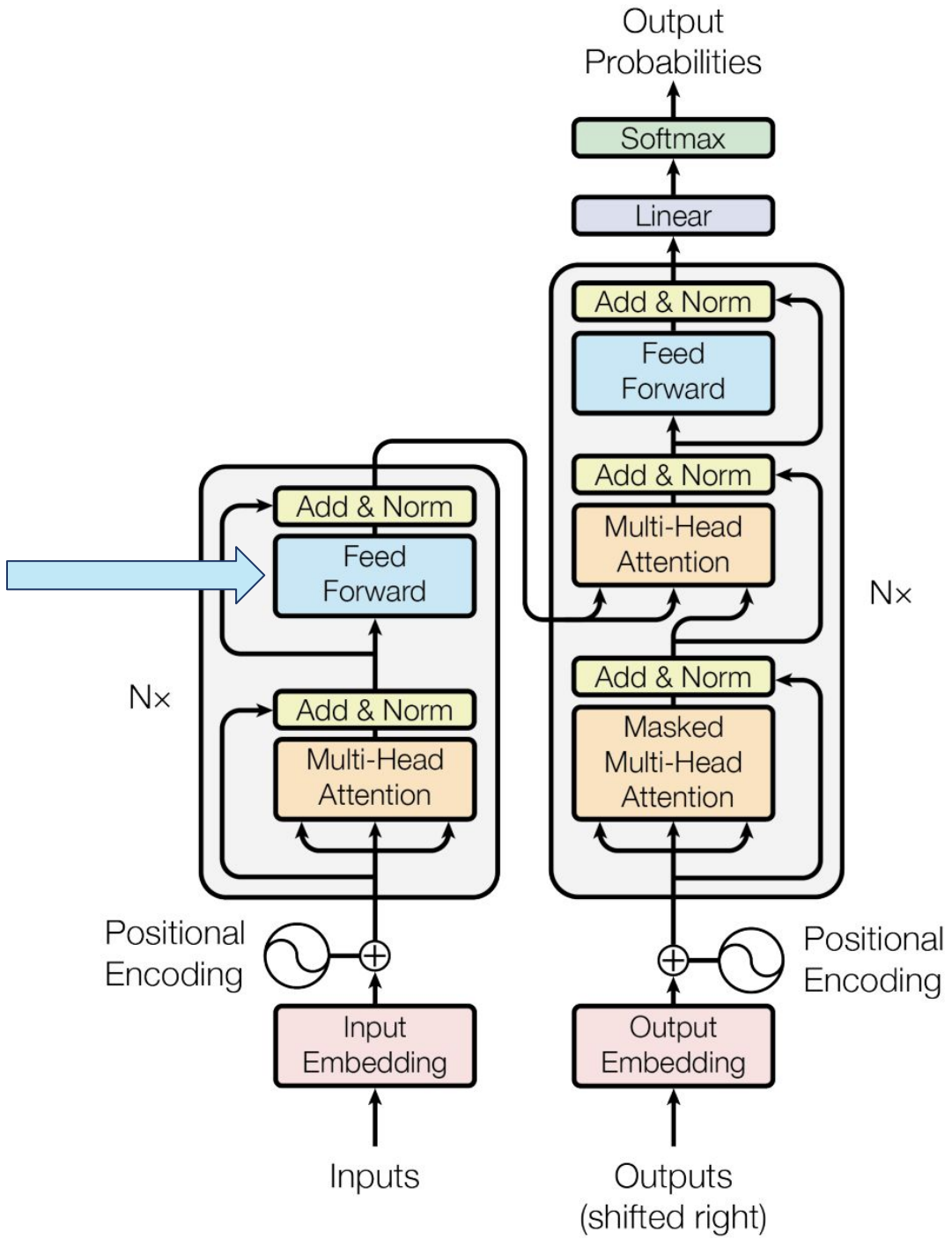
- Faster training
- Helps fight gradient (weight) explosion problem
-



[Source: Jorrit Willaert]

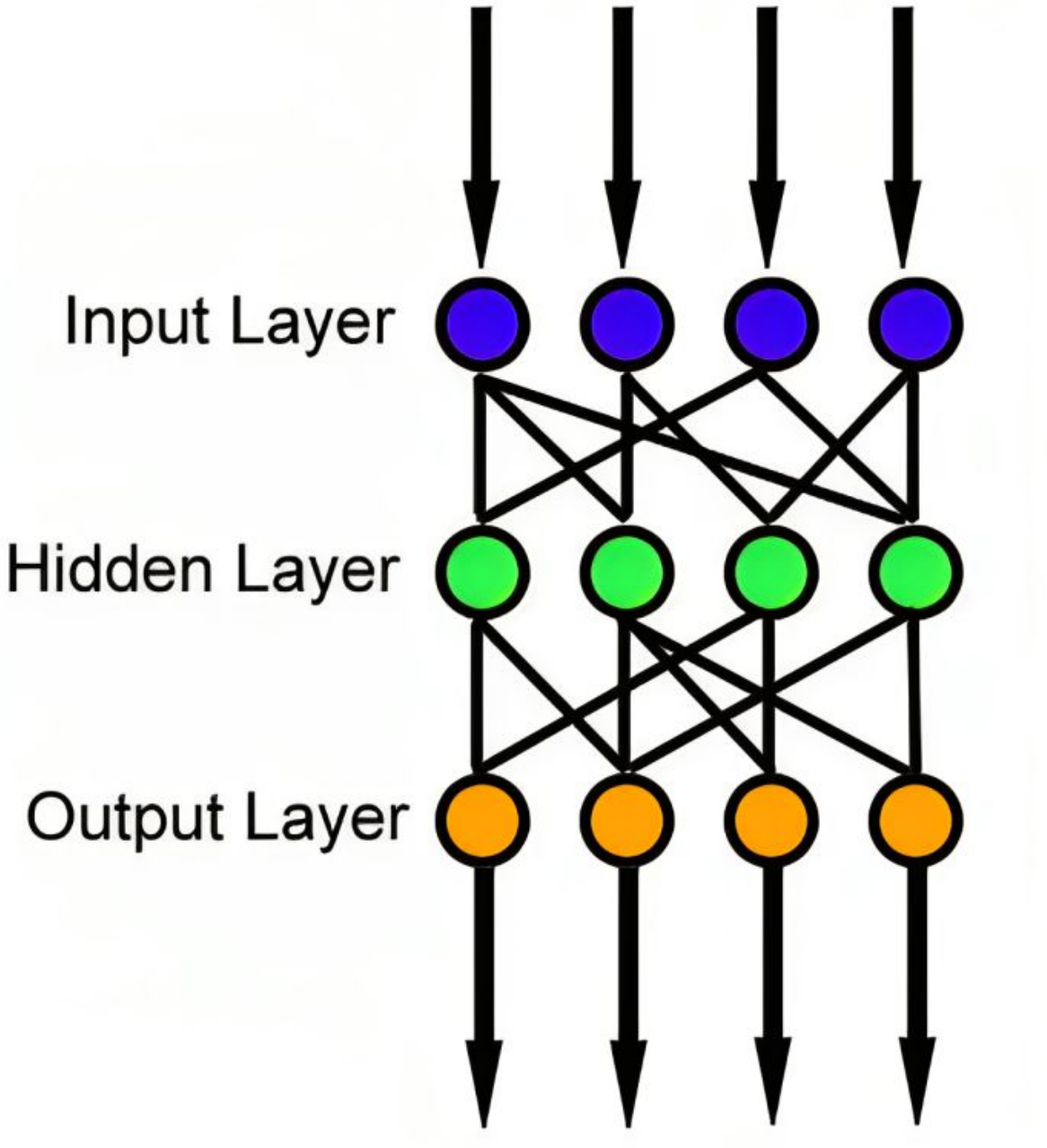
PyTorch

```python
class LayerNormalization(nn.Module):
    def __init__(self, eps: float = 1e-6) -> None:
        super(LayerNormalization, self).__init__()
        self.eps = eps
        # instead of simply doing self.alpha = torch.ones(1)
        # we use nn.Parameter() so that when we call the state dict of the model
        # we are able to see this alpha
        # only using torch.ones(1) won't allow us to see this alpha
        self.alpha = nn.Parameter(torch.ones(1))  # multiplied
        self.bias = nn.Parameter(torch.zeros(1))  # added

    def forward(self, x):
        # apply mean after the batch dimension
        # mean usually cancels the dimension to which it is applied,
        # but we want to keep it
        mean = x.mean(dim=-1, keepdim=True)
        # similarly for standard deviation
        std = x.std(dim=-1, keepdim=True)
        return self.alpha * ((x-mean)/(std**2 + self.eps)) + self.bias
```


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Feed Forward Network

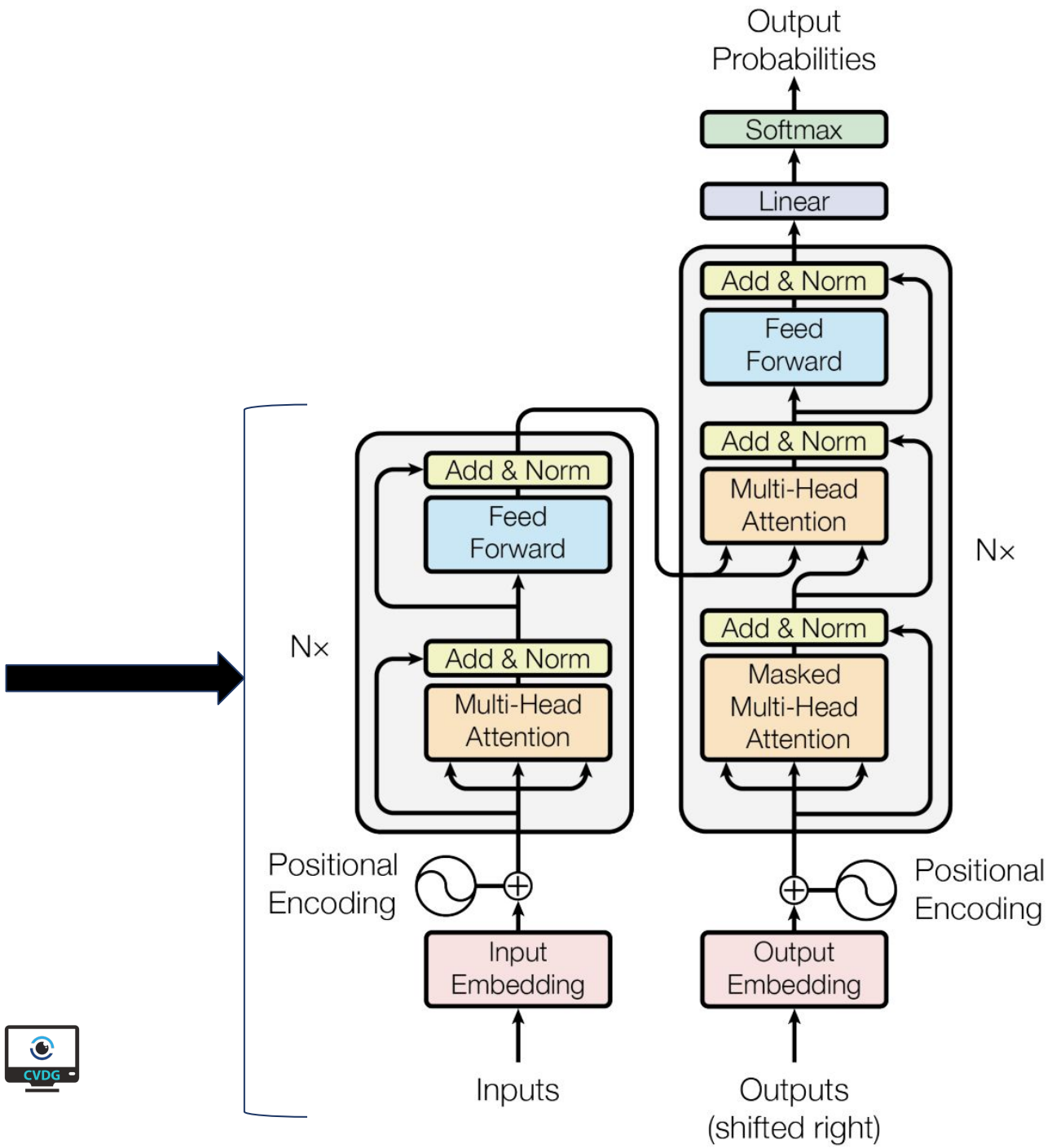
- Just glorified matrix multiplication
- The main purpose of this network is to predict the next element of a sequence



Source: Wikipedia

PyTorch

```python
class PositionWiseFeedForward(nn.Module):
    """Implements the FFN equation."""
    def __init__(self, d_model: int, d_ff: int, dropout: float) -> None:
        super(PositionWiseFeedForward, self).__init__()
        self.linear1 = nn.Linear(d_model, d_ff)   # W1 and B1
        self.linear2 = nn.Linear(d_ff, d_model)   # W2 and B2
        self.dropout = nn.Dropout(p=dropout)
        self.relu = nn.ReLU()

    def forward(self, x):
        """
        x is of the shape: (batch, seq_len, d_model)
        linear1 is of the shape: (d_model, d_ff)
        linear2 is of the shape: (d_ff, d_model)

        On multiplying x with linear1, the shape of x becomes (batch, seq_len, d_ff)
        On multiplying the new x with linear2, the shape of x changes back to the
        original one, ie, (batch, seq_len, d_model)
        """
        x = self.relu(self.linear1(x))
        x = self.dropout(x)
        x = self.linear2(x)
        return x
```


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

PyTorch

```python
class EncoderBlock(nn.Module):
    def __init__(self, features: int, selfattn_block: MultiHeadAttentionBlock,
                 feedforward_block: PositionWiseFeedForward, dropout: float) ->
None:   self.selfattn_block = selfattn_block
        self.feedforward_block = feedforward_block
        # store 2 residual connection layers
        # we'l use one after self-attention layer and the other after feed-forward
        # network as shown in figure 1 of the paper
        self.res_con = nn.ModuleList([ResidualConnection(features, dropout)
                                      for _ in range(2)])

    def forward(self, x, src_mask):
        # we apply the source mask because we don't want the padding word to
        # interact with other words
        x = self.res_con[0](x, lambda x: self.selfattn_block(x,x,x,src_mask))
        x = self.res_con[1](x, self.feedforward_block)
```

PyTorch

```
class Encoder(nn.Module):
    def __init__(self, features: int, layers: nn.ModuleList) ->
None:   super(Encoder, self).__init__()
        self.layers = layers
        self.norm = LayerNormalization(features)

    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

TensorFlow

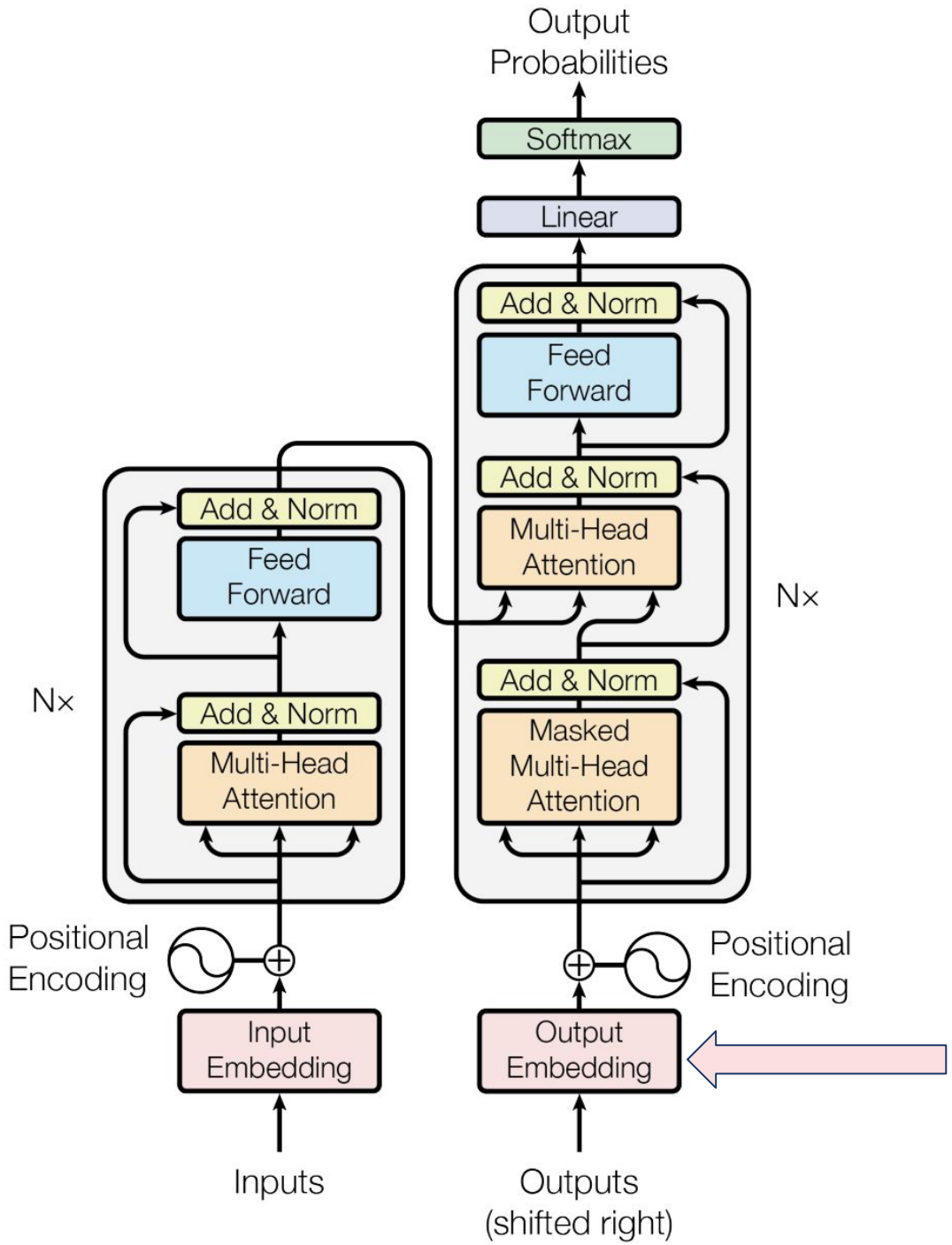
```python
class EncoderLayer(tf.keras.layers.Layer):
  """
  Args:
    d_model (int): The dimensionality of the input and output sequences.
    num_heads (int): The number of attention heads to be used in the self-attention sub-layer.
    dff (int): The number of hidden units in the feedforward sub-layer.
    dropout_rate (float): The dropout rate to be applied after the self-attention sub-layer.

  Attributes:
    self_attention (GlobalSelfAttention): A self-attention layer.
    ffn (FeedForward): A feedforward neural network layer.
  """
  def __init__(self,*, d_model, num_heads, dff, dropout_rate=0.1):
    super().__init__()

    self.self_attention = GlobalSelfAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.ffn = FeedForward(d_model, dff)

  def call(self, x):
    """
    Applies the forward pass of the encoder layer.

    Args:
      x (tf.Tensor): The input sequence tensor.

    Returns:
      tf.Tensor: The output sequence tensor.
    """
    x = self.self_attention(x)
    x = self.ffn(x)
    return x
```

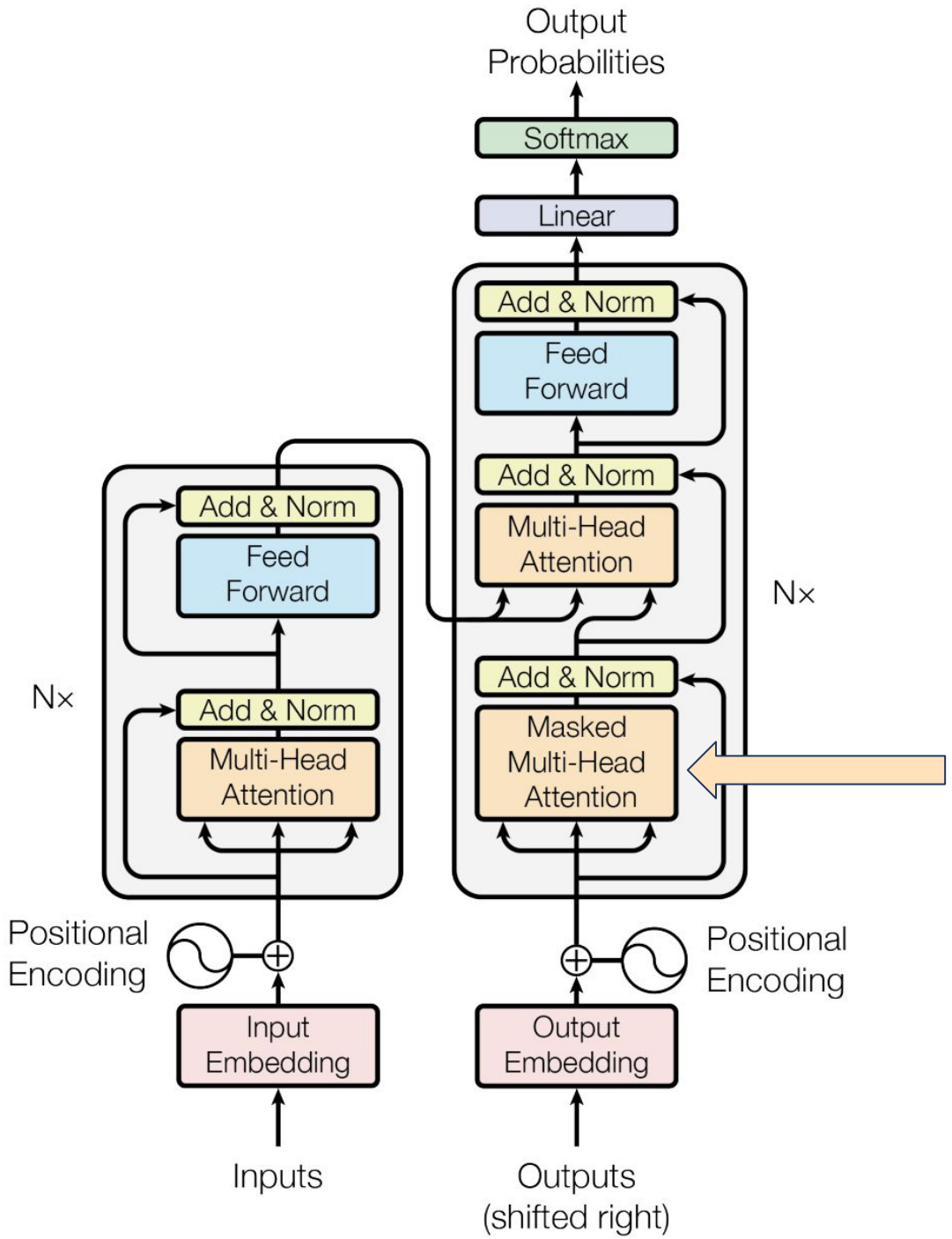
UNIVERSITY OF TORONTO | CVDG

TensorFlow

```python
class Encoder(tf.keras.layers.Layer):
  """
  A custom Keras layer that implements the encoder of a transformer-based
  neural network architecture for natural language processing tasks such
  as language translation or text classification.
  """
  def __init__(self, *, num_layers, d_model, num_heads,
               dff, vocab_size, dropout_rate=0.1):
    super().__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(
        vocab_size=vocab_size, d_model=d_model)

    self.enc_layers = [
        EncoderLayer(d_model=d_model,
                     num_heads=num_heads,
                     dff=dff,
                     dropout_rate=dropout_rate)
        for _ in range(num_layers)]
    self.dropout = tf.keras.layers.Dropout(dropout_rate)

  def call(self, x):
    """
    Perform forward pass of the `Encoder` layer.
    """
    # `x` is token-IDs shape: (batch, seq_len)
    x = self.pos_embedding(x)  # Shape `(batch_size, seq_len, d_model)`.

    # Add dropout.
    x = self.dropout(x)

    for i in range(self.num_layers):
      x = self.enc_layers[i](x)

    return x  # Shape `(batch_size, seq_len, d_model)`.
```

UNIVERSITY OF TORONTO | CVDG


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

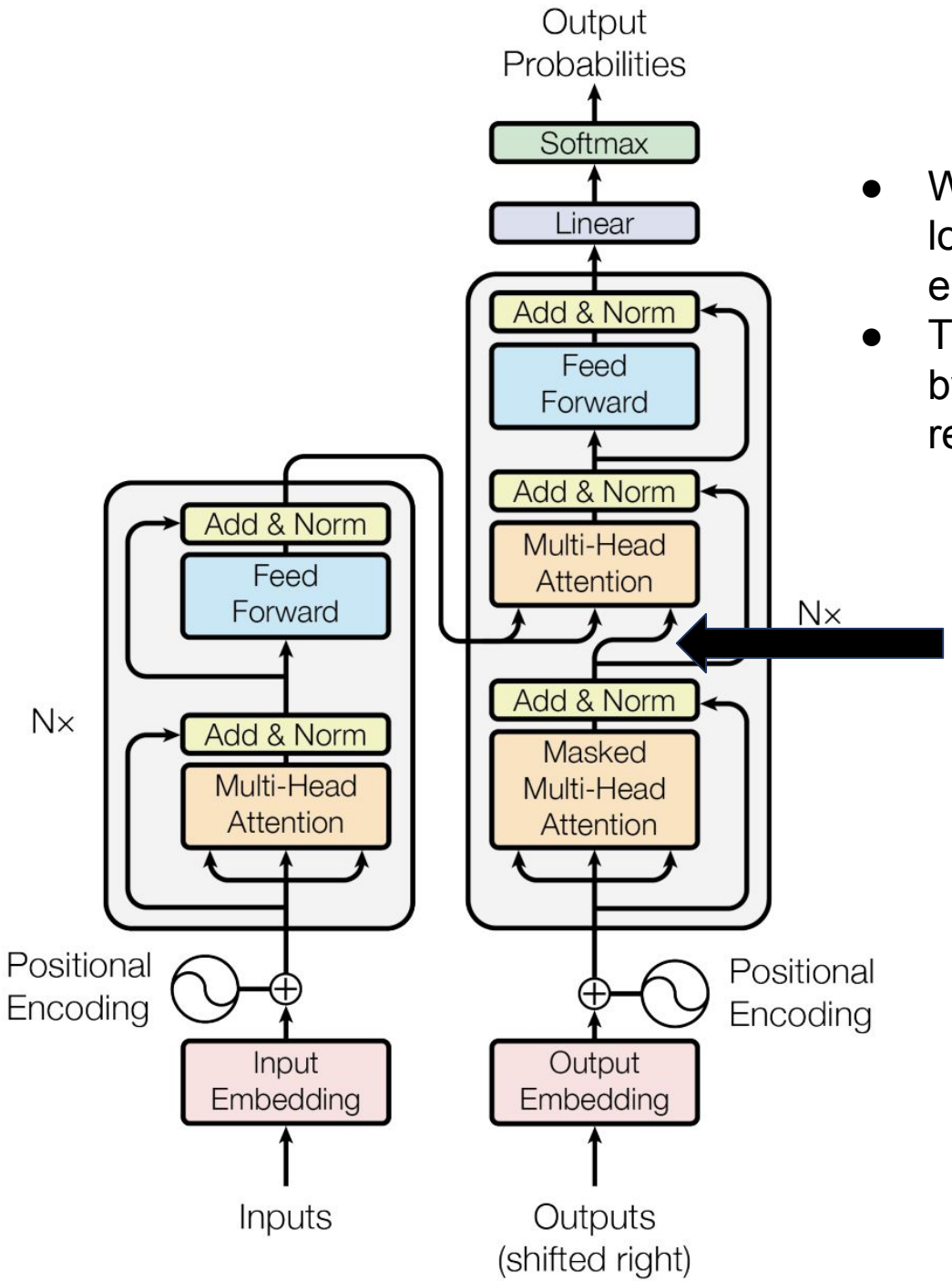

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Masked Multi Head Attention

- Write a very low value (indicating -infinity) to the positions where mask is 0
- This will tell softmax to replace those values with zero
- This will prevent the model from “cheating”, that is, force the model to predict the next words by only using the words before it

	YOUR	CAT	IS	A	LOVELY	CAT
YOUR	0.268	~~0.119~~	~~0.134~~	~~0.148~~	~~0.179~~	~~0.152~~
CAT	0.124	0.278	~~0.201~~	~~0.128~~	~~0.154~~	~~0.115~~
IS	0.147	0.132	0.262	~~0.097~~	~~0.218~~	~~0.145~~
A	0.210	0.128	0.206	0.212	~~0.119~~	~~0.125~~
LOVELY	0.146	0.158	0.152	0.143	0.227	~~0.174~~
CAT	0.195	0.114	0.203	0.103	0.157	0.229

Source: Umar Jamil’s Transformers from Scratch



- We take the queries from the decoder and look for the keys and values in the encoder representation.
- This helps us in predicting the next word by using the information from the encoder representation.

PyTorch

```python
class DecoderBlock(nn.Module):
    def __init__(self, features: int, selfattn_block: MultiHeadAttentionBlock,
                 crossattn_block: MultiHeadAttentionBlock, dropout: float,
                 feedforward_block: PositionWiseFeedForward) -> None:
        super(DecoderBlock, self).__init__()
        self.selfattn_block = selfattn_block
        self.crossattn_block = crossattn_block
        self.feedforward_block = feedforward_block
        self.res_con = nn.ModuleList([ResidualConnection(features, dropout)
                                      for _ in range(3)])

    def forward(self, x, encoder_output, src_mask, tgt_mask):
        x = self.res_con[0](x, lambda x: self.selfattn_block(x, x, x, tgt_mask))
        x = self.res_con[1](x, lambda x: self.crossattn_block(x, encoder_output,
                                                              encoder_output,
src_mask))= self.res_con[2](x, self.feedforward_block)
        return x
```

PyTorch

```python
class Decoder(nn.Module):
    def __init__(self, features: int, layers:
nn.ModuleList) -> None:
        super(Decoder, self).__init__()
        self.layers = layers
        self.norm = LayerNormalization(features)

    def forward(self, x, encoder_output, src_mask,
tgt_mask):
        for layer in self.layers:
            x = layer(x, encoder_output, src_mask, tgt_mask)
        return self.norm(x)
```

TensorFlow

```python
class DecoderLayer(tf.keras.layers.Layer):
  def __init__(self,
               *,
               d_model,
               num_heads,
               dff,
               dropout_rate=0.1):
    super(DecoderLayer, self).__init__()

    self.causal_self_attention = CausalSelfAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.cross_attention = CrossAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.ffn = FeedForward(d_model, dff)

  def call(self, x, context):
    """
    Forward pass of the `DecoderLayer`.
    """
    x = self.causal_self_attention(x=x)
    x = self.cross_attention(x=x, context=context)

    # Cache the last attention scores for plotting later
    self.last_attn_scores =
self.cross_attention.last_attn_scores
    x = self.ffn(x)  # Shape `(batch_size, seq_len, d_model)`.
    return x
```

UNIVERSITY OF TORONTO

CVDG

TensorFlow

```python
class Decoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads, dff,
vocab_size,    dropout_rate=0.1):
    super(Decoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(vocab_size=vocab_size,
                                             d_model=d_model)
    self.dropout = tf.keras.layers.Dropout(dropout_rate)
    self.dec_layers = [
        DecoderLayer(d_model=d_model, num_heads=num_heads,
                     dff=dff, dropout_rate=dropout_rate)
        for _ in range(num_layers)]

    self.last_attn_scores = None

  def call(self, x, context):
    """
    Implements the forward pass for the decoder layer.
    """
    # `x` is token-IDs shape (batch, target_seq_len)
    x = self.pos_embedding(x)  # (batch_size, target_seq_len, d_model)

    x = self.dropout(x)

    for i in range(self.num_layers):
      x  = self.dec_layers[i](x, context)

    self.last_attn_scores = self.dec_layers[-1].last_attn_scores

    # The shape of x is (batch_size, target_seq_len, d_model).
    return x
```

UNIVERSITY OF TORONTO | CVDG

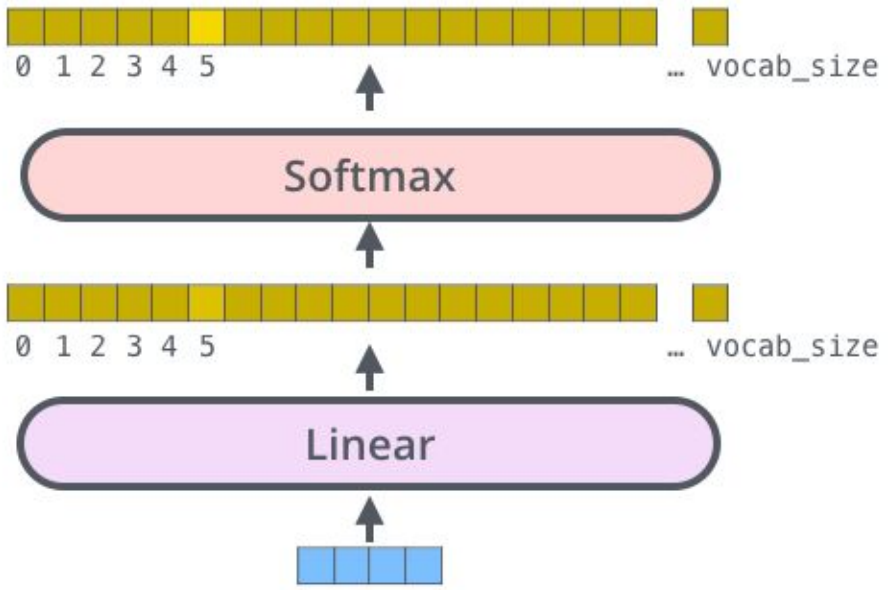
Linear - Softmax - Output

Which word in our vocabulary is associated with this index?

am

Get the index of the cell with the highest value (**argmax**)

5



Source: Jay Alammar's Blog

PyTorch

```python
class Transformer(nn.Module):
    def __init__(self, encoder: Encoder, decoder: Decoder, src_embed:
InputEmbedding,  tgt_embed: InputEmbedding, src_pos: PositionalEncoding,
                 tgt_pos: PositionalEncoding, proj_layer: ProjectionLayer) -> None:
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.src_pos = src_pos
        self.tgt_pos = tgt_pos
        self.proj_layer = proj_layer

    def encode(self, src, src_mask):
        # (batch, seq_len, d_model)
        src = self.src_embed(src)
        src = self.src_pos(src)
        return self.encoder(src, src_mask)

    def decode(self, encoder_output, src_mask, tgt, tgt_mask):
        # (batch, seq_len, d_model)
        tgt = self.tgt_embed(tgt)
        tgt = self.tgt_pos(tgt)
        return self.decoder(tgt, encoder_output, src_mask, tgt_mask)

    def project(self, x):
        # (batch, seq_len, vocab_size)
        return self.proj_layer(x)
```

TensorFlow

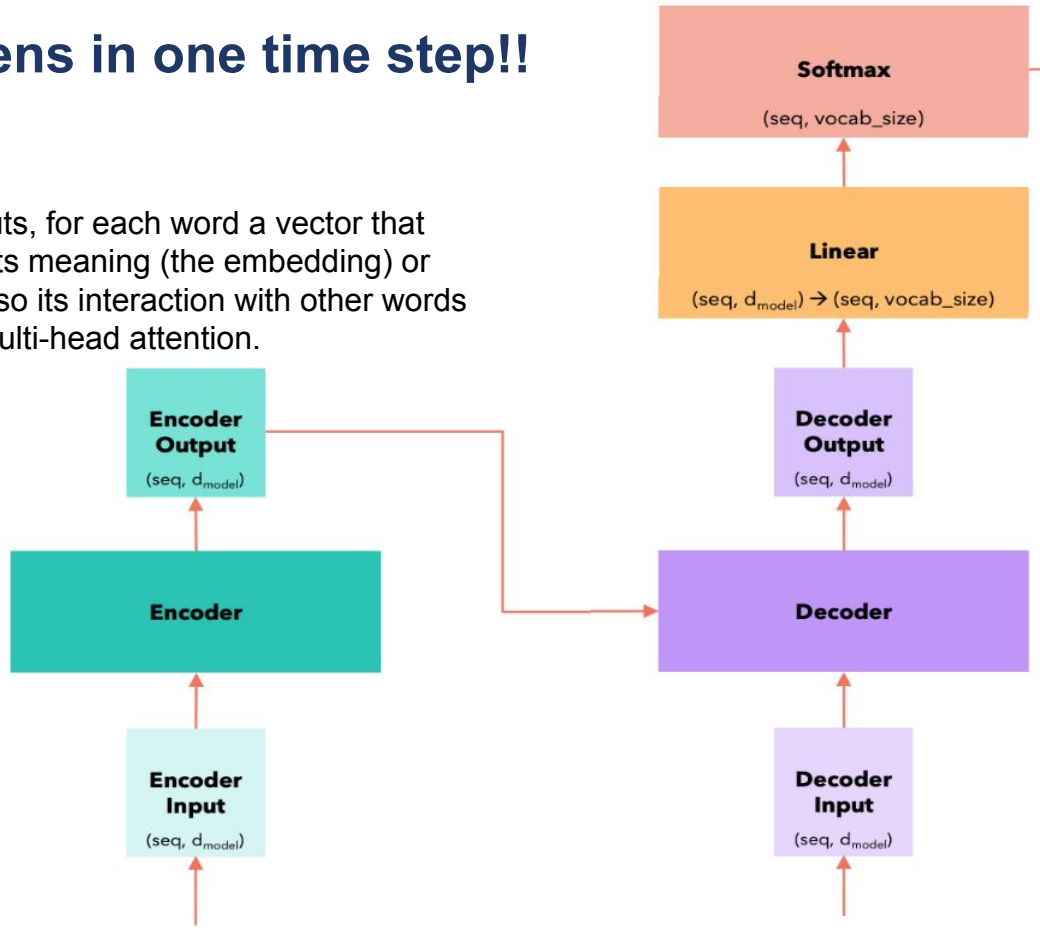
```python
class Transformer(tf.keras.Model):
  def __init__(self, *, num_layers, d_model, num_heads, dff,
               input_vocab_size, target_vocab_size, dropout_rate=0.1):
    super().__init__()
    self.encoder = Encoder(num_layers=num_layers, d_model=d_model,
                           num_heads=num_heads, dff=dff,
                           vocab_size=input_vocab_size,
                           dropout_rate=dropout_rate)

    self.decoder = Decoder(num_layers=num_layers, d_model=d_model,
                           num_heads=num_heads, dff=dff,
                           vocab_size=target_vocab_size,
                           dropout_rate=dropout_rate)

    self.final_layer = tf.keras.layers.Dense(target_vocab_size)

  def call(self, inputs):
    """
    Forward pass of the transformer model.
    """

    # To use a Keras model with `.fit` you must pass all your inputs in the
    # first argument.
    context, x  = inputs

    context = self.encoder(context)  # (batch_size, context_len, d_model)

    x = self.decoder(x, context)  # (batch_size, target_len, d_model)

    # Final linear layer output.
    logits = self.final_layer(x)  # (batch_size, target_len,
target_vocab_size)
    try:
      # Drop the keras mask, so it doesn't scale the losses/metrics.
      # b/250038731
      del logits._keras_mask
    except AttributeError:
      pass

    # Return the final output and the attention weights.
    return logits
```

UNIVERSITY OF TORONTO | CVDG

Training Walkthrough

It all happens in one time step!!

The encoder outputs, for each word a vector that not only captures its meaning (the embedding) or the position, but also its interaction with other words by means of the multi-head attention.



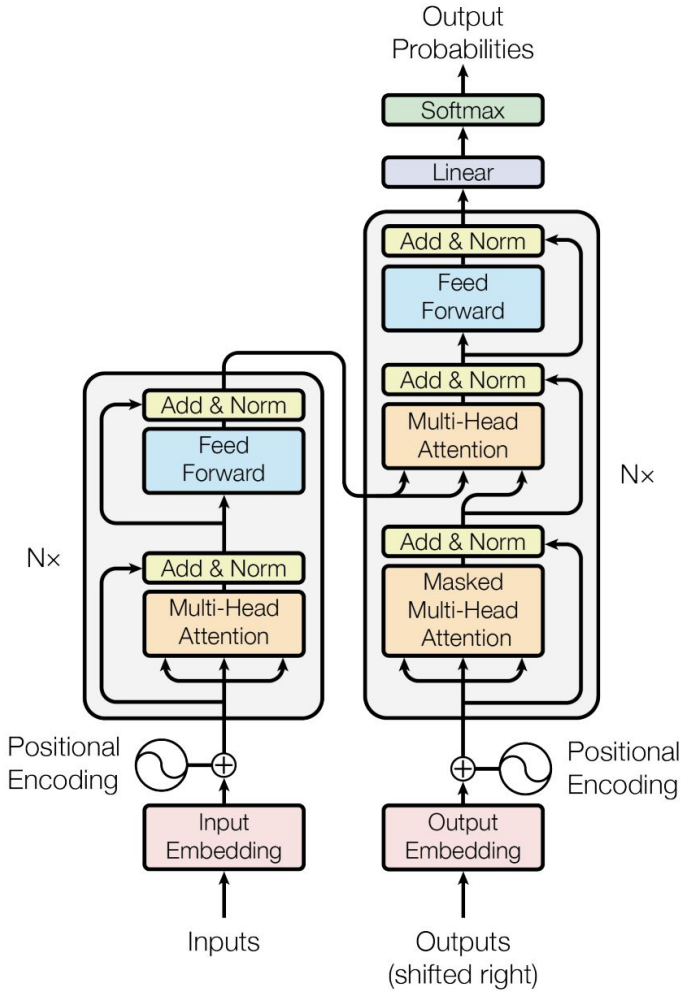
<SOS> I love you <EOS>

<SOS> Ich liebe dich

We prepend the <SOS> token at the beginning. That's why the paper says that the decoder input is shifted right.

Ich liebe dich <EOS>

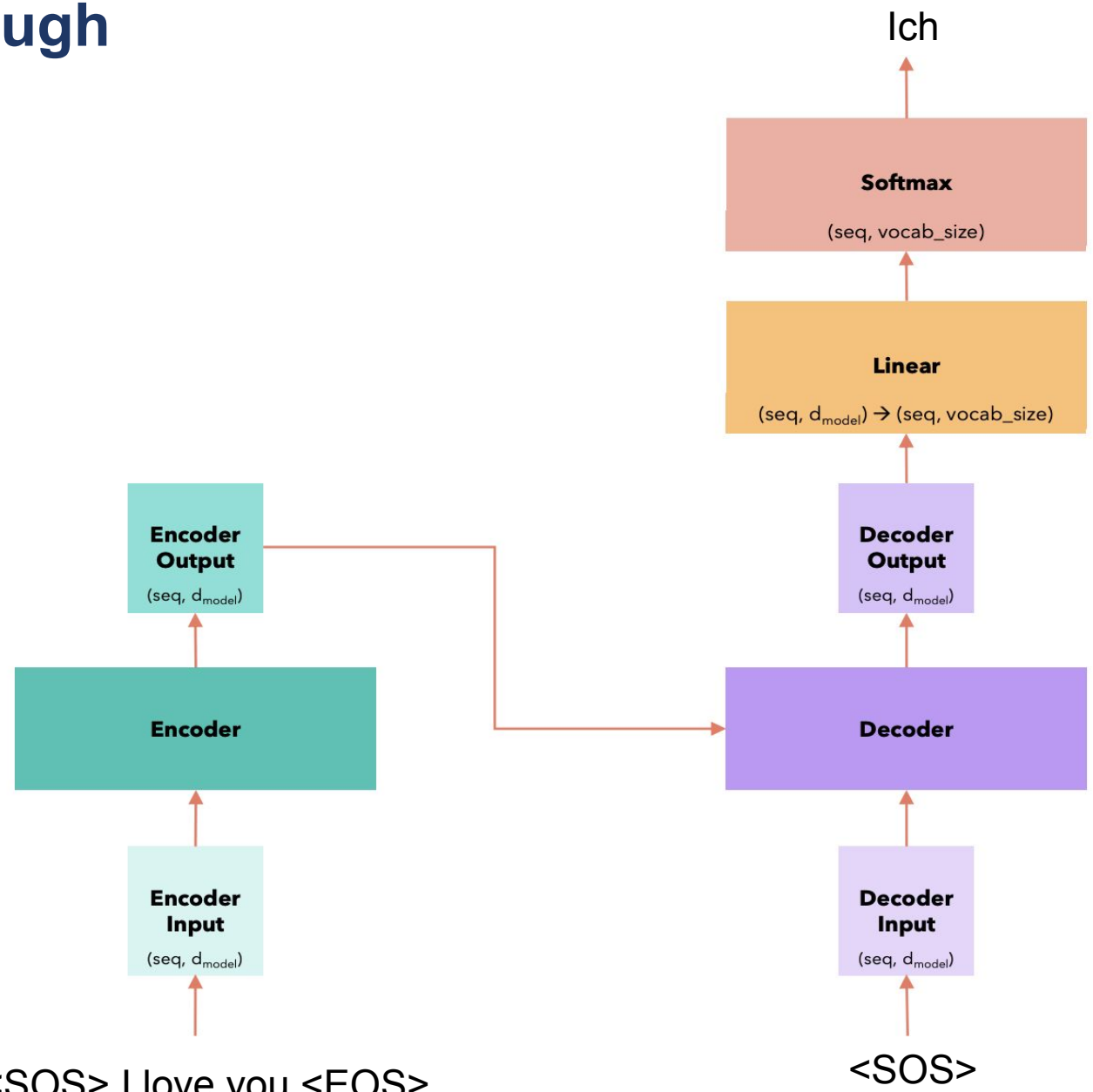
Cross Entropy Loss



Source: Umar Jamil's Transformers from Scratch

Inference Walkthrough

Time Step = 1

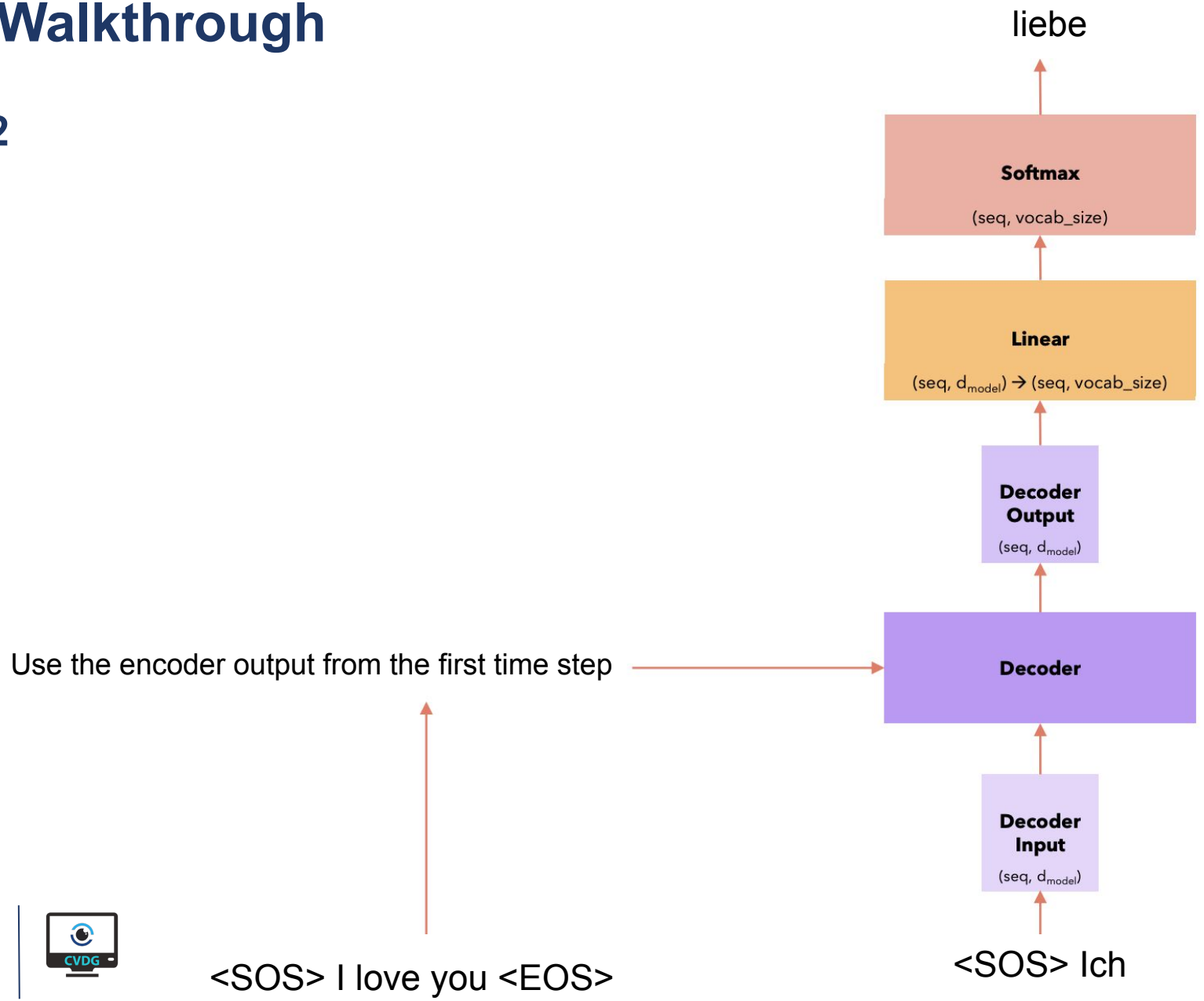


We select a token from the vocabulary corresponding to the position of the token with the maximum value.

Source: Umar Jamil's Transformers from Scratch

Inference Walkthrough

Time Step = 2

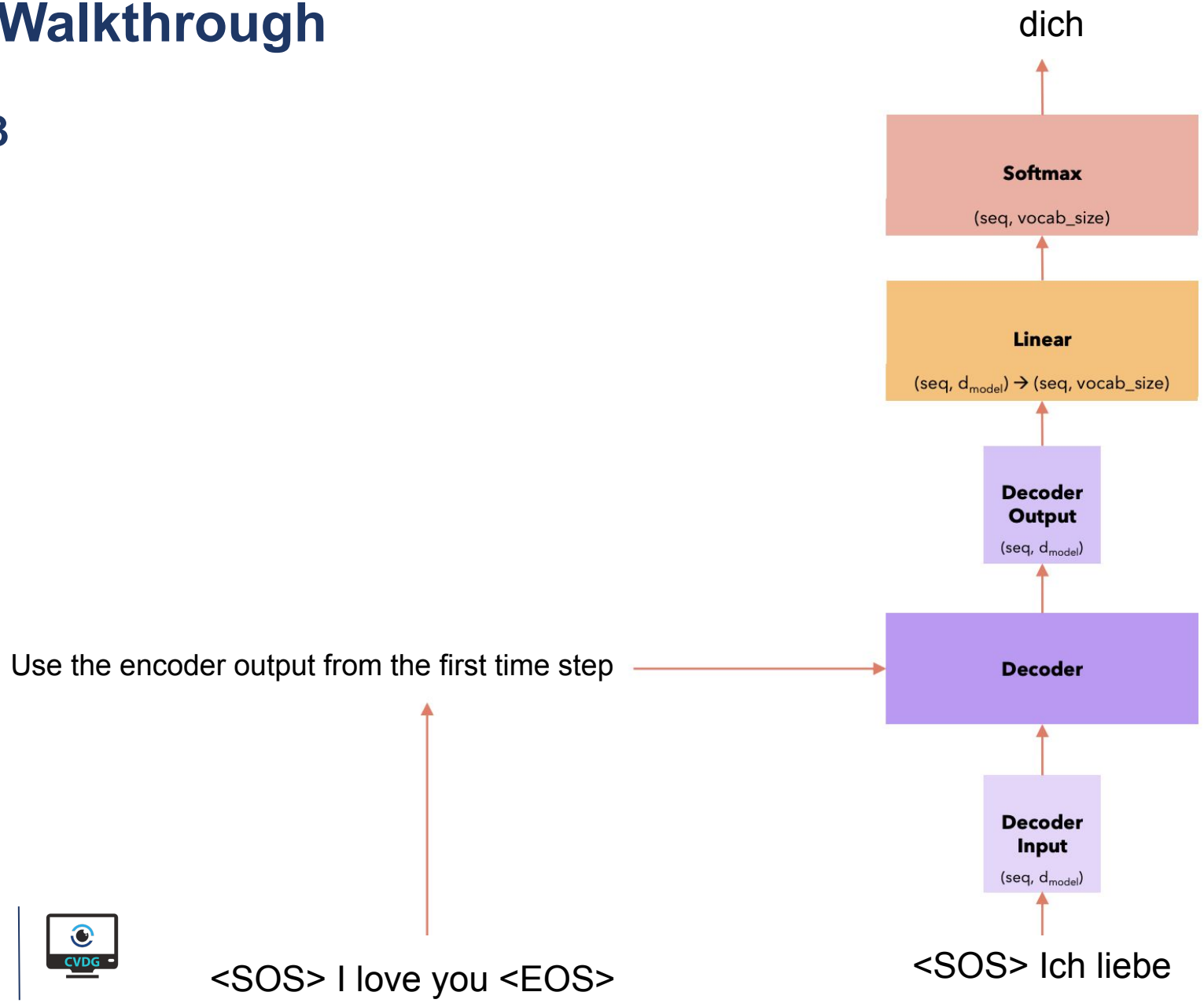


Since decoder input now contains two tokens, we select the softmax corresponding to the second token.

Source: Umar Jamil's Transformers from Scratch

Inference Walkthrough

Time Step = 3

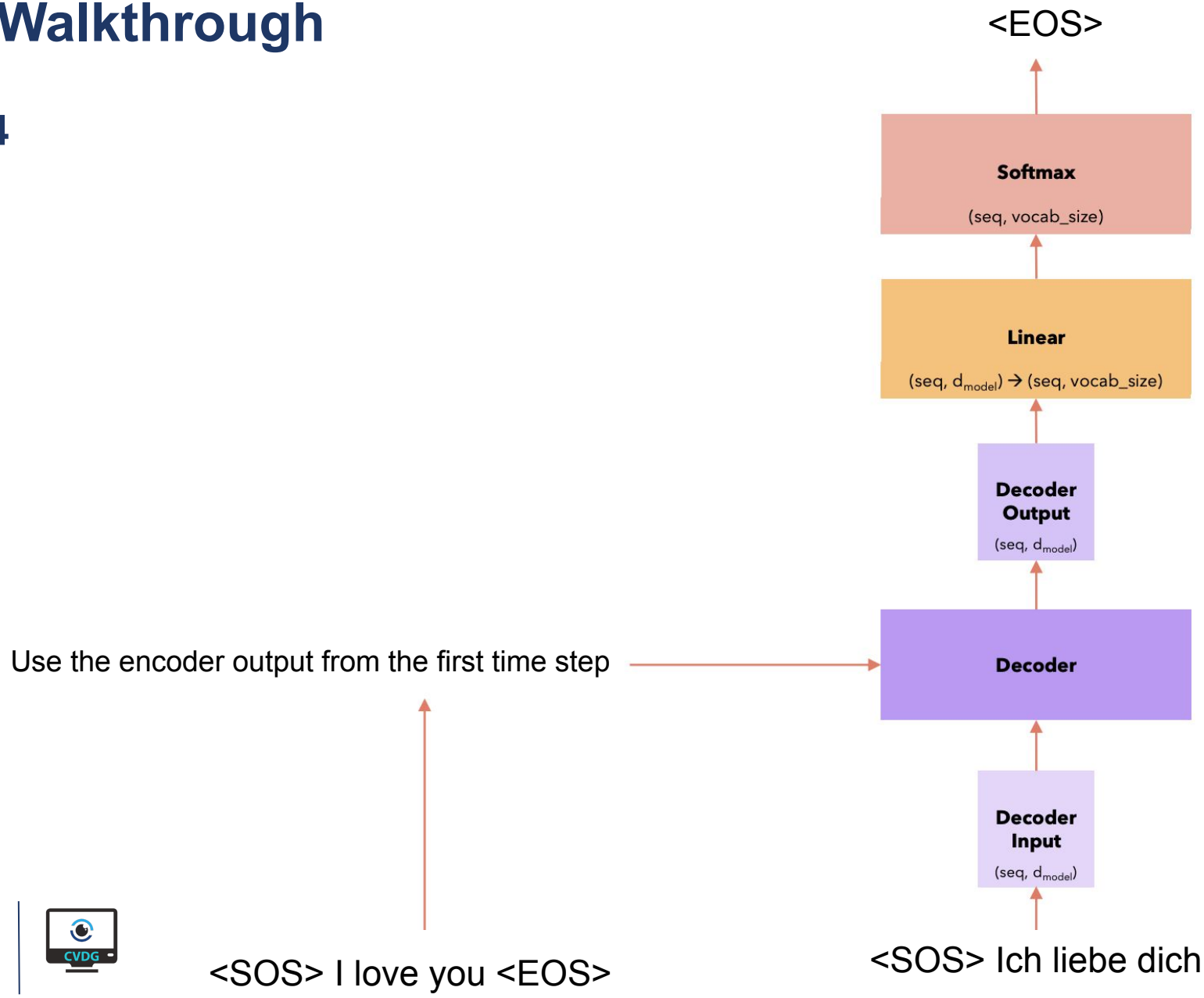


Since decoder input now contains three tokens, we select the softmax corresponding to the third token.

Source: Umar Jamil's Transformers from Scratch

Inference Walkthrough

Time Step = 4



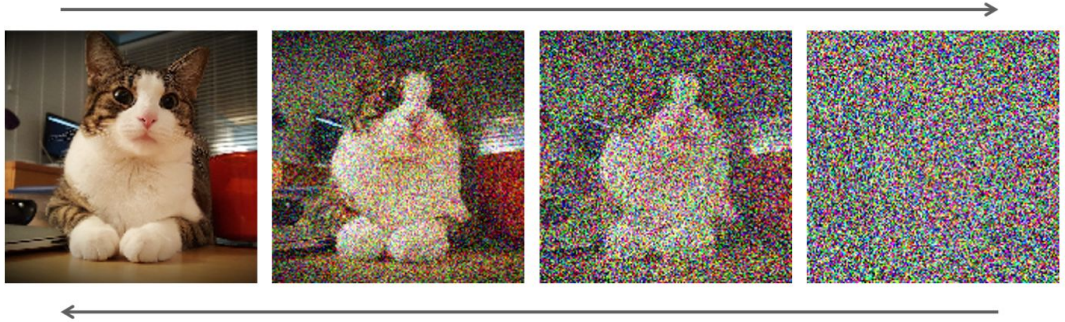
Since decoder input now contains four tokens, we select the softmax corresponding to the fourth token.

Source: Umar Jamil's Transformers from Scratch

Complete implementation on github.com/ut-cvdg/paper-breakdown

Thank You

Up Next: Diffusion Models



Follow us on social media

UNIVERSITY OF TORONTO

CVDG