Breaking down the Transformer

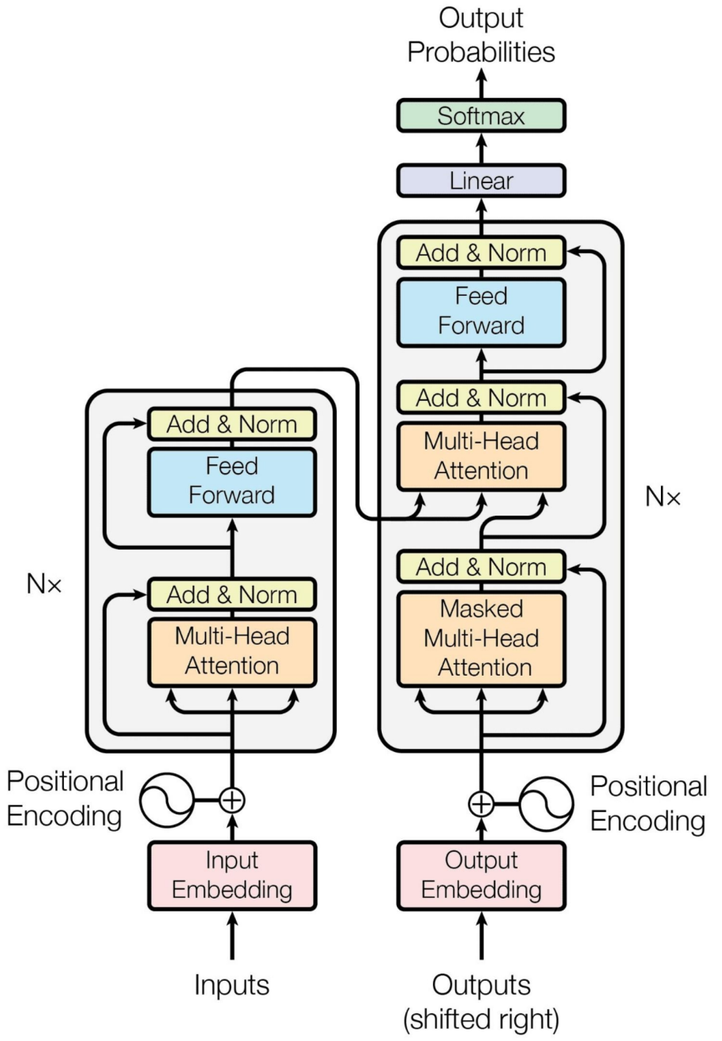
TRANSFORMER MODELS WITH PYTORCH

James Chapman
Curriculum Manager, DataCamp

datacamp

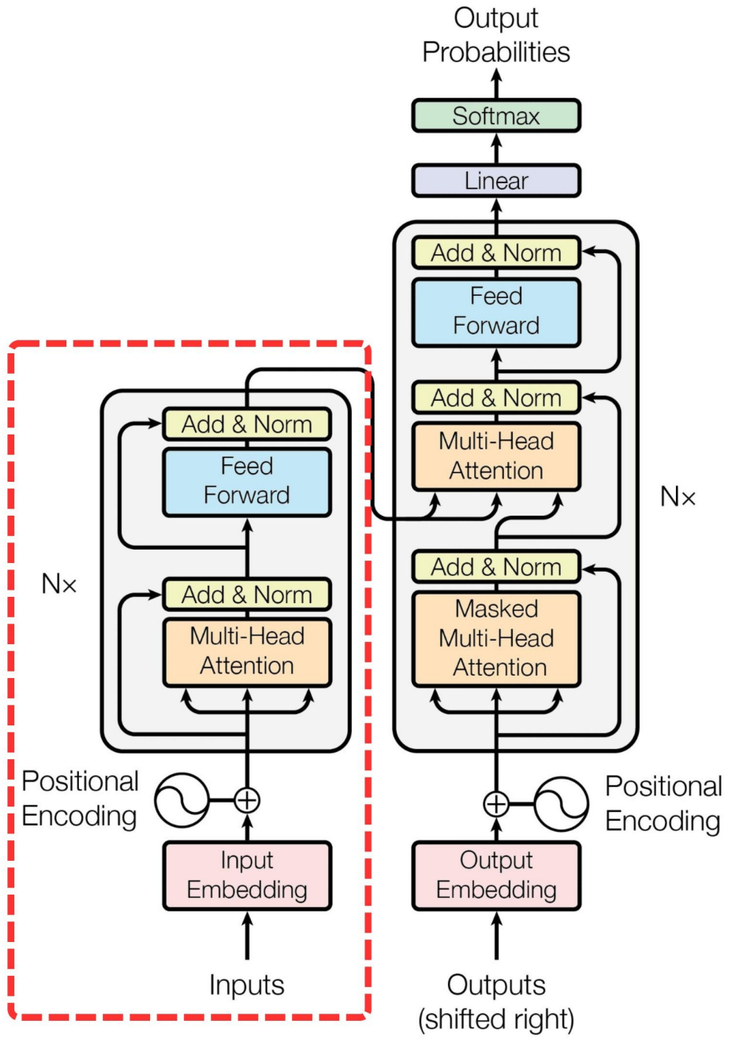
The paper that changed everything...

- **Attention Is All You Need** by Ashish Vaswani et al. (arXiv:1706.03762)
  - Attention mechanisms
  - Optimized for text modeling
  - Used in *Large Language Models (LLMs)*



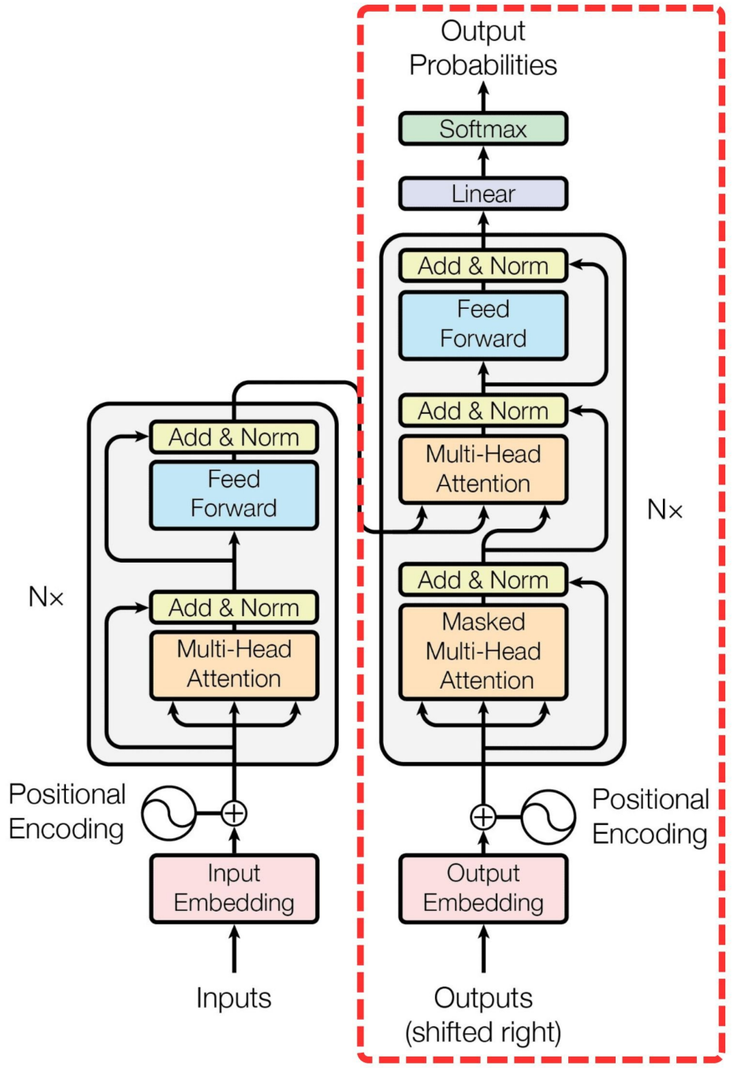
The paper that changed everything...

- **Attention Is All You Need** by Ashish Vaswani et al.
  - Attention mechanisms
  - Optimized for text modeling
  - Used in *Large Language Models (LLMs)*



The paper that changed everything...

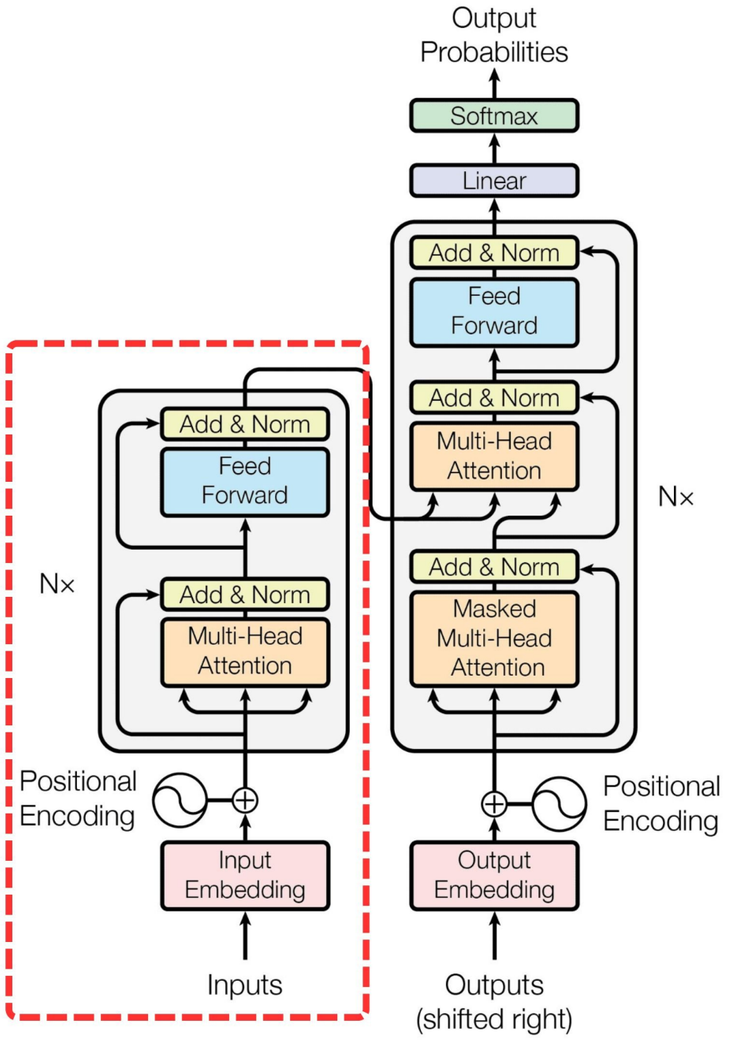
- **Attention Is All You Need** by Ashish Vaswani et al.
  - Attention mechanisms
  - Optimized for text modeling
  - Used in *Large Language Models (LLMs)*



Unpacking the Transformer...

Encoder block

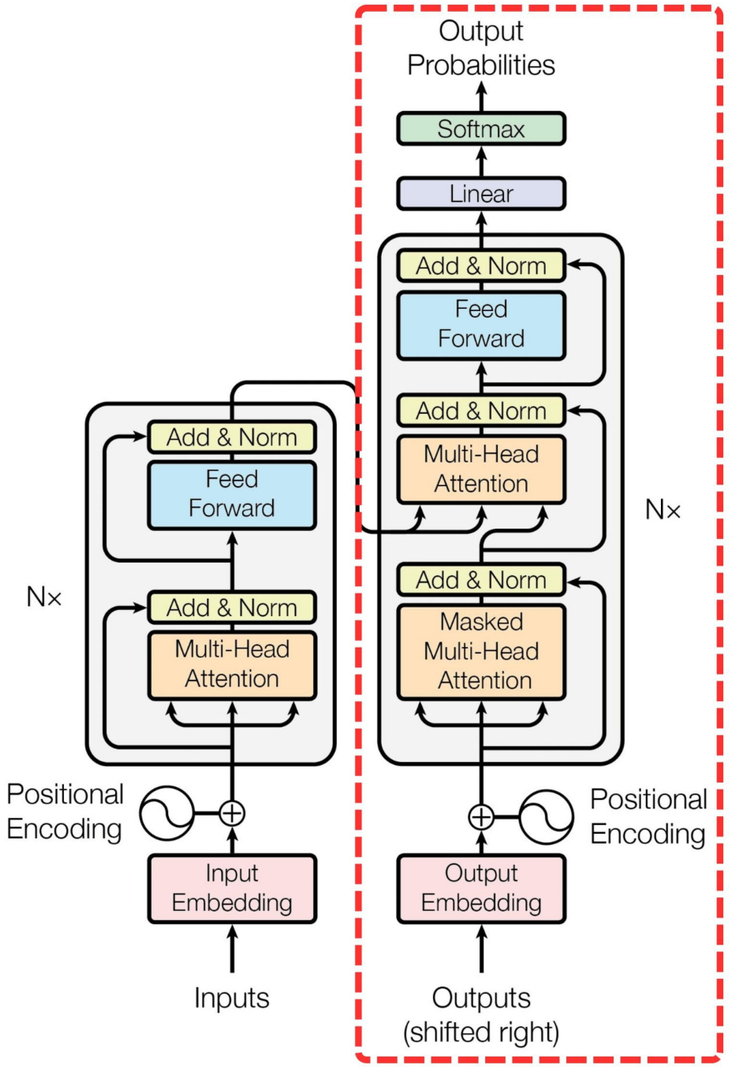
- Multiple identical layers
- **Reading** and **processing** input sequence
- Generate *context-rich* numerical representations
- Uses **self-attention** and **feed-forward networks**



Unpacking the Transformer...

Decoder block

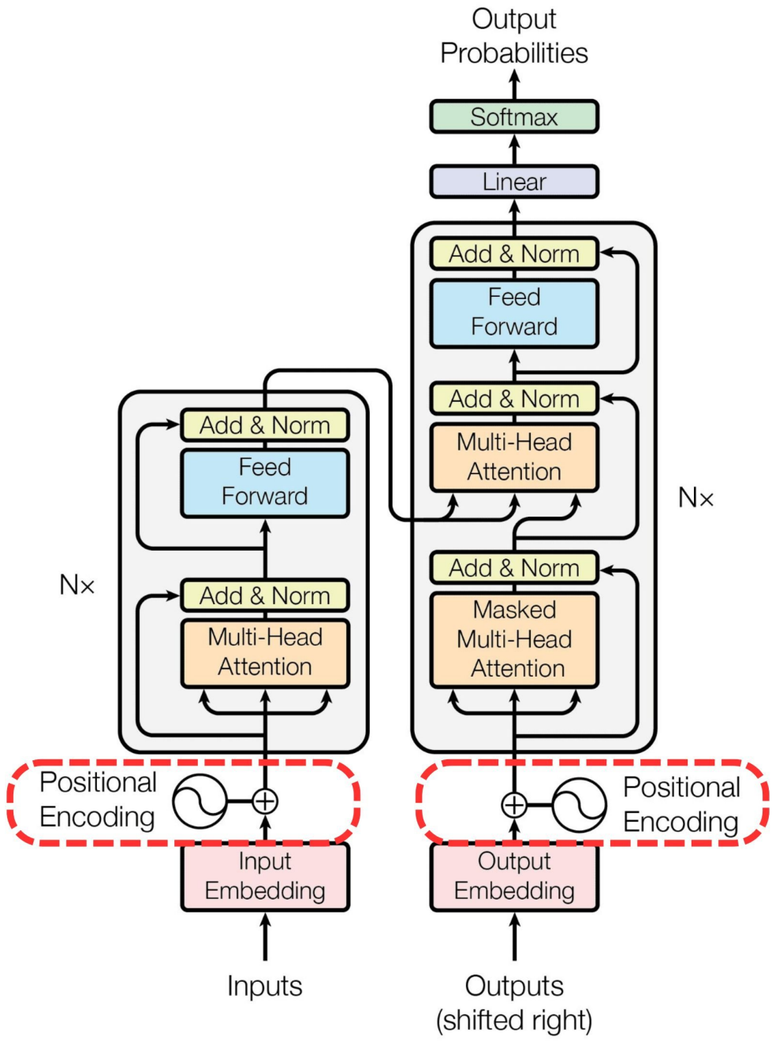
- Encoded input sequence → output sequence



Unpacking the Transformer...

Positional encoding

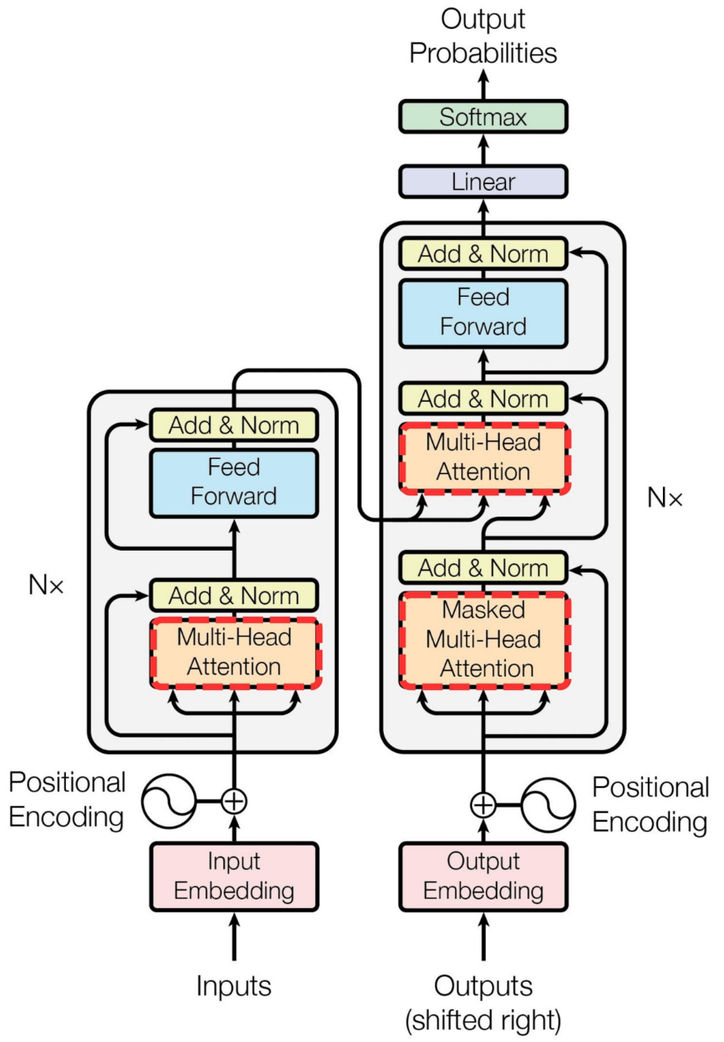
- Encode each token's *position* in the sequence
- Order is crucial for modeling sequences



Unpacking the Transformer...

Attention mechanisms

- Focus on important tokens and their relationships
- Improve text generation



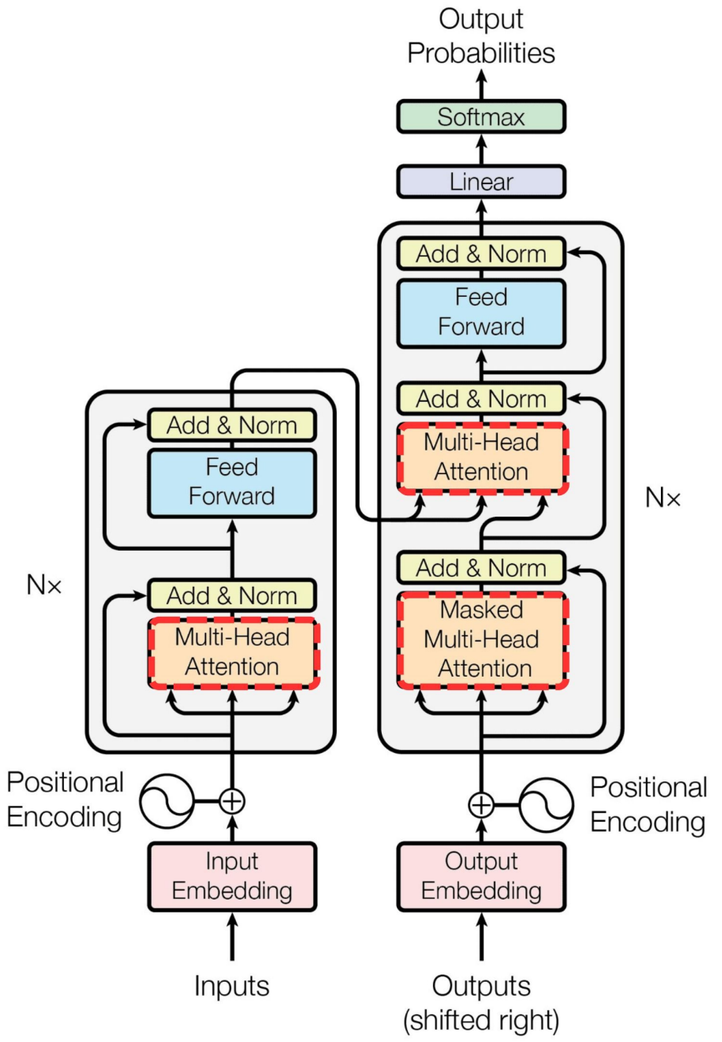
Unpacking the Transformer...

Attention mechanisms

- Focus on important tokens and their relationships
- Improve text generation

Self-attention

- *Weighting* token importance
- Captures long-range dependencies



Unpacking the Transformer...

Attention mechanisms

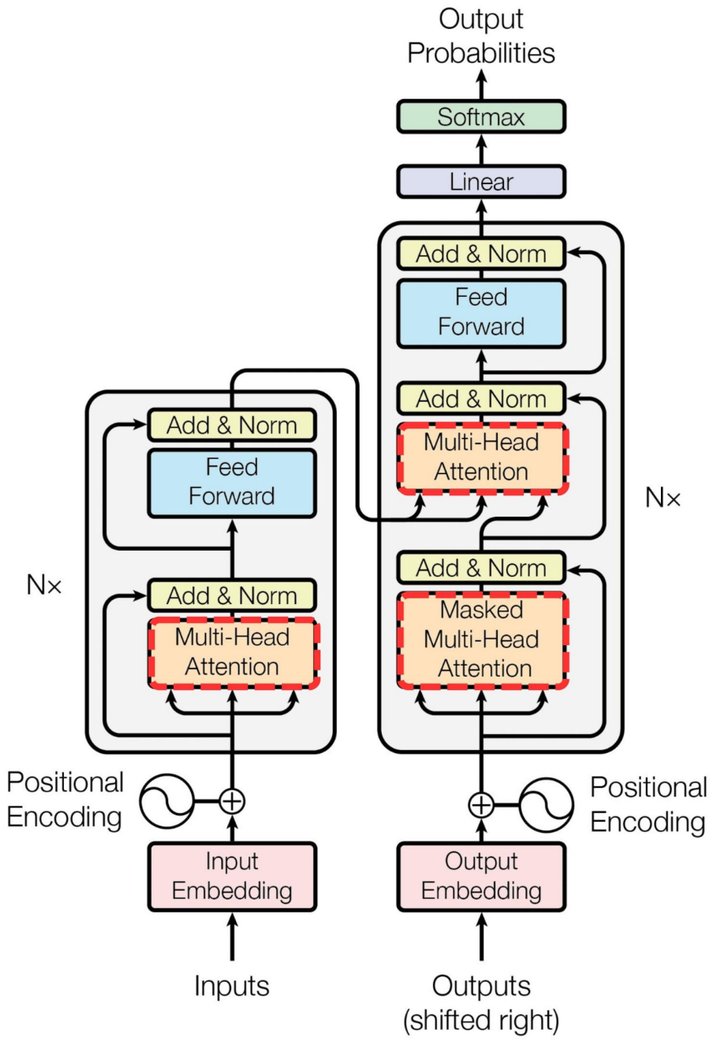
- Focus on important tokens and their relationships
- Improve text generation

Self-attention

- *Weighting* token importance
- Captures long-range dependencies

Multi-head attention

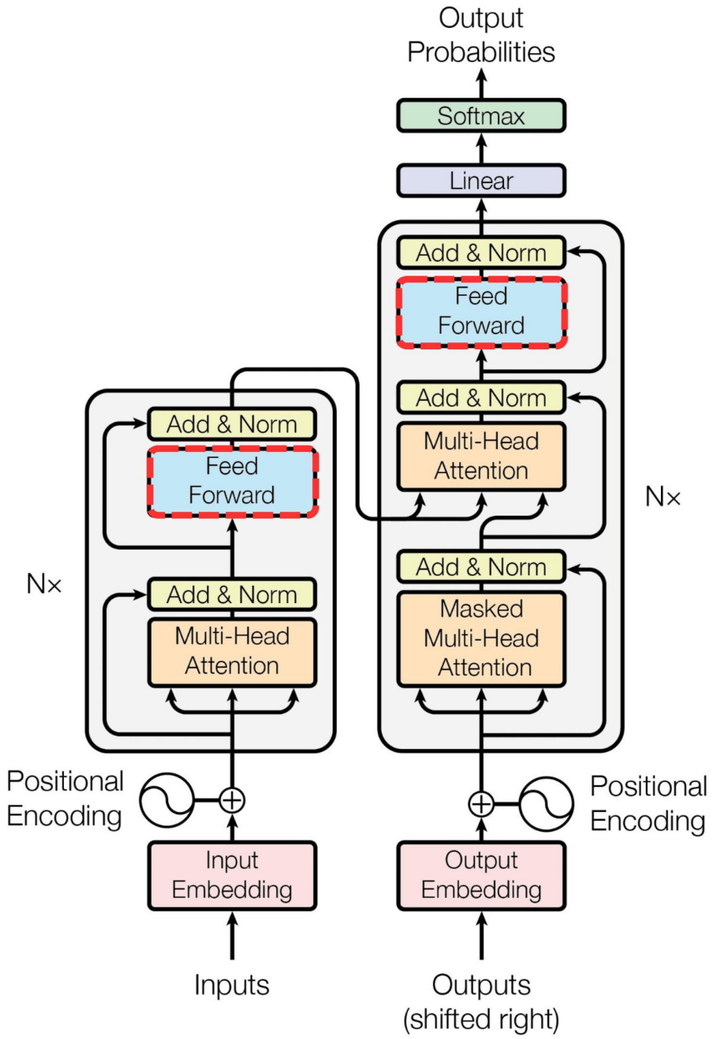
- Splits input into multiple **heads**
- Heads capture distinct patterns, leading to richer representations



Unpacking the Transformer...

Position-wise feed-forward networks

- Simple NNs that apply transformations
- Each token is transformed *independently*
- Position-independent/"position-wise"



Transformers in PyTorch

- `d_model` : Dimensionality of model inputs
- `nheads` : Number of attention heads
- `num_encoder_layers` : Number of encoder layers
- `num_decoder_layers` : Number of decoder layers

```
import torch.nn as nn

model = nn.Transformer(
    d_model=512,
    nhead=8,
    num_encoder_layers=6,
    num_decoder_layers=6
)

print(model)
```

```
Transformer(
  (encoder): TransformerEncoder(
    (layers): ModuleList(
      (0-5): 6 x TransformerEncoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
        )
        (linear1): Linear(in_features=512, out_features=2048, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=2048, out_features=512, bias=True)
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
      )
    )
    (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  )
  ...
```

```
  (decoder): TransformerDecoder(
    (layers): ModuleList(
      (0-5): 6 x TransformerDecoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
        )
        (multihead_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
        )
        (linear1): Linear(in_features=512, out_features=2048, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=2048, out_features=512, bias=True)
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (norm3): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
        (dropout3): Dropout(p=0.1, inplace=False)
      )
    )
    (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
  )
)
```

Let's practice!

TRANSFORMER MODELS WITH PYTORCH

Embedding and positional encoding

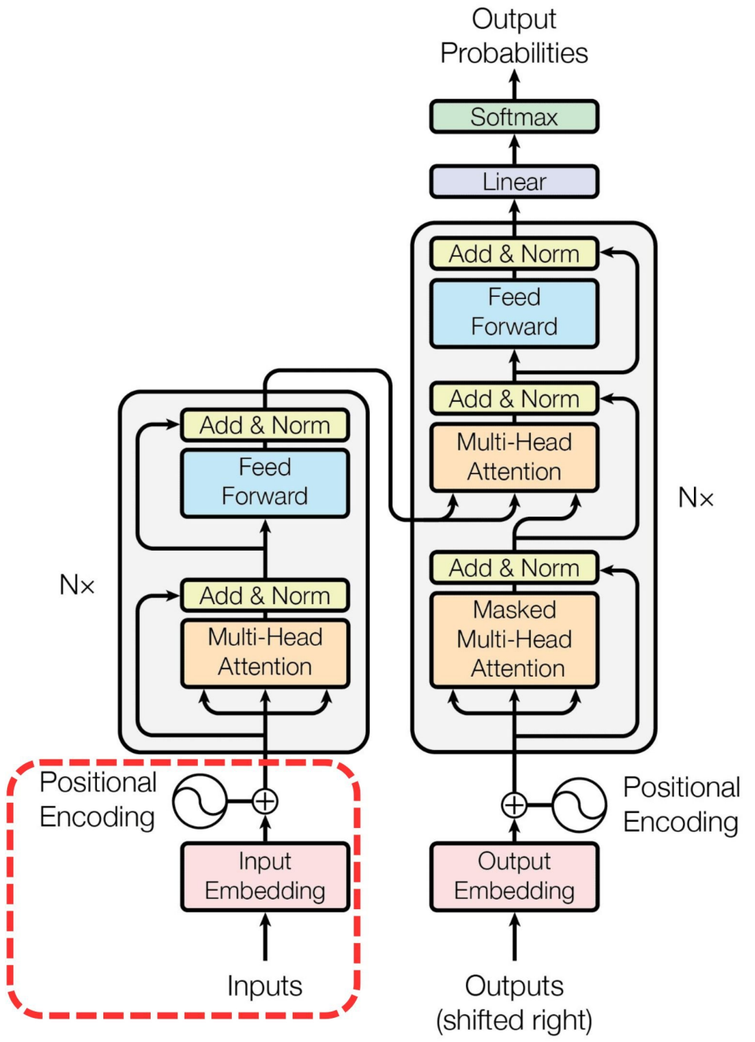
TRANSFORMER MODELS WITH PYTORCH

James Chapman
Curriculum Manager, DataCamp

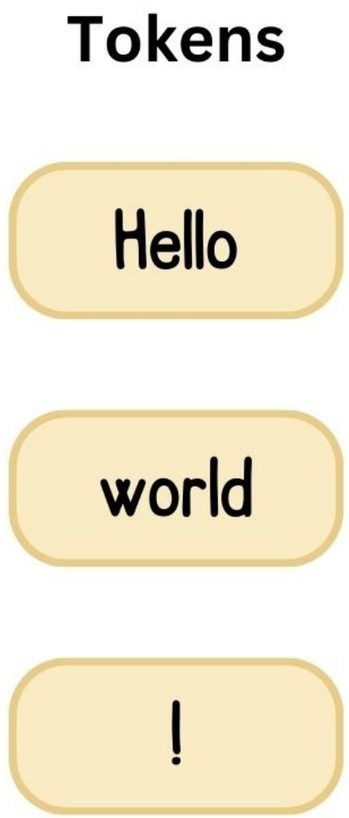
datacamp

Embedding and positional encoding in transformers

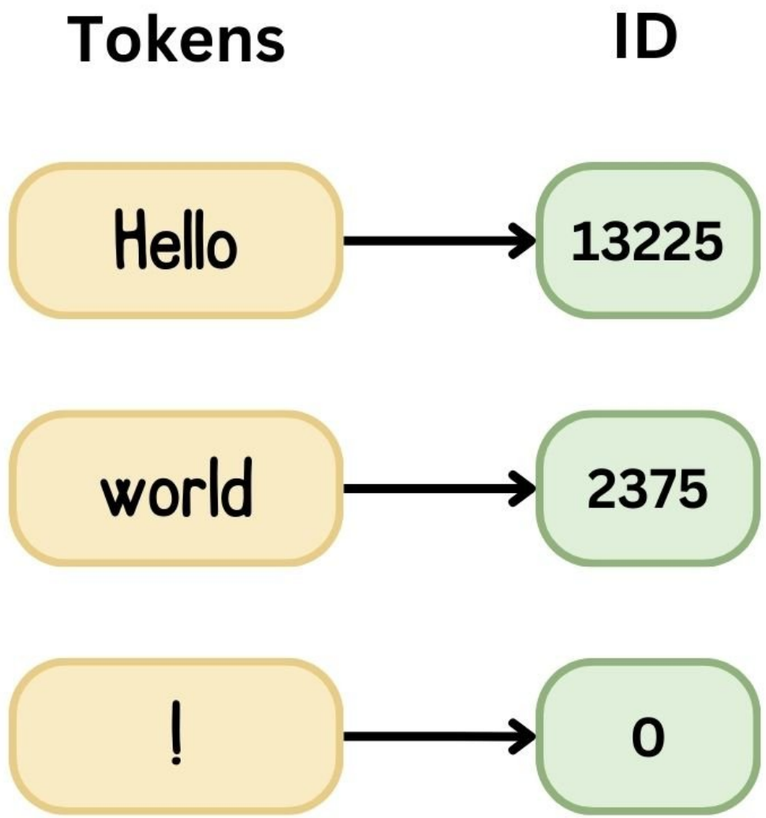
- **Embedding:** tokens → embedding vector
- **Positional encoding:** Token position + embedding vector → positional encoding



Embedding sequences



Embedding sequences



Embedding sequences

```python
import torch
import math
import torch.nn as nn

class InputEmbeddings(nn.Module):
    def __init__(self, vocab_size: int, d_model: int) -> None:
        super().__init__()
        self.d_model = d_model
        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)
```

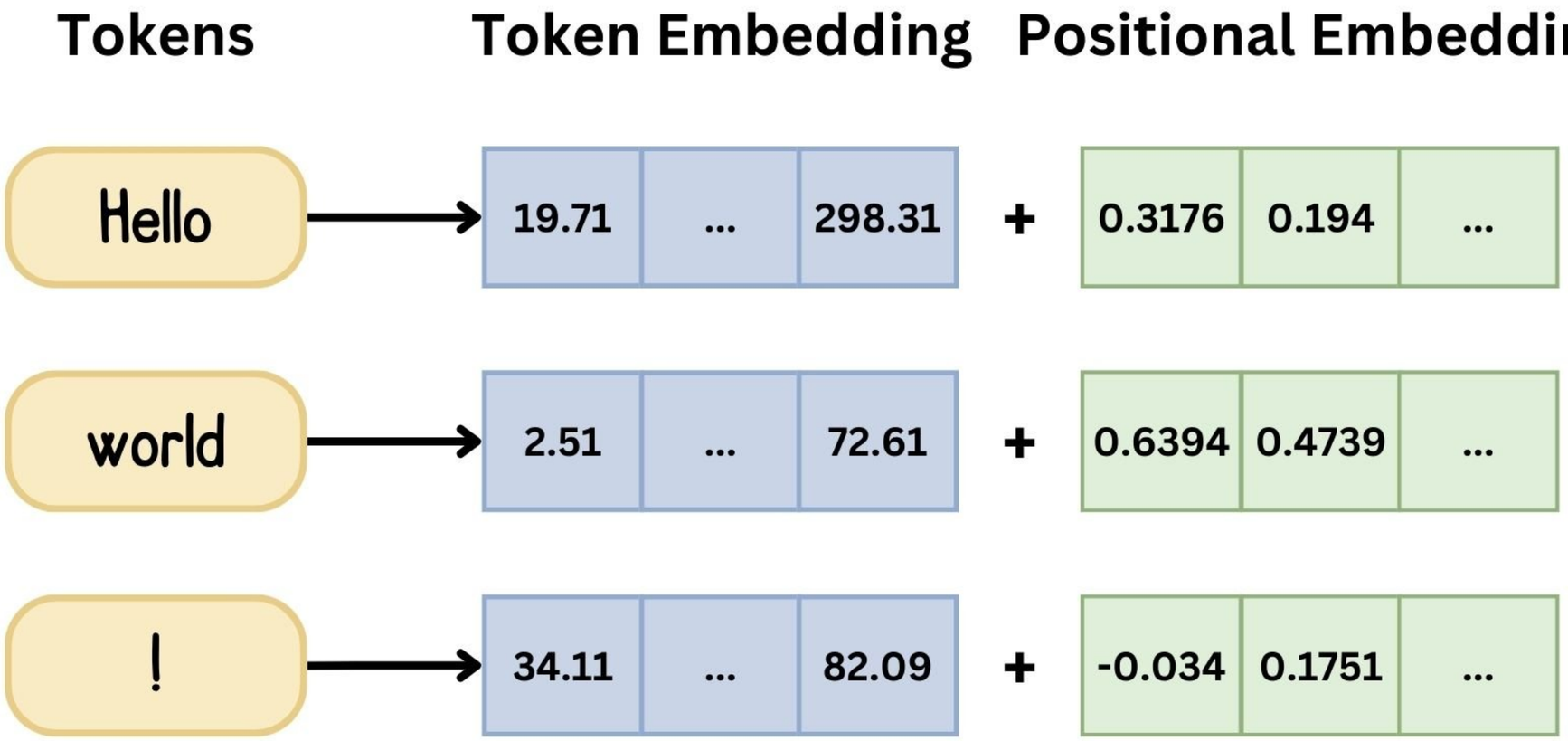
- **Standard Practice:** scaling by $\sqrt{d_{model}}$

Creating embeddings

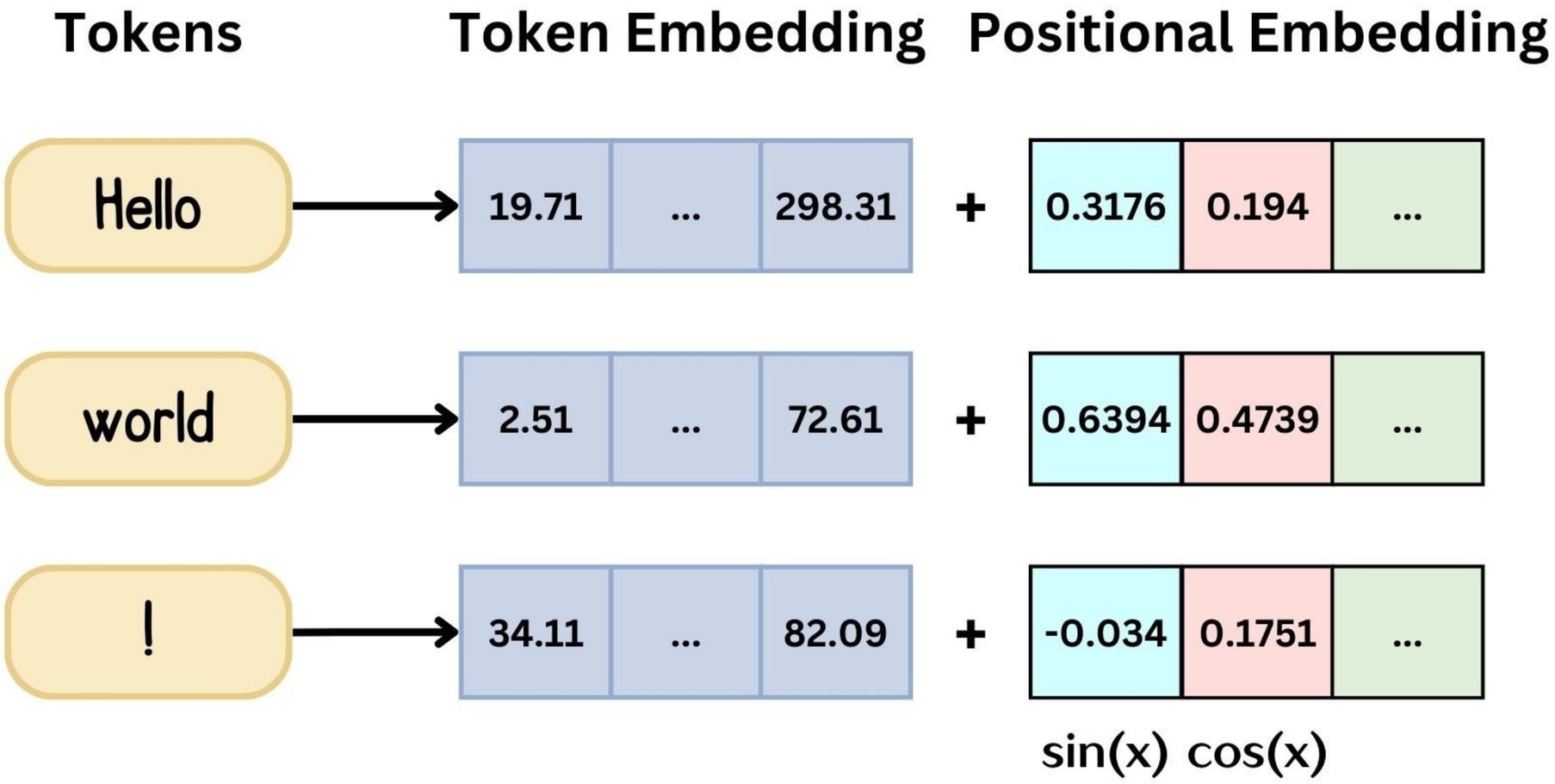
```
embedding_layer = InputEmbeddings(vocab_size=10_000, d_model=512)
embedded_output = embedding_layer(torch.tensor([[1, 2, 3, 4], [5, 6, 7, 8]]))
print(embedded_output.shape)
```

```
torch.Size([2, 4, 512])
```

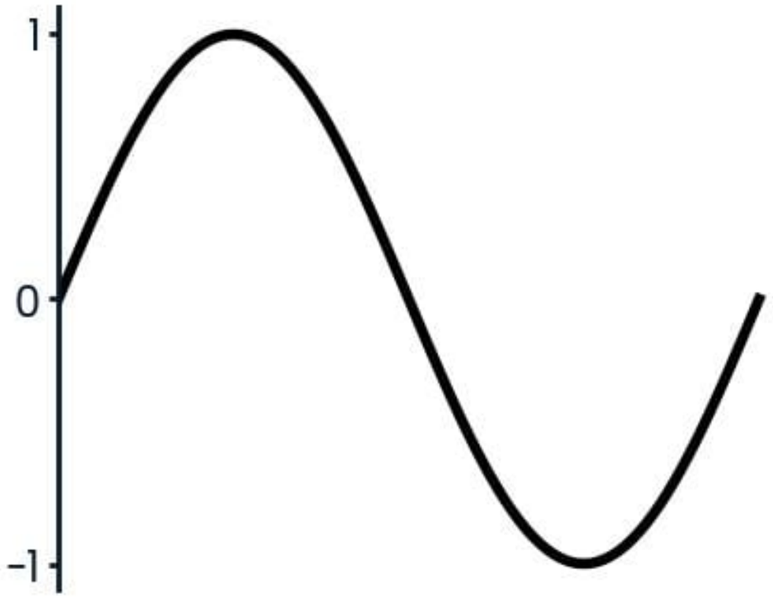
Positional encoding



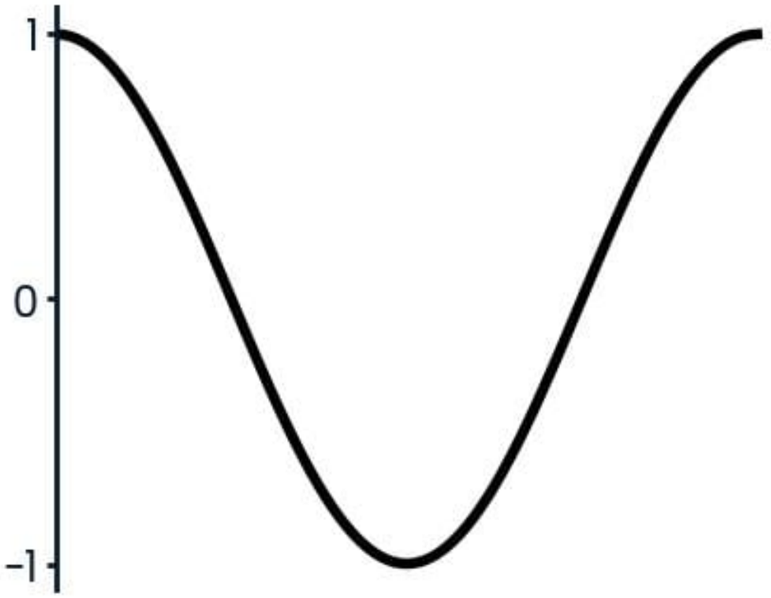
Positional encoding



sin(x)



cos(x)



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

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

Building a positional encoder

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_length):
        super().__init__()

        pe = torch.zeros(max_seq_length, d_model)
        position = torch.arange(0, max_seq_length, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2, dtype=torch.float) * -(math.log(10000.0) / d_model))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]
```

Creating positional encodings

```
pos_encoding_layer = PositionalEncoding(d_model=512, max_seq_length=4)
pos_encoded_output = pos_encoding_layer(embedded_output)
print(pos_encoded_output.shape)
```

```
torch.Size([2, 4, 512])
```

Let's practice!

TRANSFORMER MODELS WITH PYTORCH

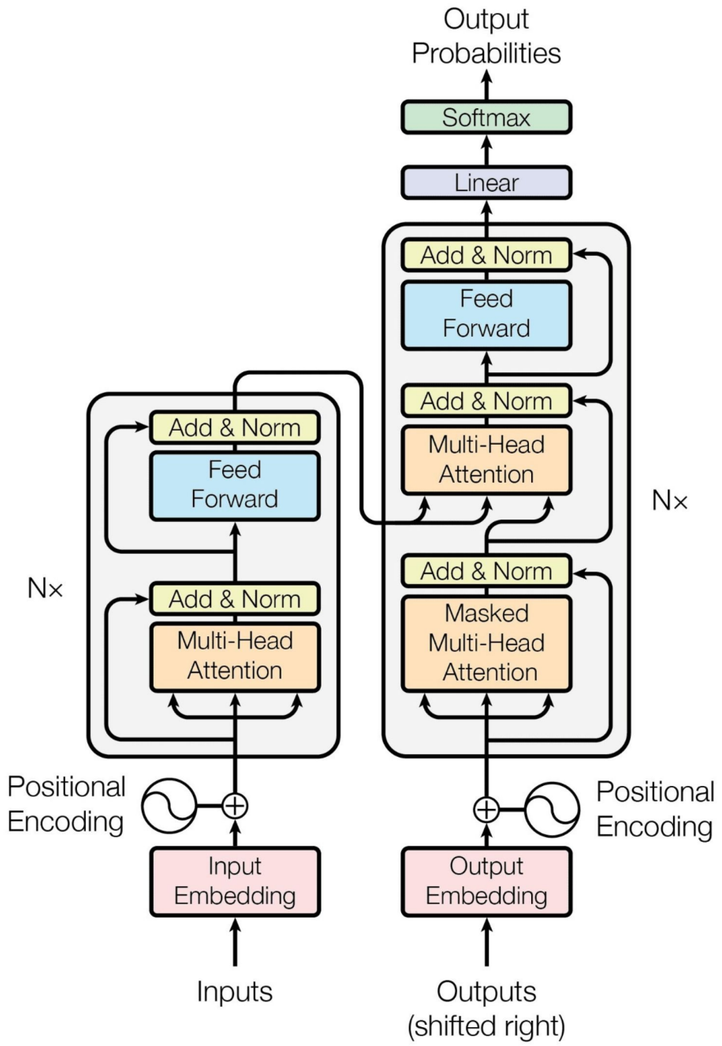
Multi-head self-attention

TRANSFORMER MODELS WITH PYTORCH

James Chapman
Curriculum Manager, DataCamp

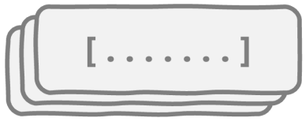
datacamp

Multi-head attention in transformers

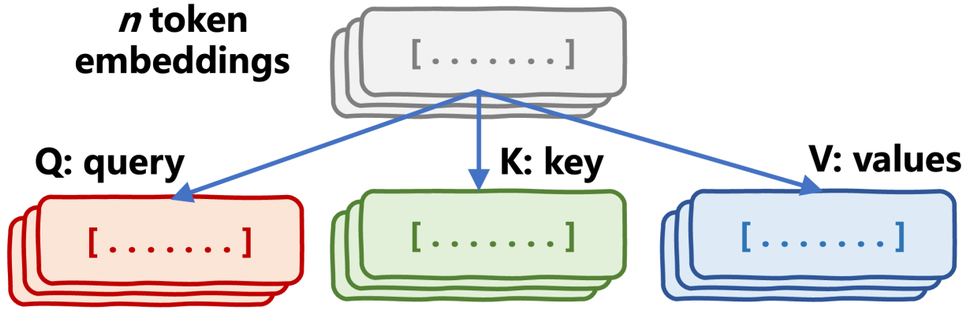


Self-attention mechanism

***n* token embeddings**

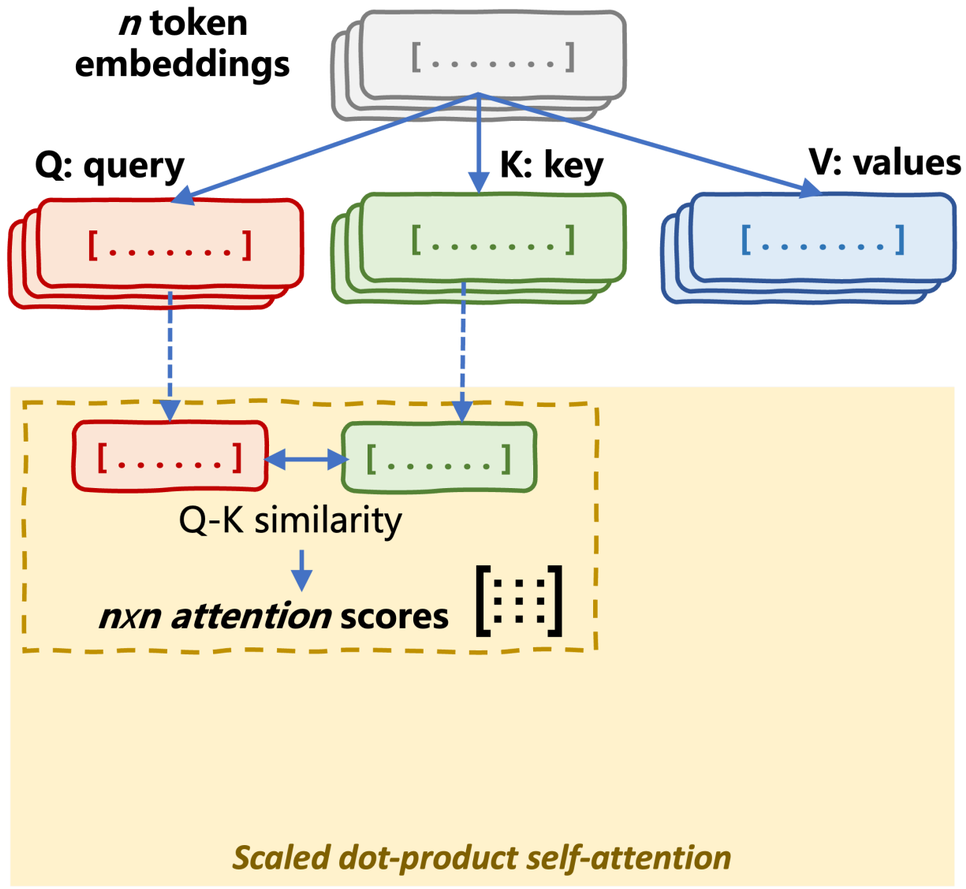


Self-attention mechanism



- **Q**: Indicate what each token is "looking for" in other tokens
- **K**: Represent content of each token
- **V**: Actual content to be aggregated or weighted

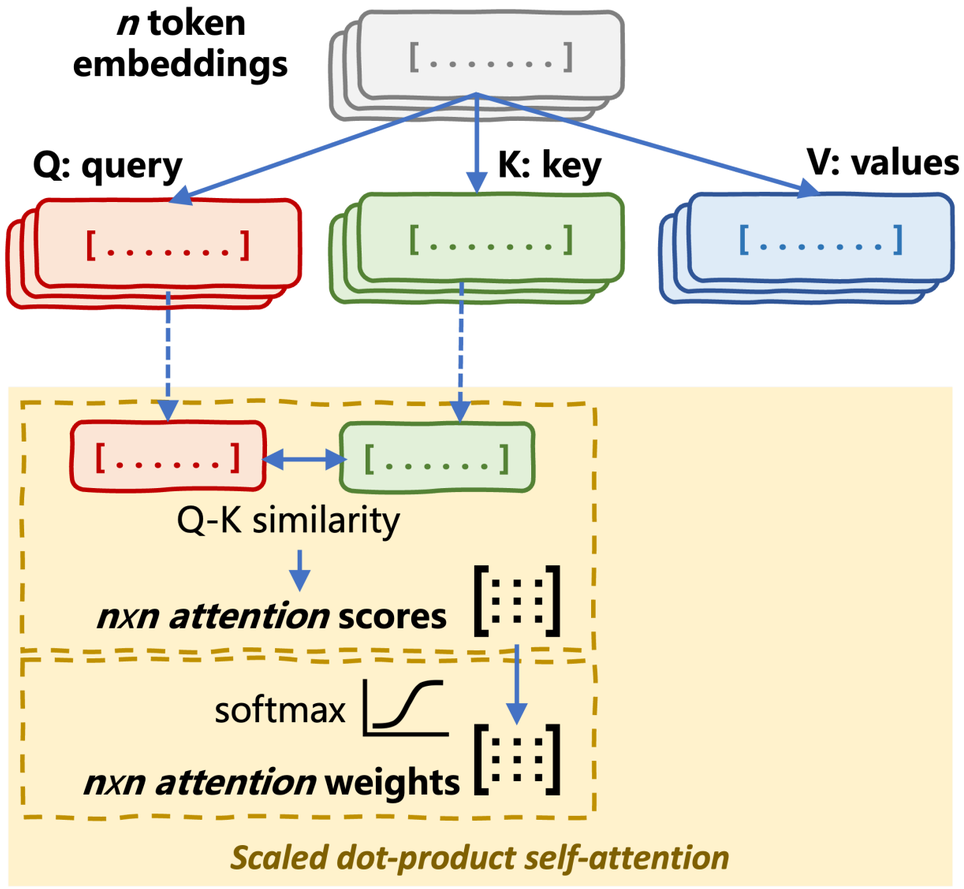
Self-attention mechanism



- **Q**: Indicate what each token is "looking for" in other tokens
- **K**: Represent content of each token
- **V**: Actual content to be aggregated or weighted

- **Attention Scores**: Q-K similarity → *dot-product*

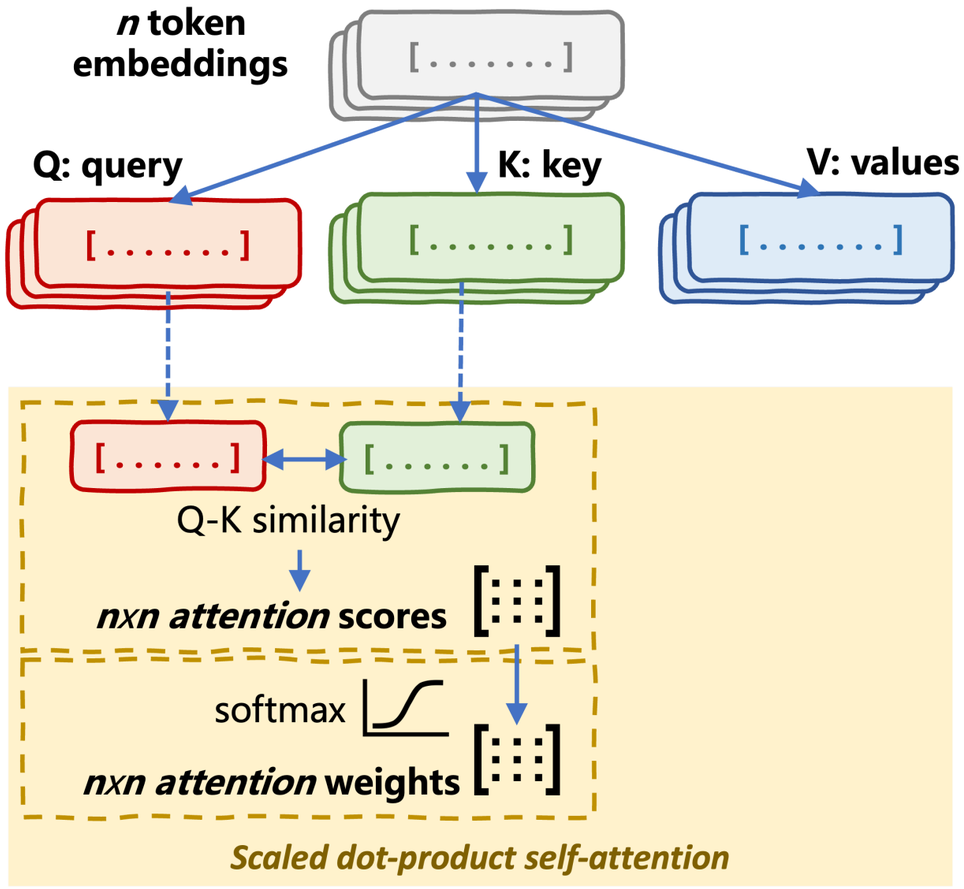
Self-attention mechanism



- **Q**: Indicate what each token is "looking for" in other tokens
- **K**: Represent content of each token
- **V**: Actual content to be aggregated or weighted

- **Attention Scores**: Q-K similarity → dot-product
- **Attention Weights**: *softmax* scaling

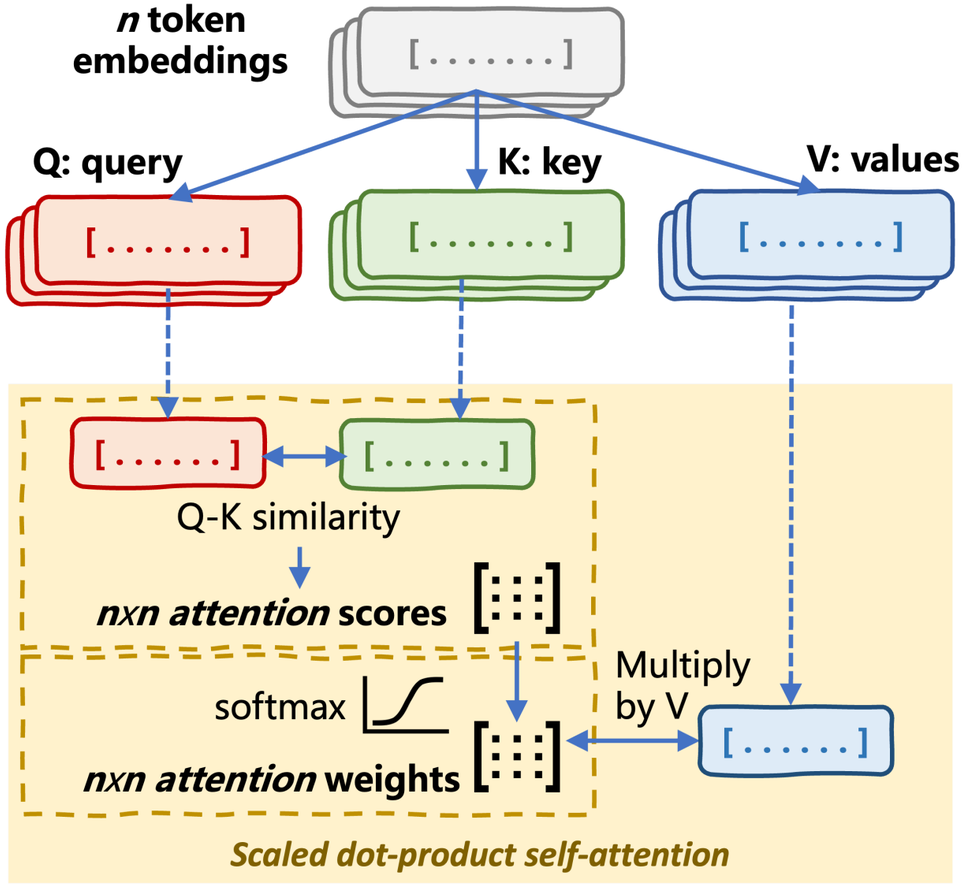
Self-attention mechanism



- **Attention Scores:** Q-K similarity → dot-product
- **Attention Weights:** *softmax* scaling

	Orange	is	my	favorite	fruit
Query:	Orange				
Attention weights:	.21	.03	.05	**.31**	**.40**

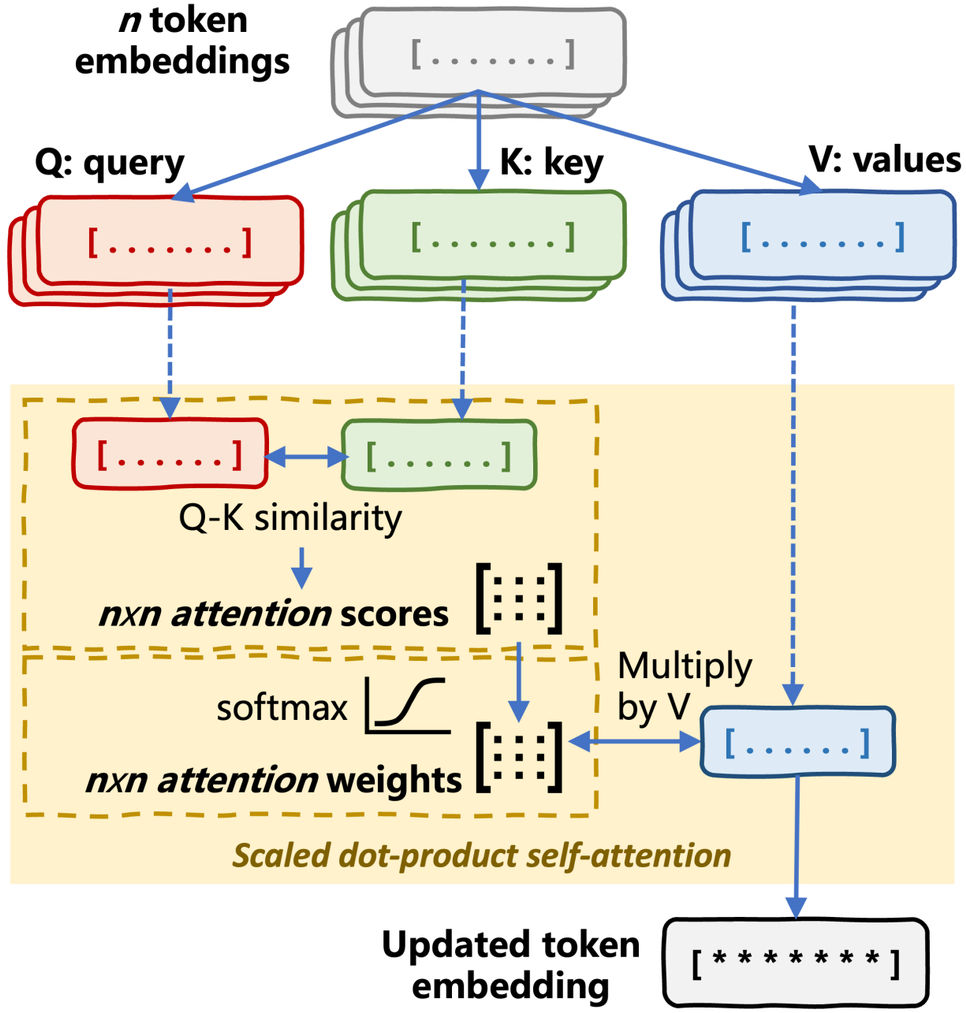
Self-attention mechanism



- **Attention Scores:** Q-K similarity → dot-product
- **Attention Weights:** *softmax* scaling

	Orange	is	my	favorite	fruit
Query:	Orange				
Attention weights:	.21	.03	.05	**.31**	**.40**

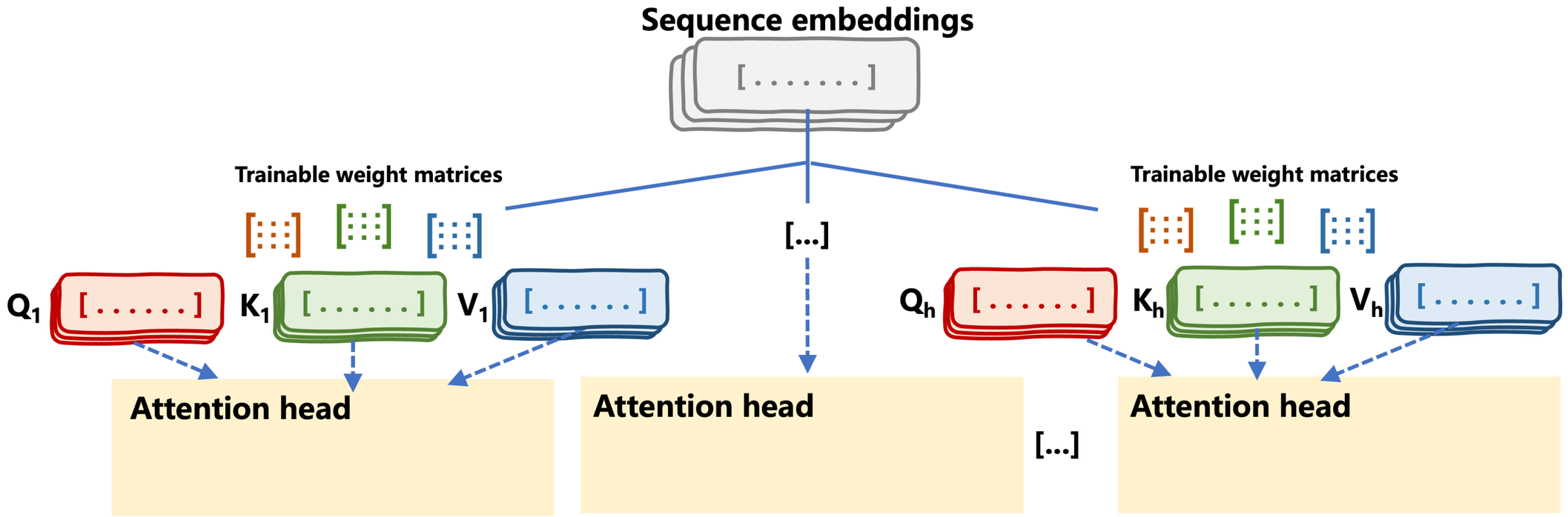
Self-attention mechanism



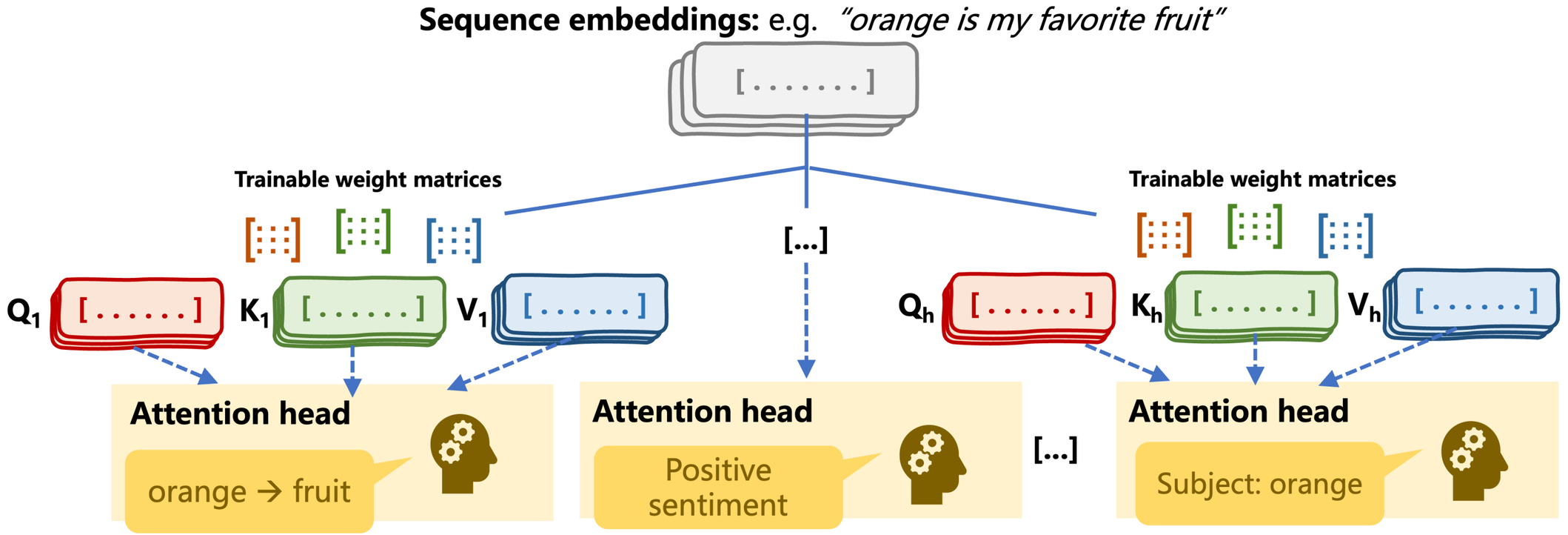
- **Attention Scores:** Q-K similarity → dot-product
- **Attention Weights:** *softmax* scaling

	Orange	is	my	favorite	fruit
Query:	Orange				
Attention weights:	.21	.03	.05	**.31**	**.40**

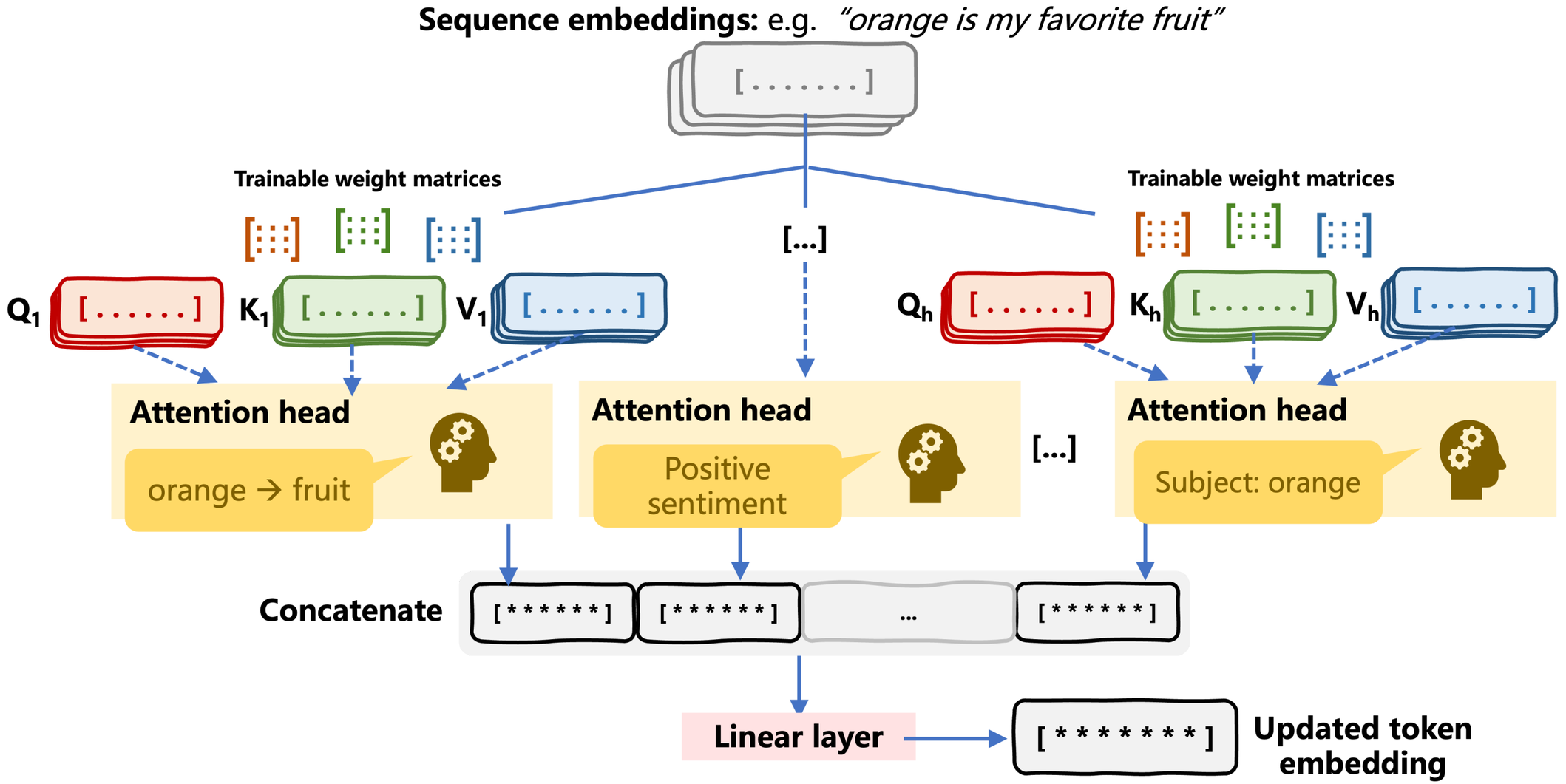
Multi-head attention



Multi-head attention



Multi-head attention



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

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads."
        self.num_heads = num_heads
        self.d_model = d_model
        self.head_dim = d_model // num_heads
        self.query_linear = nn.Linear(d_model, d_model, bias=False)
        self.key_linear = nn.Linear(d_model, d_model, bias=False)
        self.value_linear = nn.Linear(d_model, d_model, bias=False)
        self.output_linear = nn.Linear(d_model, d_model)
```

- `num_heads` : no. of attention heads, each handling embeddings of size `head_dim`
- `bias=False` : no impact on performance while reducing complexity (**only for inputs**)

```
    def split_heads(self, x, batch_size):
        seq_length = x.size(1)
        x = x.reshape(batch_size, seq_length, self.num_heads, self.head_dim)
        return x.permute(0, 2, 1, 3)

    def compute_attention(self, query, key, value, mask=None):
        scores = torch.matmul(query, key.transpose(-2, -1)) / (self.head_dim ** 0.5)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        attention_weights = F.softmax(scores, dim=-1)
        return torch.matmul(attention_weights, value)

    def combine_heads(self, x, batch_size):
        x = x.permute(0, 2, 1, 3).contiguous()
        return x.view(batch_size, -1, self.d_model)
```

- `compute_attention()` : compute attention weights using `F.softmax()`
- `torch.matmul(attention_weights, value)` : weighted sum of values

```
    def forward(self, query, key, value, mask=None):
        batch_size = query.size(0)

        query = self.split_heads(self.query_linear(query), batch_size)
        key = self.split_heads(self.key_linear(key), batch_size)
        value = self.split_heads(self.value_linear(value), batch_size)

        attention_weights = self.compute_attention(query, key, value, mask)

        output = self.combine_heads(attention_weights, batch_size)
        return self.output_linear(output)
```

- `self.output_linear()` : concatenate and project head outputs

Let's practice!

TRANSFORMER MODELS WITH PYTORCH