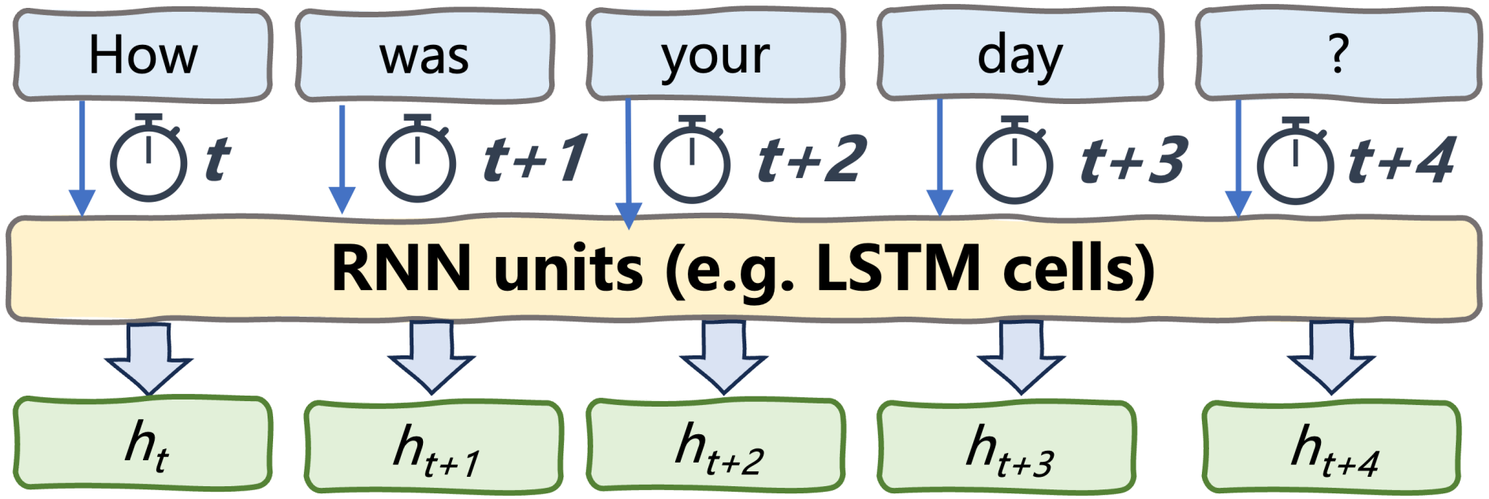
Attention mechanisms and positional encoding

INTRODUCTION TO LLMS IN PYTHON

Iván Palomares Carrascosa, PhD
Senior Data Science & AI Manager

datacamp

Why attention mechanisms?

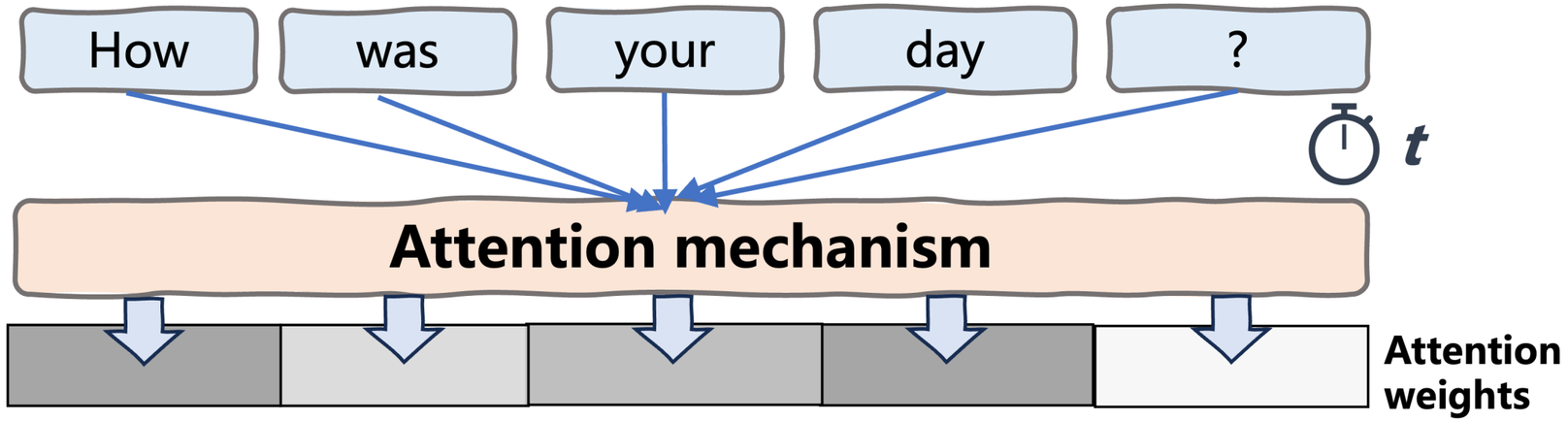


Recurrent architectures:

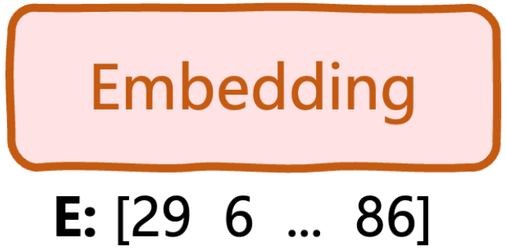
- Process sequence token by token
- Memorize recently processed information

Why attention mechanisms?

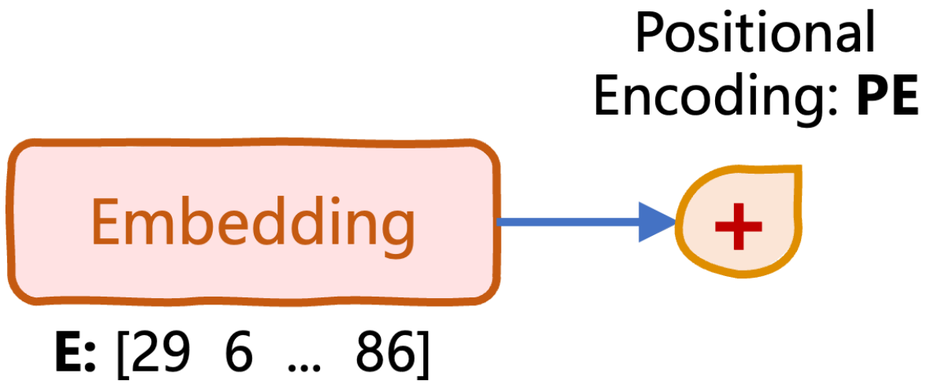
Self-attention
- Use whole sequence to weigh each element at time ***t***



Positional encoding

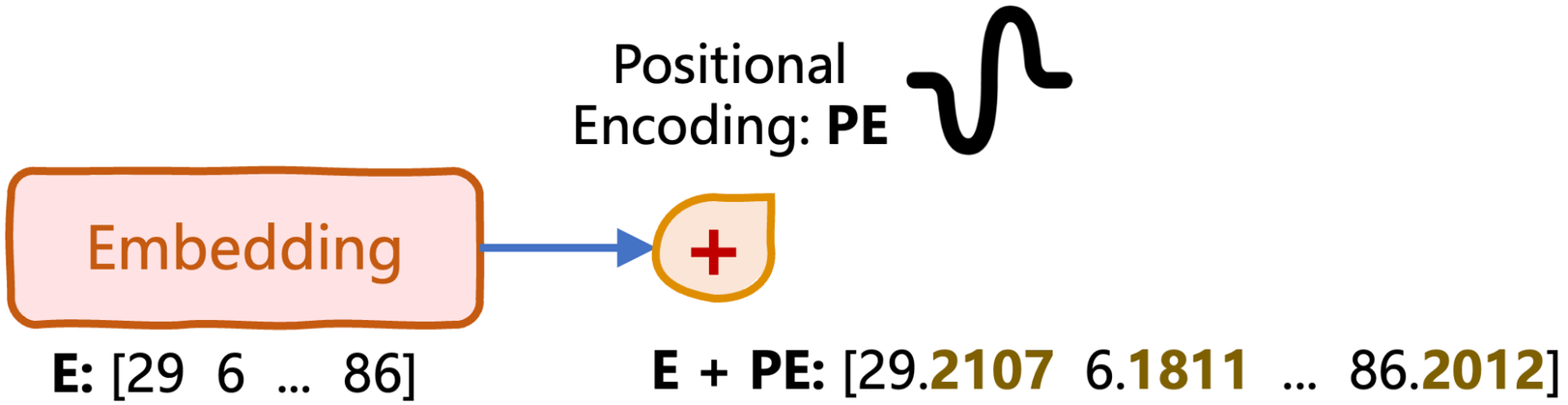


Positional encoding



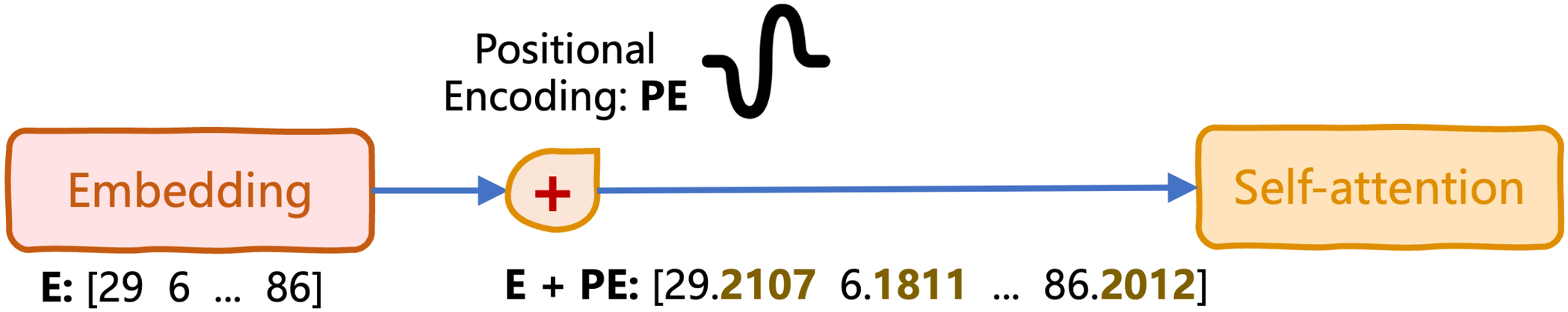
- Create positional encodings vector **PE** for token embedding **E**

Positional encoding



- Create positional encodings vector **PE** for token embedding **E**
  - Based on *sine* and *cosine* functions
- Add **PE** to **E**

Positional encoding



- Create positional encodings vector **PE** for token embedding **E**
  - Based on *sine* and *cosine* functions
- Add **PE** to **E**

Positional encoder class

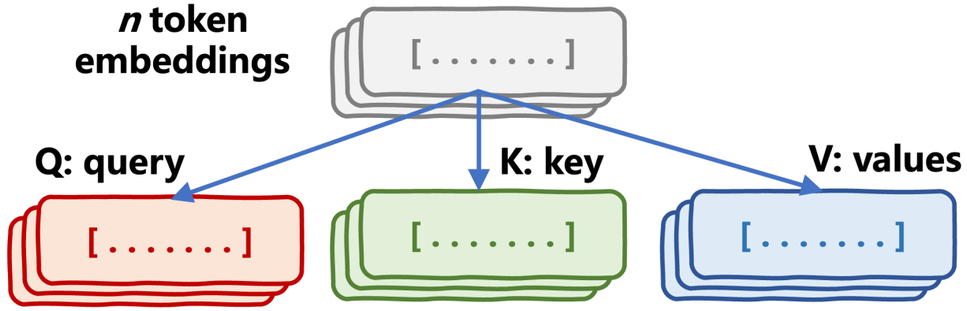
```
class PositionalEncoder(nn.Module):
    def __init__(self, d_model, max_seq_length=512):
        super(PositionalEncoder, self).__init__()
        self.d_model = d_model
        self.max_seq_length = max_seq_length

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

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

- Set `max_seq_length` and embedding size: `d_model`
- Create positional encoding matrix `pe`, for sequences up to `max_seq_length`
- `position`: position indices in the sequence
- `div_term`: a term to scale position indices
- Alternately apply sine and cosine to `pe`
  - `register_buffer()`: set `pe` as non-trainable
- Add positional encodings `pe` to input tensor of sequence embeddings, `x`

Self-attention mechanism anatomy

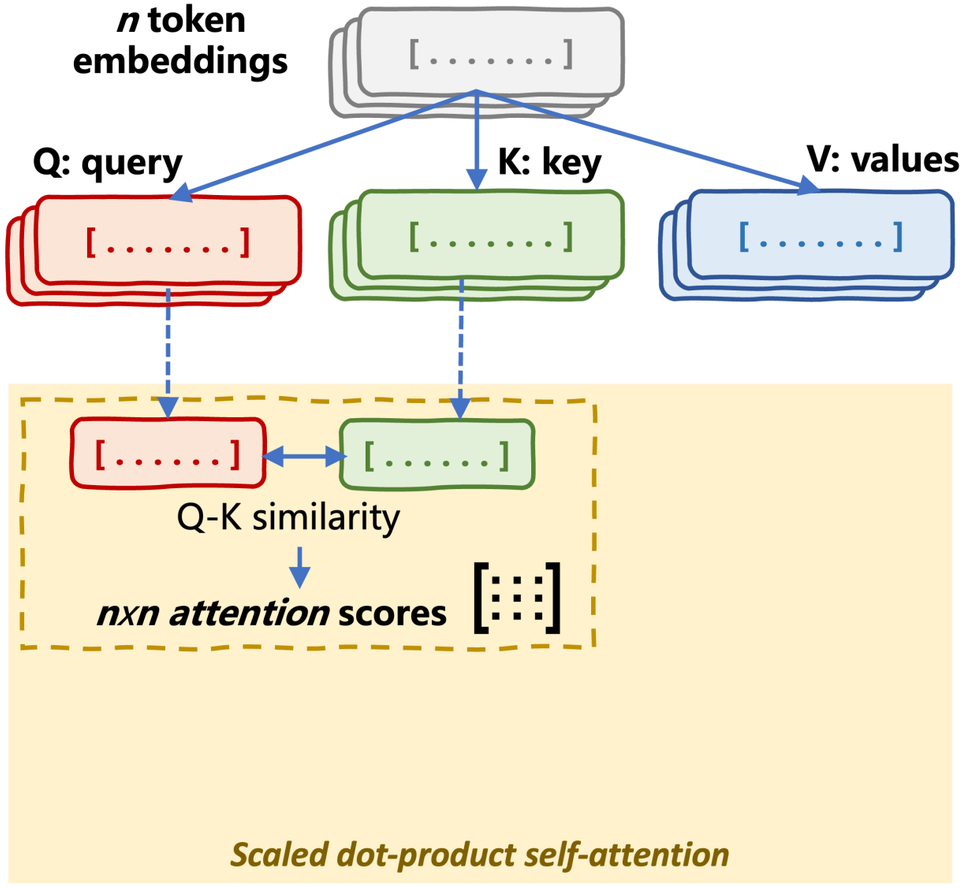
***n* token embeddings**

Self-attention mechanism anatomy



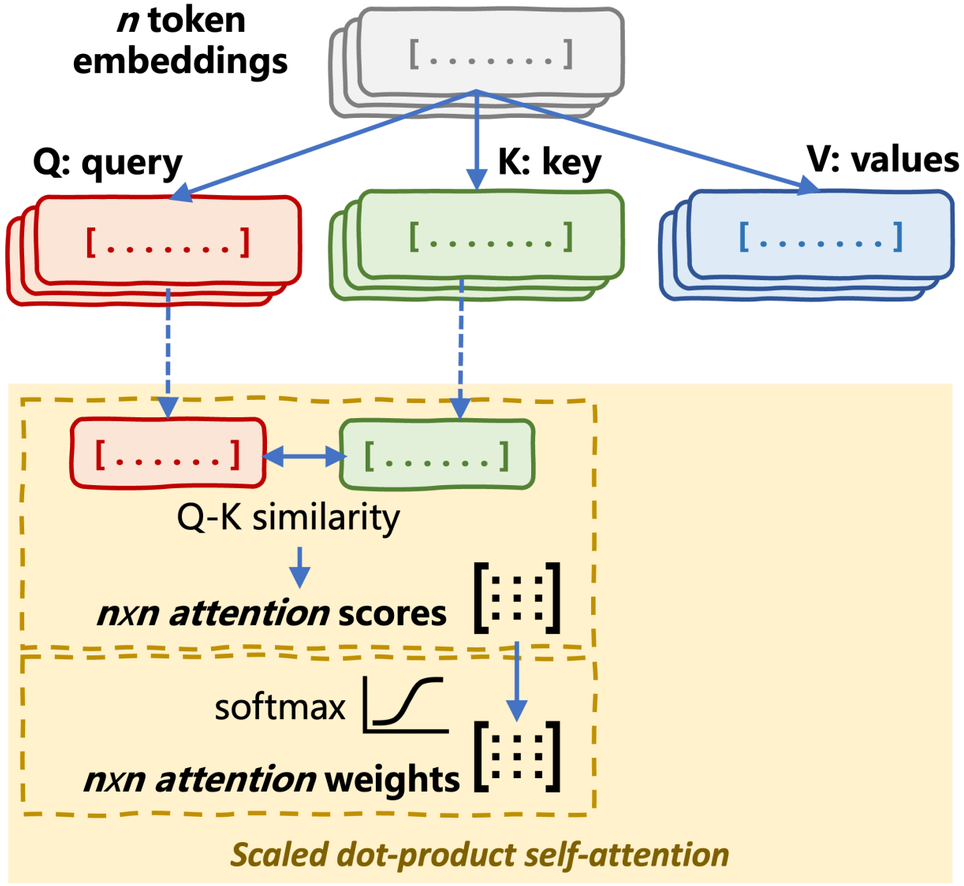
- **Q, K, V**: three linear projections of each token embedding

Self-attention mechanism anatomy



- **Q, K, V**: three linear projections of each token embedding
- Similarity between **Q** and **K** pairs
  - Attention scores matrix

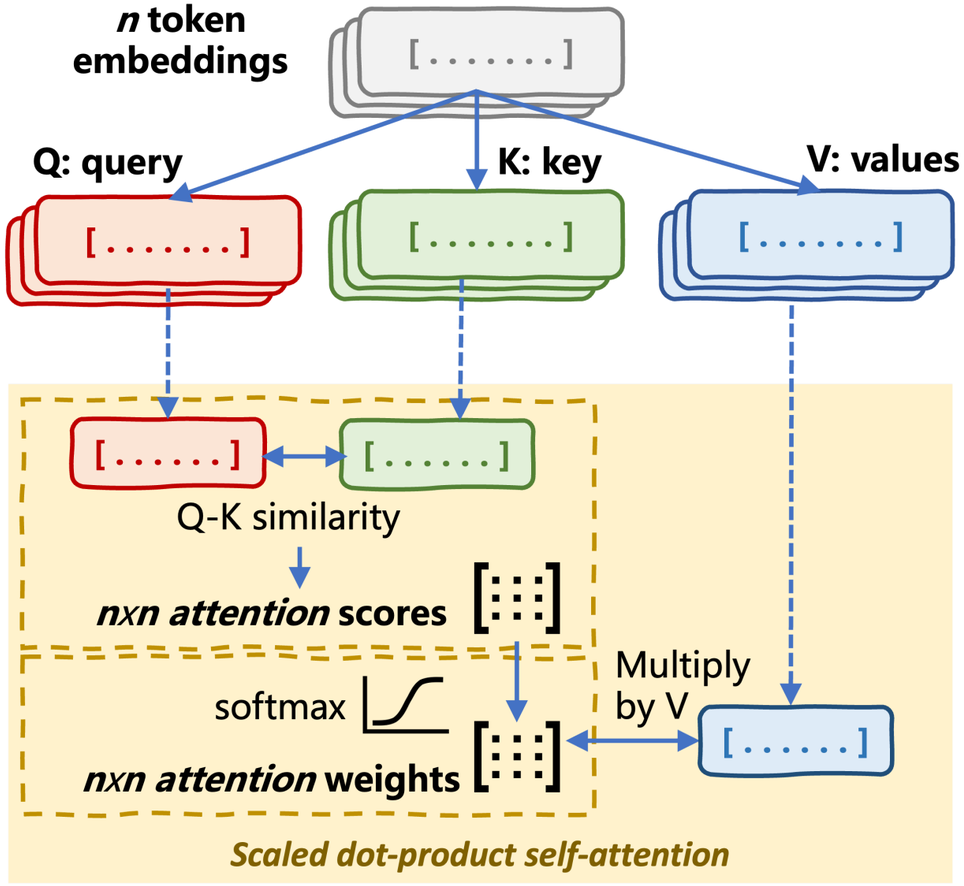
Self-attention mechanism anatomy



- **Q, K, V**: three linear projections of each token embedding
- Similarity between **Q** and **K** pairs
  - **Attention scores** matrix
- Softmax scaling
  - **Attention weights** matrix

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

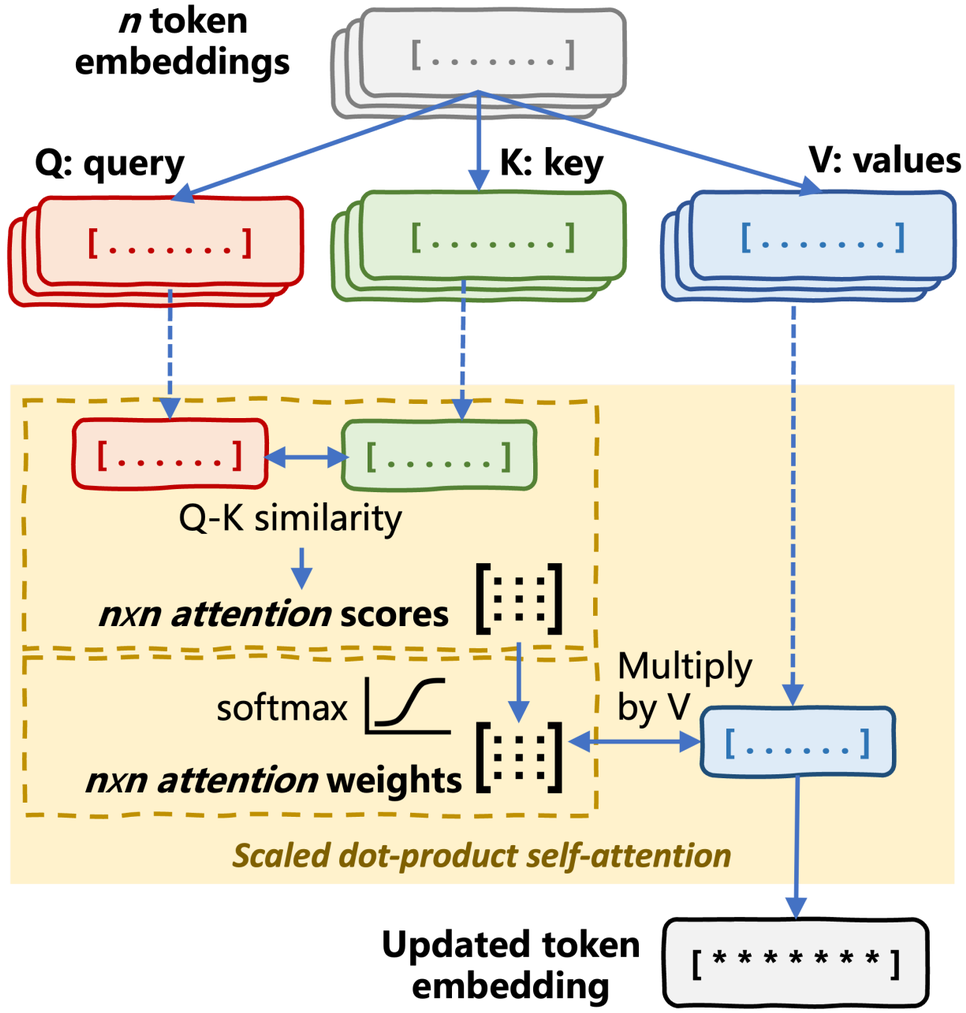
Self-attention mechanism anatomy



- **Q, K, V**: three linear projections of each token embedding
- Similarity between **Q** and **K** pairs
  - **Attention scores** matrix
- Softmax scaling
  - **Attention weights** matrix

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

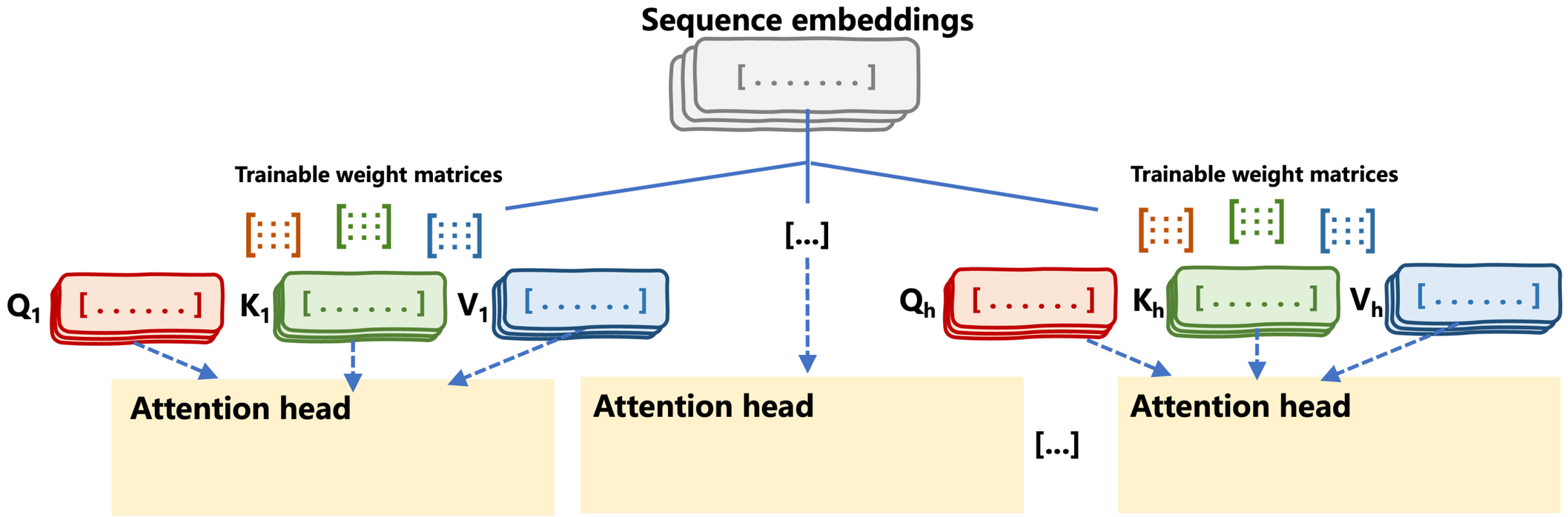
Self-attention mechanism anatomy



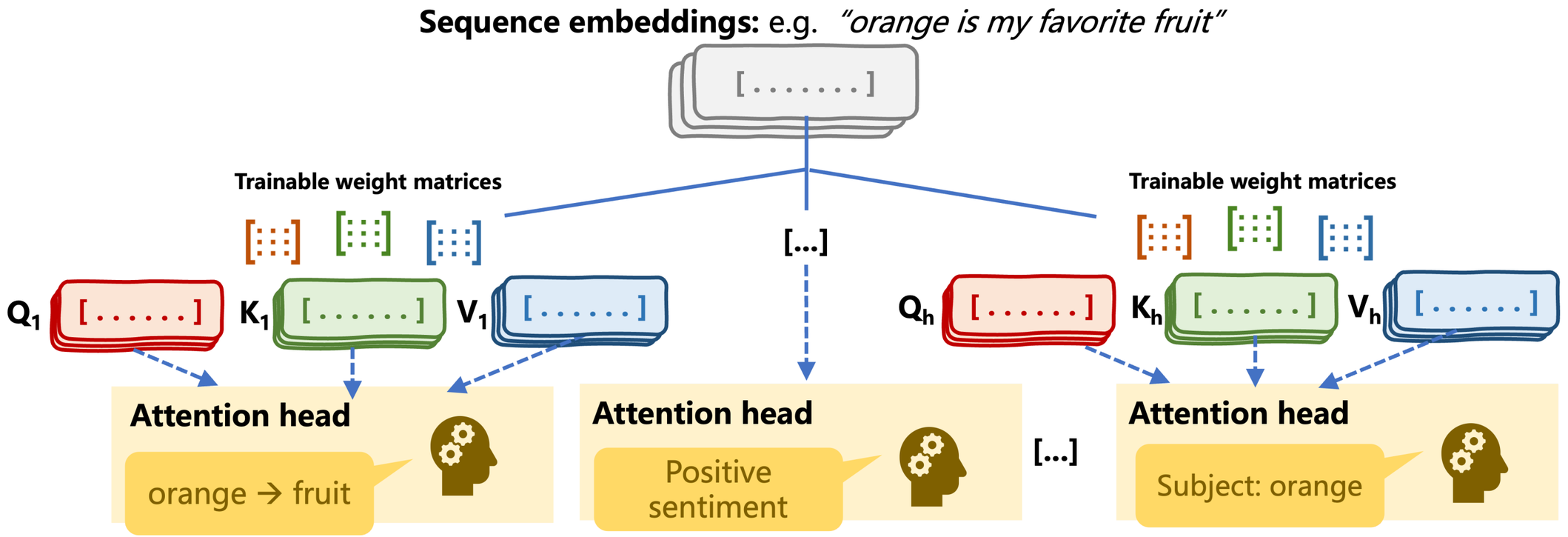
- **Q, K, V**: three linear projections of each token embedding
- Similarity between **Q** and **K** pairs
  - **Attention scores** matrix
- Softmax scaling
  - **Attention weights** matrix

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

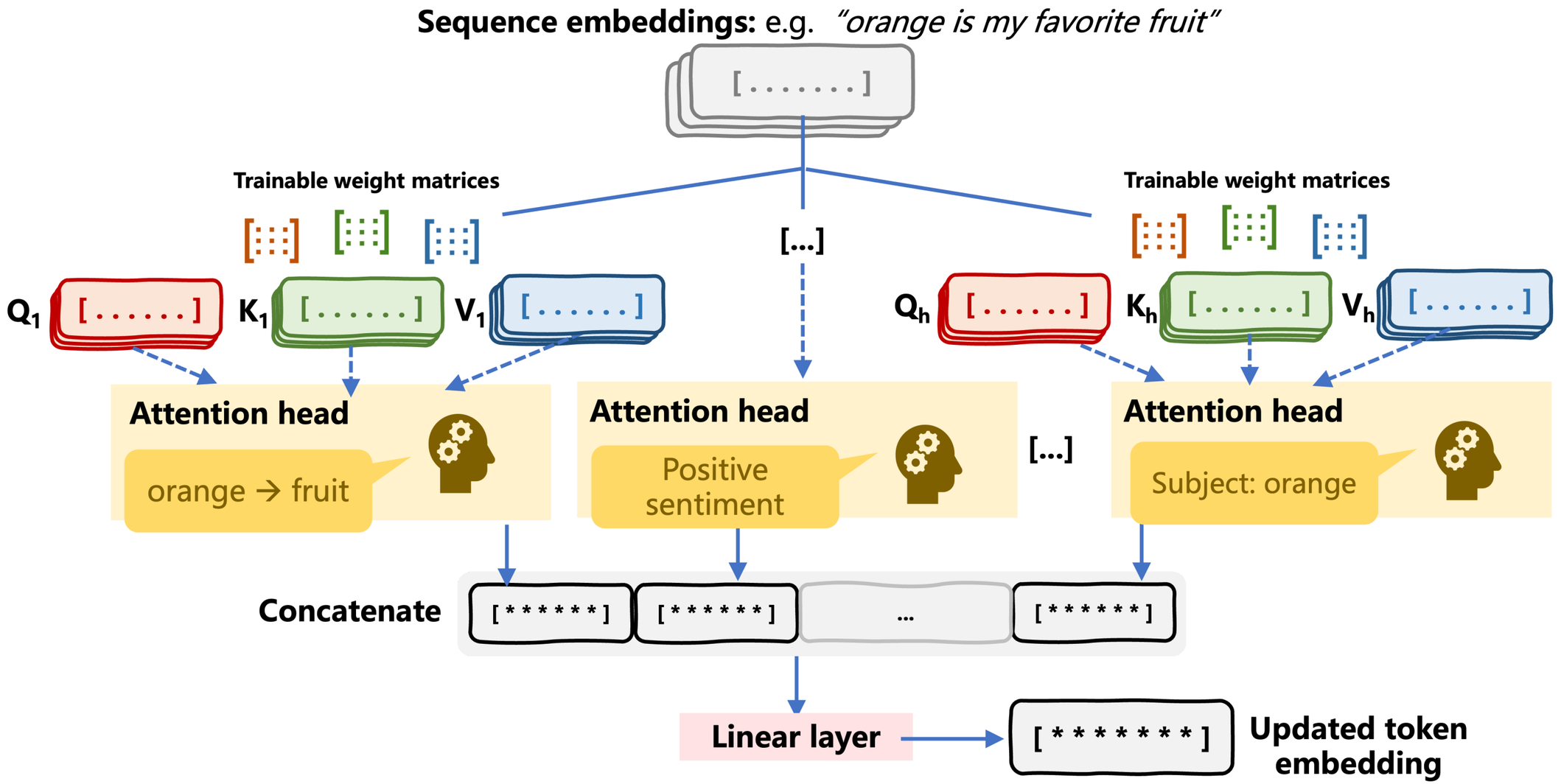
Multi-headed self-attention



Multi-headed self-attention



Multi-headed self-attention



Multi-headed attention class

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

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        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)
        self.key_linear = nn.Linear(d_model, d_model)
        self.value_linear = nn.Linear(d_model, d_model)
        self.output_linear = nn.Linear(d_model, d_model)
```

- `num_heads` : number of attention heads, each handling embeddings of size `head_dim`
- Setting up linear transformations (`nn.Linear()`) for attention inputs and output

Multi-headed attention class

```
    def split_heads(self, x, batch_size):
        x = x.view(batch_size, -1, self.num_heads, self.head_dim)
        return x.permute(0, 2, 1, 3).contiguous().
          view(batch_size * self.num_heads, -1, self.head_dim)

    def compute_attention(self, query, key, mask=None):
        scores = torch.matmul(query, key.permute(1, 2, 0))
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float("-1e9"))
        attention_weights = F.softmax(scores, dim=-1)
        return attention_weights
```

- `split_heads()` : splits the input across attention heads
- `compute_attention()` : compute attention weights using `F.softmax()`

Multi-headed attention class

```
    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, mask)

        output = torch.matmul(attention_weights, value)
        output = output.view(batch_size, self.num_heads, -1, self.head_dim).
                 permute(0, 2, 1, 3).contiguous().view(batch_size, -1, self.d_model)
        return self.output_linear(output)
```

- `forward()` : orchestrate multi-headed attention mechanism workflow.
  - Concatenate and project head outputs with `self.output_linear()`

Let's practice!

INTRODUCTION TO LLMS IN PYTHON

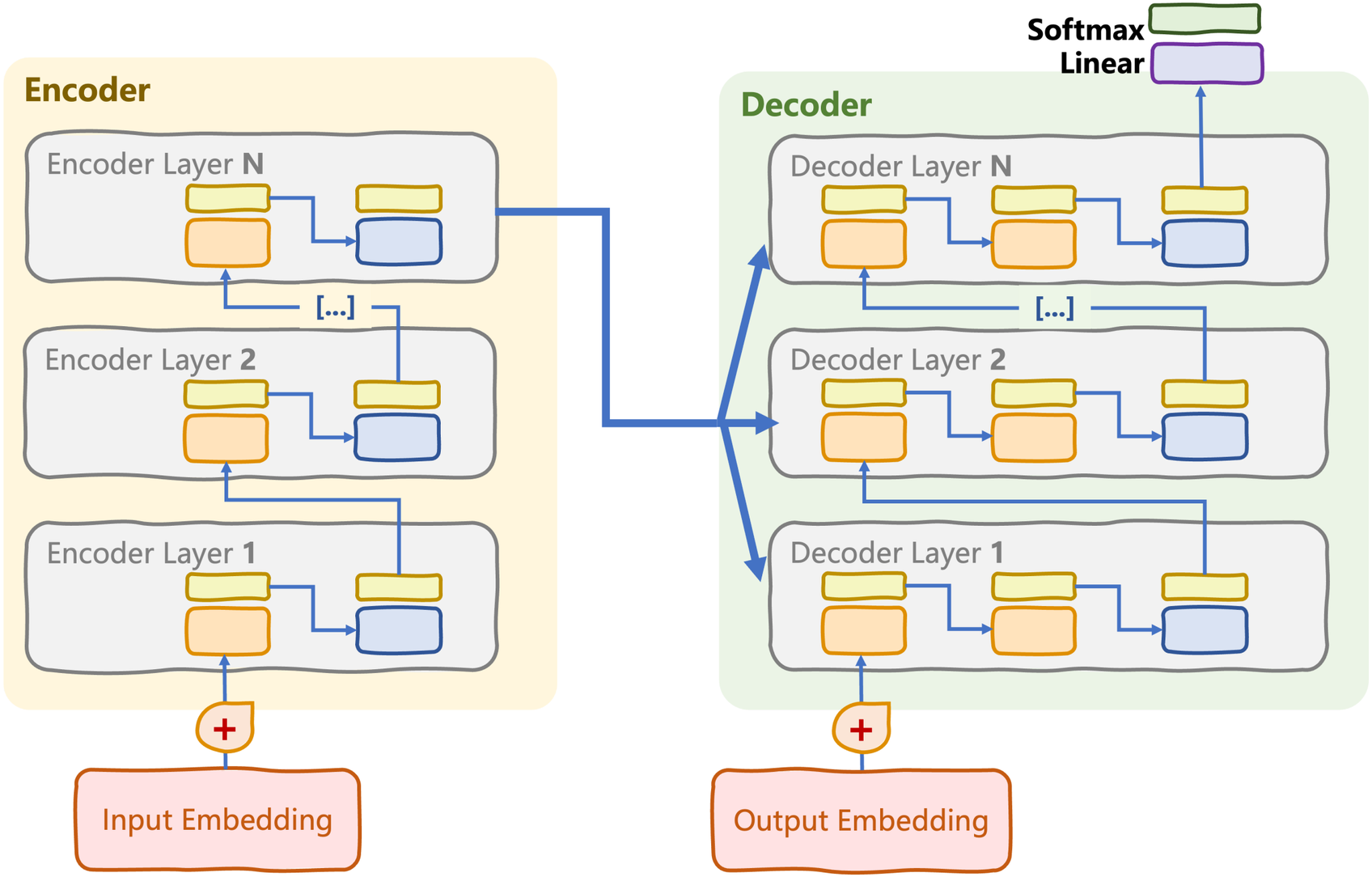
Building an encoder transformer

INTRODUCTION TO LLMS IN PYTHON

Iván Palomares Carrascosa, PhD
Senior Data Science & AI Manager

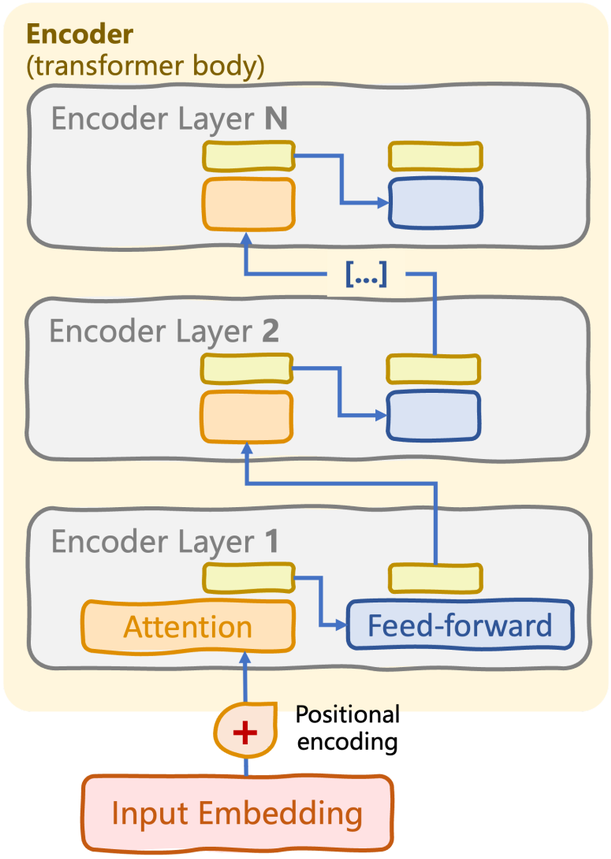
datacamp

From original to encoder-only transformer



From original to encoder-only transformer

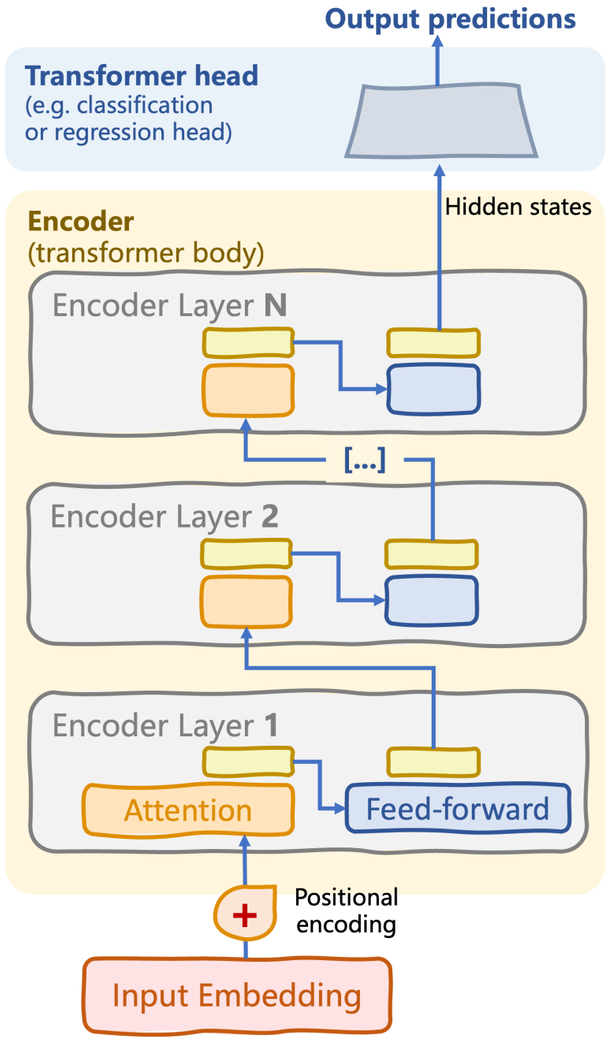
Transformer body: encoder stack with N encoder layers



Encoder layer

- Multi-headed self-attention
- Feed-forward (sub)layers
- Layer normalizations, skip connections, dropouts

From original to encoder-only transformer



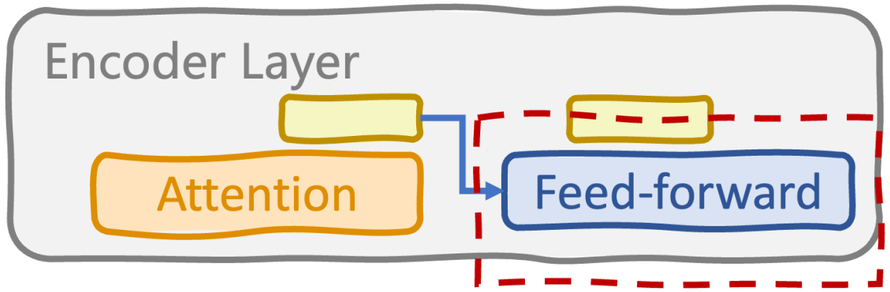
Transformer body: encoder stack with N encoder layers

> **Encoder layer**
>
> - Multi-headed self-attention
> - Feed-forward (sub)layers
> - Layer normalizations, skip connections, dropouts

Transformer head: process encoded inputs (hidden states) to produce output prediction

> **Supervised task:** classification, regression

Feed-forward sublayer in encoder layer



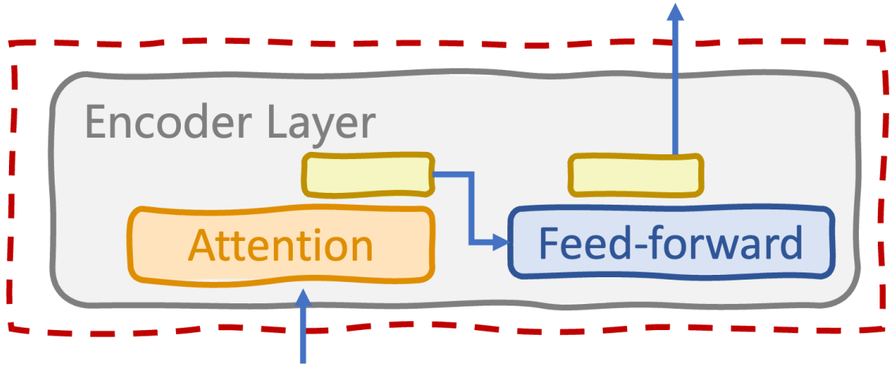

```
class FeedForwardSubLayer(nn.Module):
    def __init__(self, d_model, d_ff):
        super(FeedForwardSubLayer, self).__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.fc2(self.relu(self.fc1(x)))
```

2 x fully connected + ReLU activation

- `d_ff` : dimension between linear layers
- `forward()` : processes attention outputs to capture complex, non-linear patterns

Encoder layer

```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = FeedForwardSubLayer(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x
```



- Multi-headed self-attention
- Feed-forward sublayer
- Layer normalizations and dropouts

`forward()` : forward-pass through encoder layer

- `mask` prevents processing padding tokens

Masking the attention process

"I really like to travel"

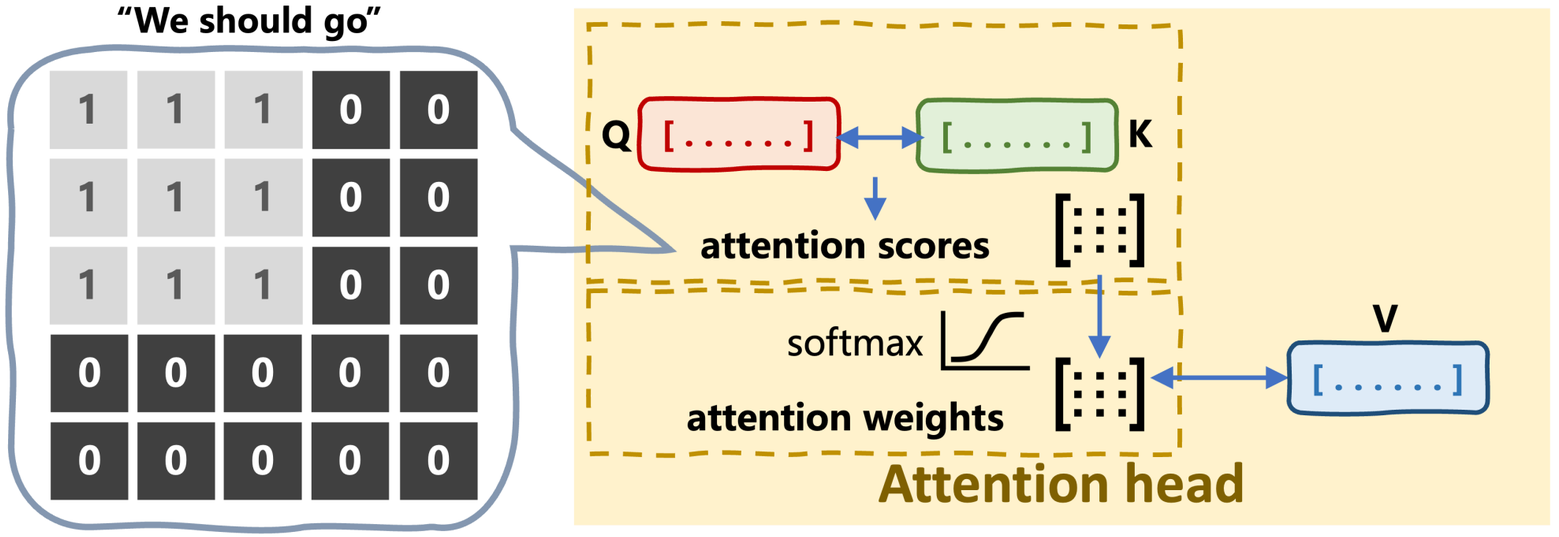
835	8246	94	302	7547

"Eastern Asia looks fascinating"

5839	236	492	28	0

"We should go"

12	641	46	0	0



Transformer body: encoder

```
class TransformerEncoder(nn.Module):
    def __init__(self, vocab_size, d_model, num_layers,
            num_heads, d_ff, dropout, max_sequence_length):
        super(TransformerEncoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)

        self.positional_encoding =
           PositionalEncoding(d_model, max_sequence_length)
        self.layers = nn.ModuleList(
          [EncoderLayer(d_model, num_heads, d_ff, dropout)
           for _ in range(num_layers)]
        )

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

Encoder structure:

- Input embeddings based on `vocab_size`
- Positional encoding
- Stack of `num_layers` encoder layers, using `nn.ModuleList()`

- `forward()`: forward-pass `x` through transformer body

Transformer head

```
class ClassifierHead(nn.Module):
    def __init__(self, d_model, num_classes):
        super(ClassifierHead, self).__init__()
        self.fc = nn.Linear(d_model, num_classes)

    def forward(self, x):
        logits = self.fc(x)
        return F.log_softmax(logits, dim=-1)
```

```
class RegressionHead(nn.Module):
    def __init__(self, d_model, output_dim):
        super(RegressionHead, self).__init__()
        self.fc = nn.Linear(d_model, output_dim)

    def forward(self, x):
        return self.fc(x)
```

Classification head

- **Tasks:** text classification, sentiment analysis, NER, extractive QA, etc.
- `fc` : **fully connected linear layer**
  - Transforms encoder hidden states into `num_classes` class probabilities

Regression head

- **Tasks:** estimate text readability, language complexity, etc.
  - `output_dim` is 1 when predicting a single numerical value

Let's practice!

INTRODUCTION TO LLMS IN PYTHON

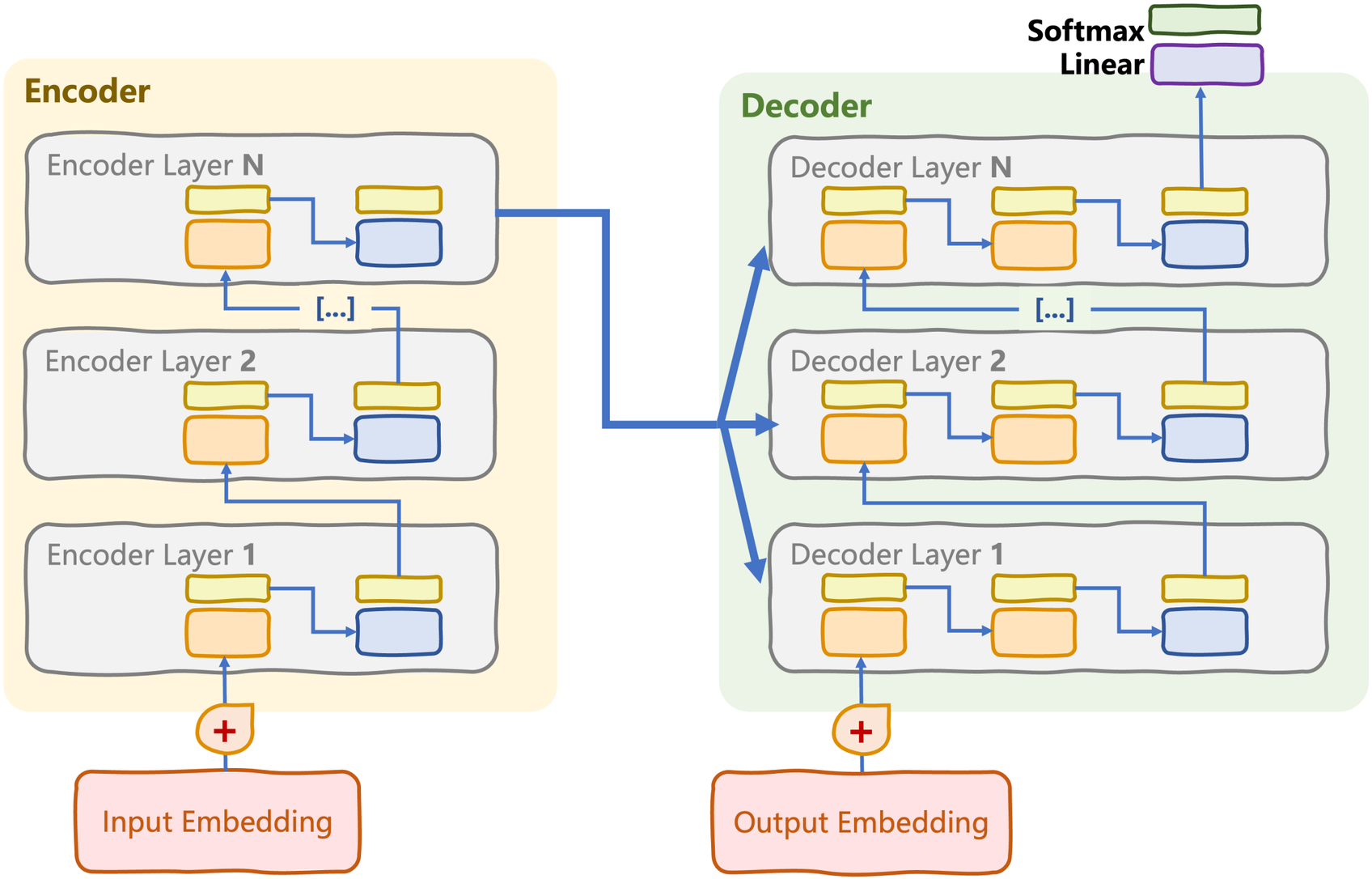
Building a decoder transformer

INTRODUCTION TO LLMS IN PYTHON

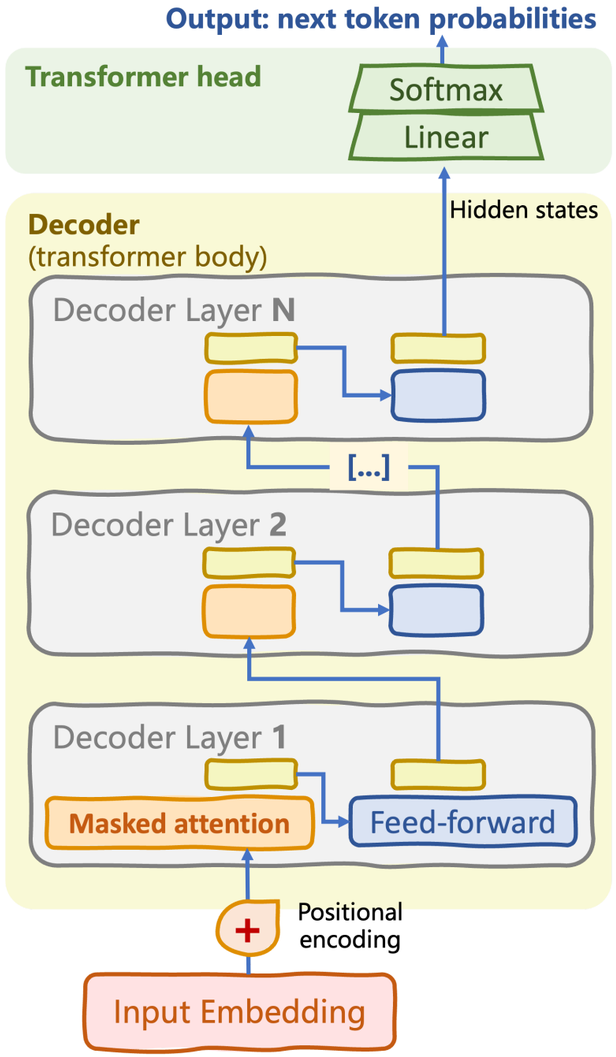
Iván Palomares Carrascosa, PhD
Senior Data Science & AI Manager

datacamp

From original to decoder-only transformer

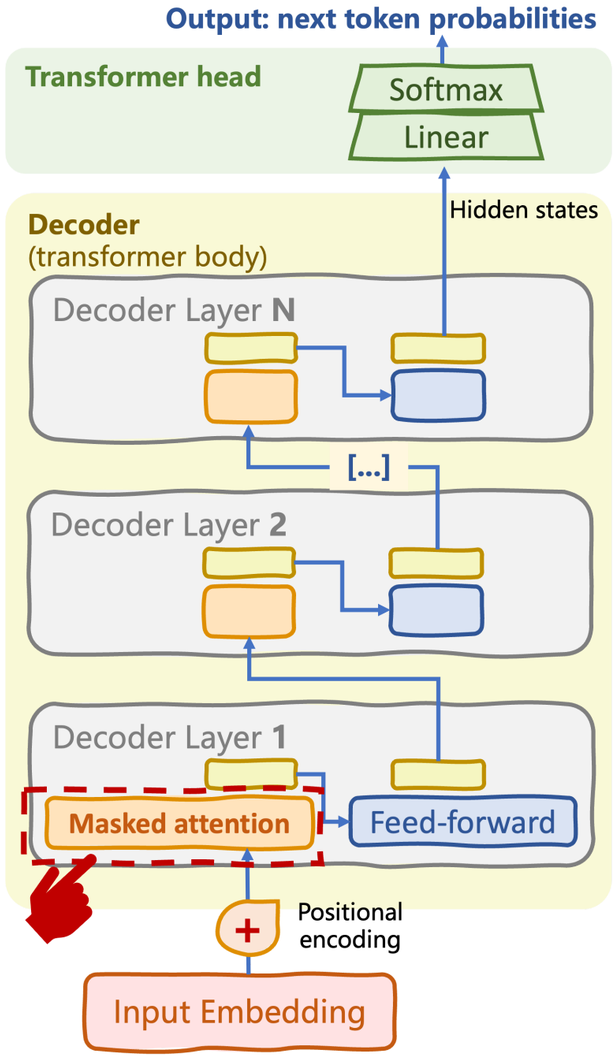


From original to decoder-only transformer



Autoregressive sequence generation: text generation and completion

From original to decoder-only transformer

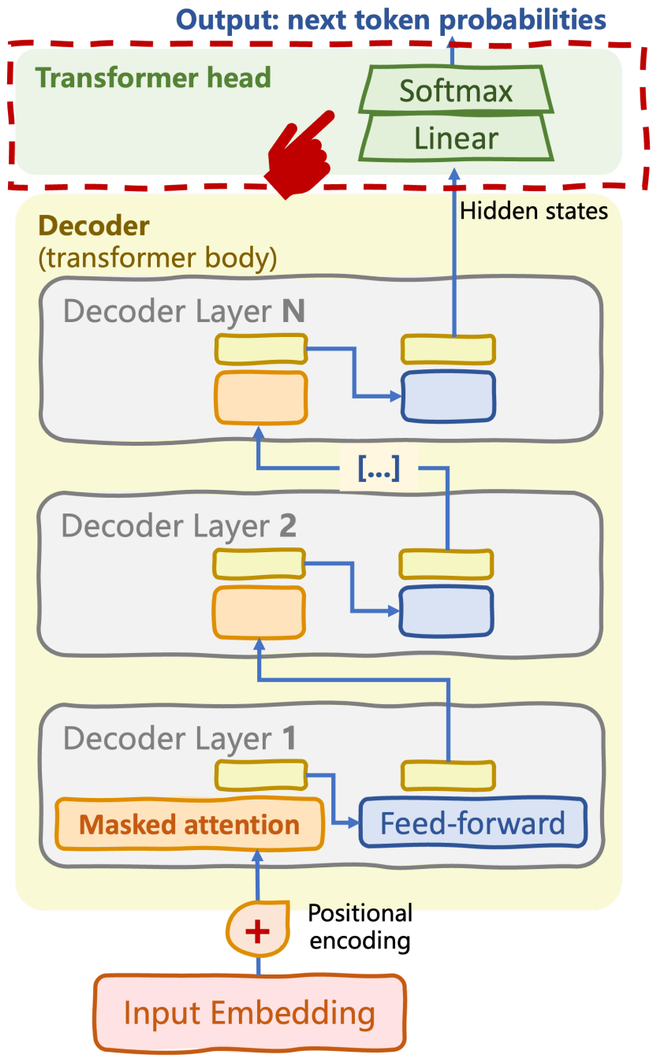


Autoregressive sequence generation: text generation and completion

Masked multi-head self-attention

- **Upper triangular mask**
  - Hide future positions in sequence
  - Only attend tokens before current one

From original to decoder-only transformer



Autoregressive sequence generation: text generation and completion

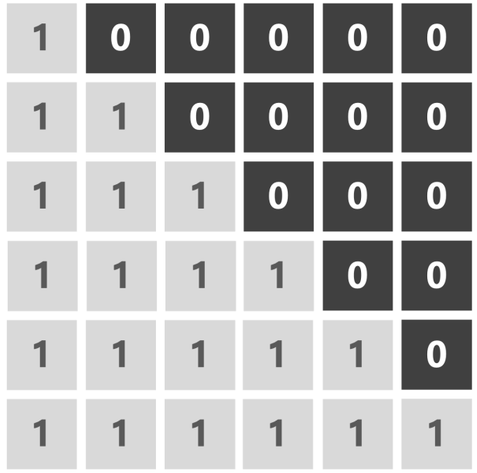
Masked multi-head self-attention

- **Upper triangular mask**
  - Hide future positions in sequence
  - Only attend tokens before current one

Decoder-only transformed head

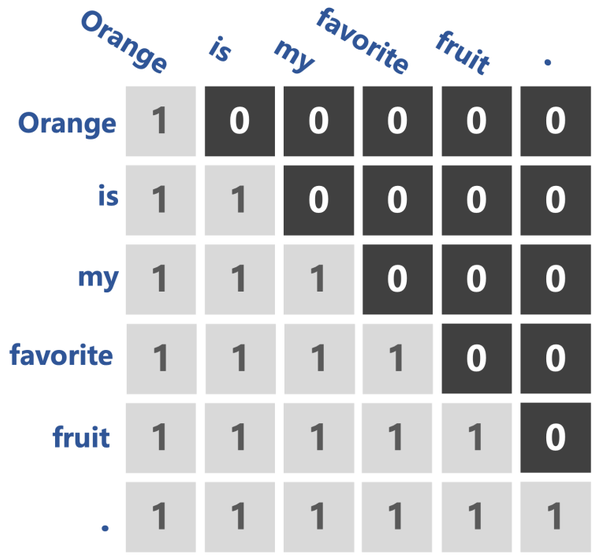
- Linear + Softmax over vocabulary
- Predict most *likely* next token

Masked self-attention



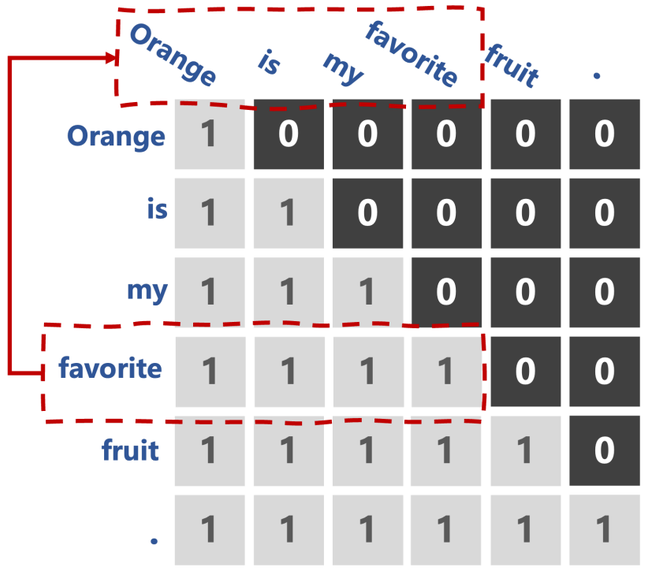
- Key to **autoregressive** or **causal** behavior
- Triangular (causal) attention mask

Masked self-attention



- Key to **autoregressive** or **causal** behavior
- Triangular (causal) attention mask
- A token only pays **attention to "past"** (left-hand side) **information** in the sequence

Masked self-attention




```
self_attention_mask = (1 - torch.triu(
  torch.ones(1, sequence_length, sequence_length),
  diagonal=1)).bool()
(...)
output = decoder(input_sequence, self_attention_mask)
```

- Key to **autoregressive** or **causal** behavior
- Triangular (causal) attention mask
- A token only pays **attention to "past"** (left-hand side) **information** in the sequence
  - ***"favorite"***: *"orange"*, *"is"*, *"my"*, *"favorite"*
- Enforced causal attention: predict next word to generate, e.g. *"fruit"*
- Mask built and passed to the model
  - `self_attention_mask`
  - **Same multi-headed attention mechanism:** only the mask is different

Transformer body (decoder) and head

```
class DecoderOnlyTransformer(nn.Module):
    def __init__(self, vocab_size, d_model, num_layers, num_heads, d_ff, dropout, max_sequence_length):
        super(TransformerDecoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_sequence_length)
        self.layers = nn.ModuleList([DecoderLayer(d_model, num_heads, d_ff, dropout) for _ in range(num_layers)])
        self.fc = nn.Linear(d_model, vocab_size)

    def forward(self, x, self_mask):
        x = self.embedding(x)
        x = self.positional_encoding(x)
        for layer in self.layers:
            x = layer(x, self_mask)
        x = self.fc(x)
        return F.log_softmax(x, dim=-1)
```

- `self.fc` : output linear layer with `vocab_size` neurons
- Add `self.fc` and *softmax* activation in forward pass

Let's practice!

INTRODUCTION TO LLMS IN PYTHON

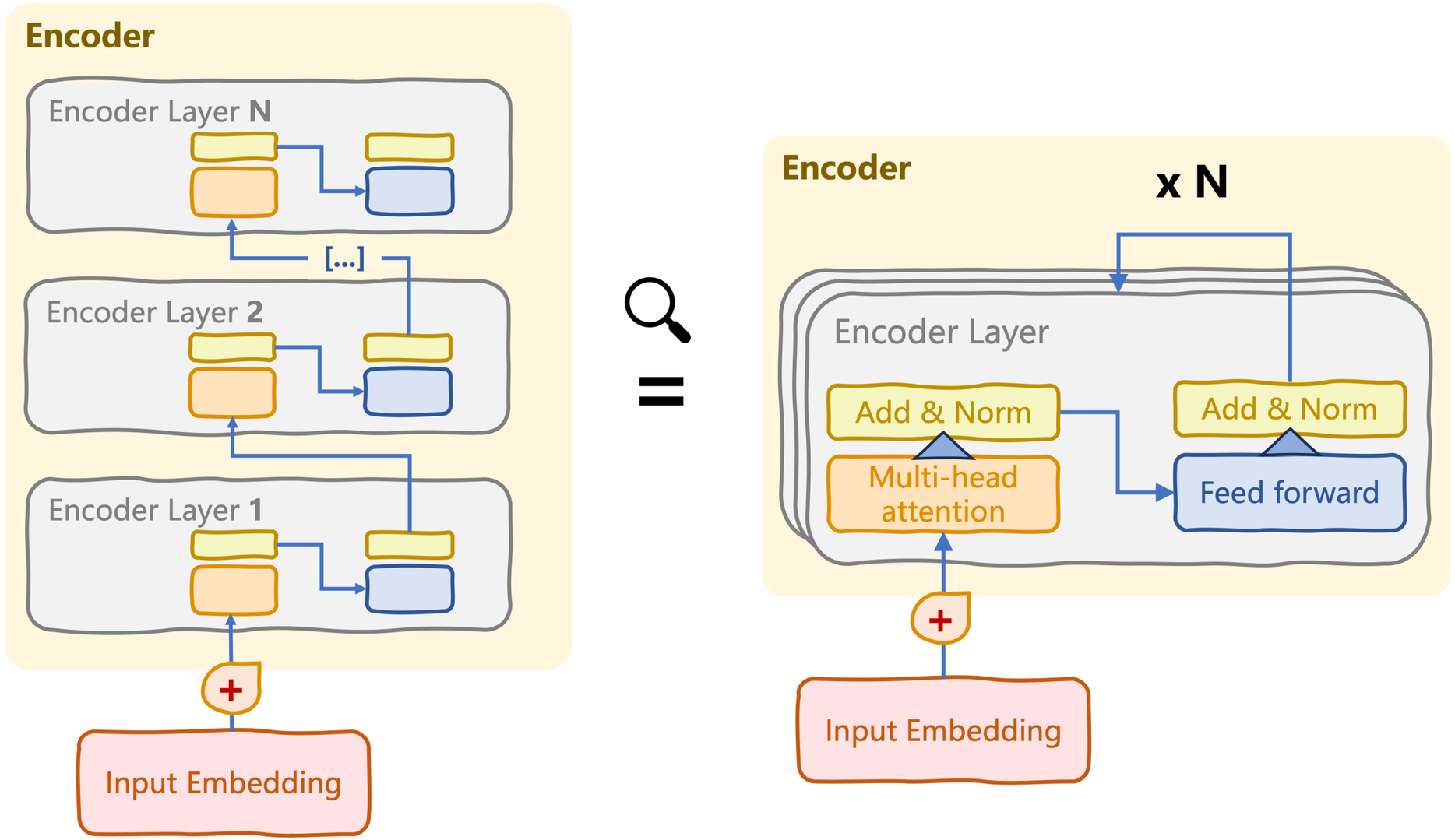
Building an encoder-decoder transformer

INTRODUCTION TO LLMS IN PYTHON

Iván Palomares Carrascosa, PhD
Senior Data Science & AI Manager

datacamp

Transformer architecture: encoder recap

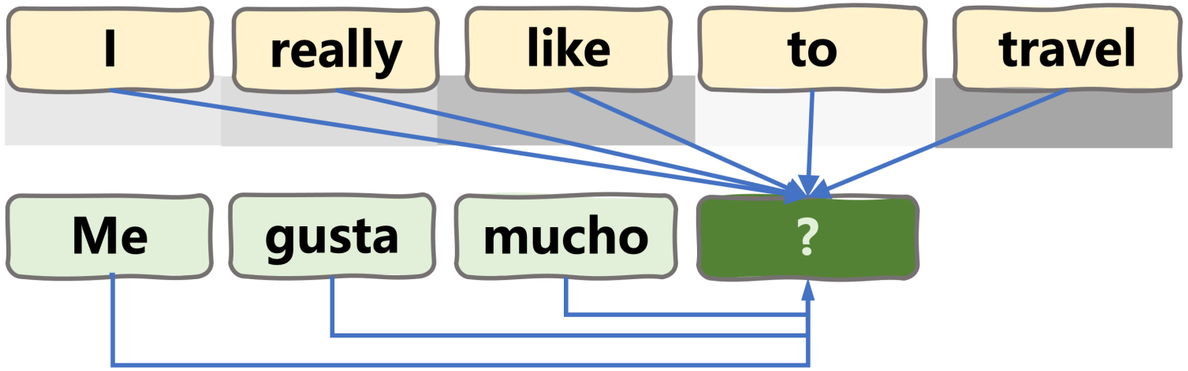


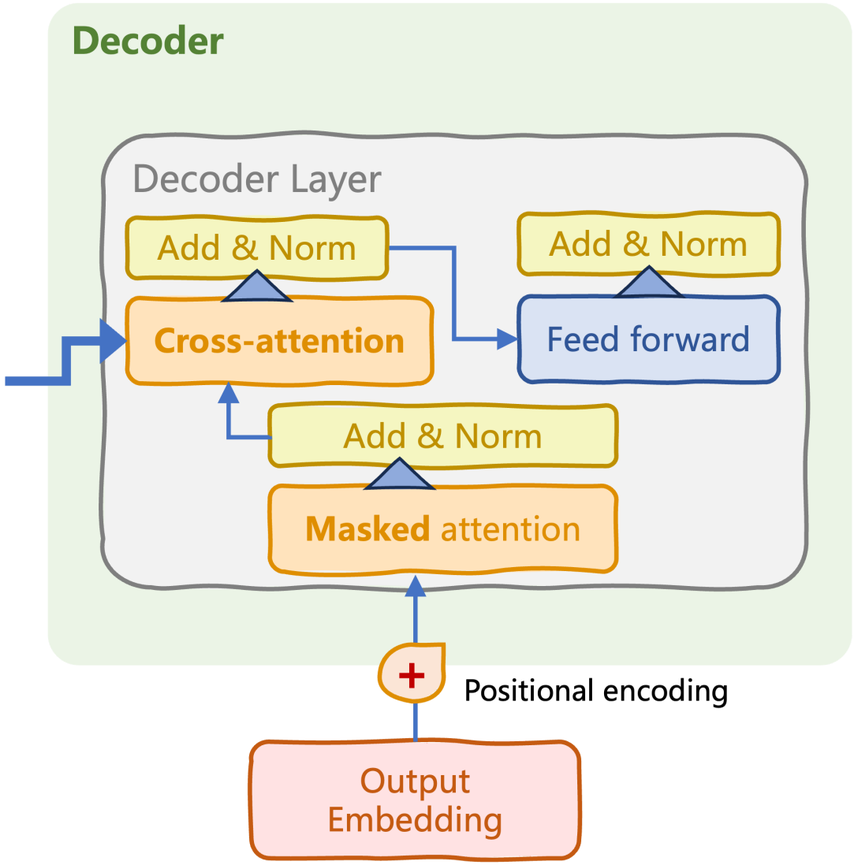
Cross-attention mechanism

Cross-attention: double inputs.

1. Information processed throughout **decoder**
2. Final hidden states from **encoder** block

> *Look back at processed input sequence.*
> *Find out next output token to generate.*





Cross-attention mechanism

Cross-attention: double inputs.

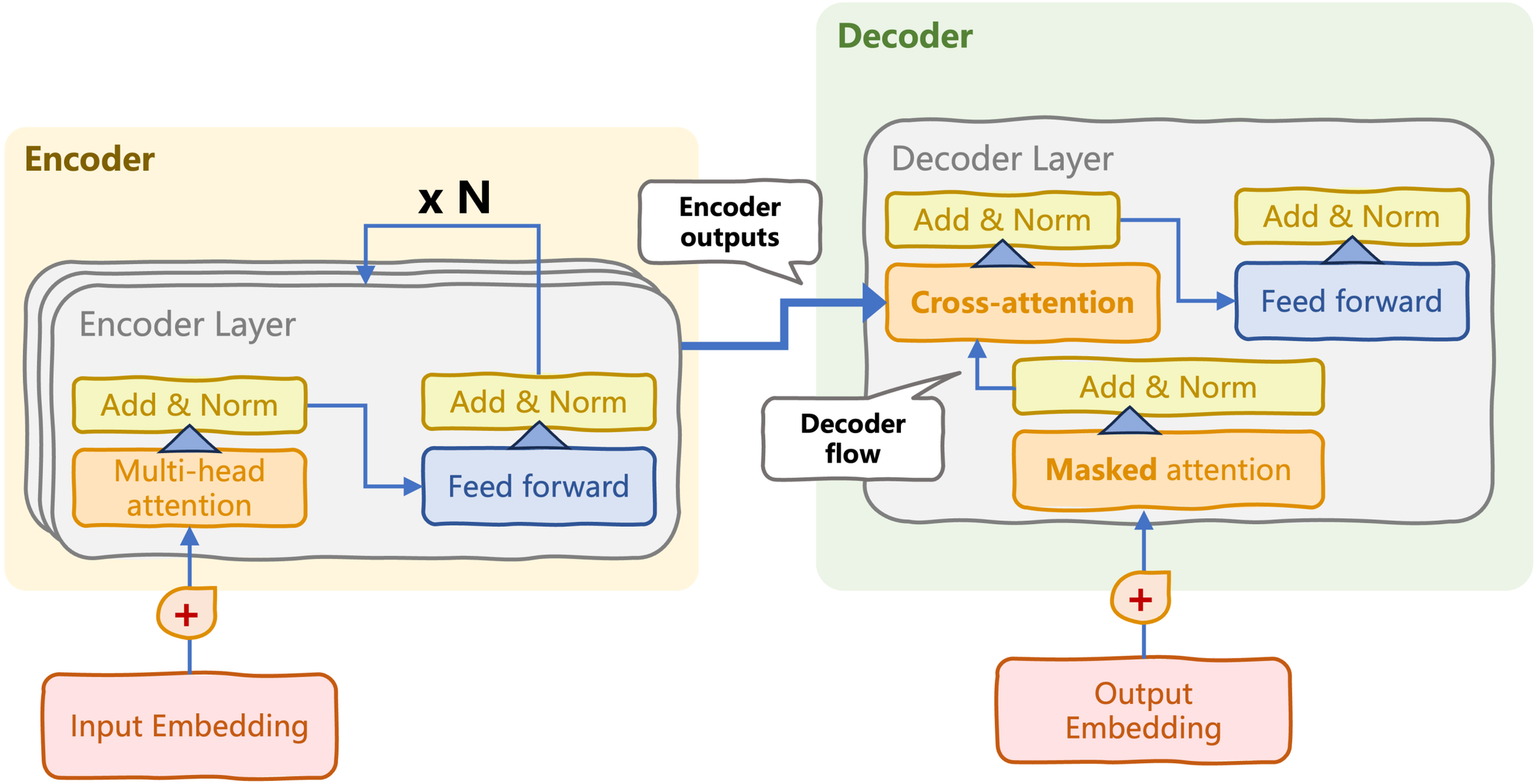
1. Information processed throughout **decoder**
2. Final hidden states from **encoder** block

> *Look back at processed input sequence.*
> *Find out next output token to generate.*

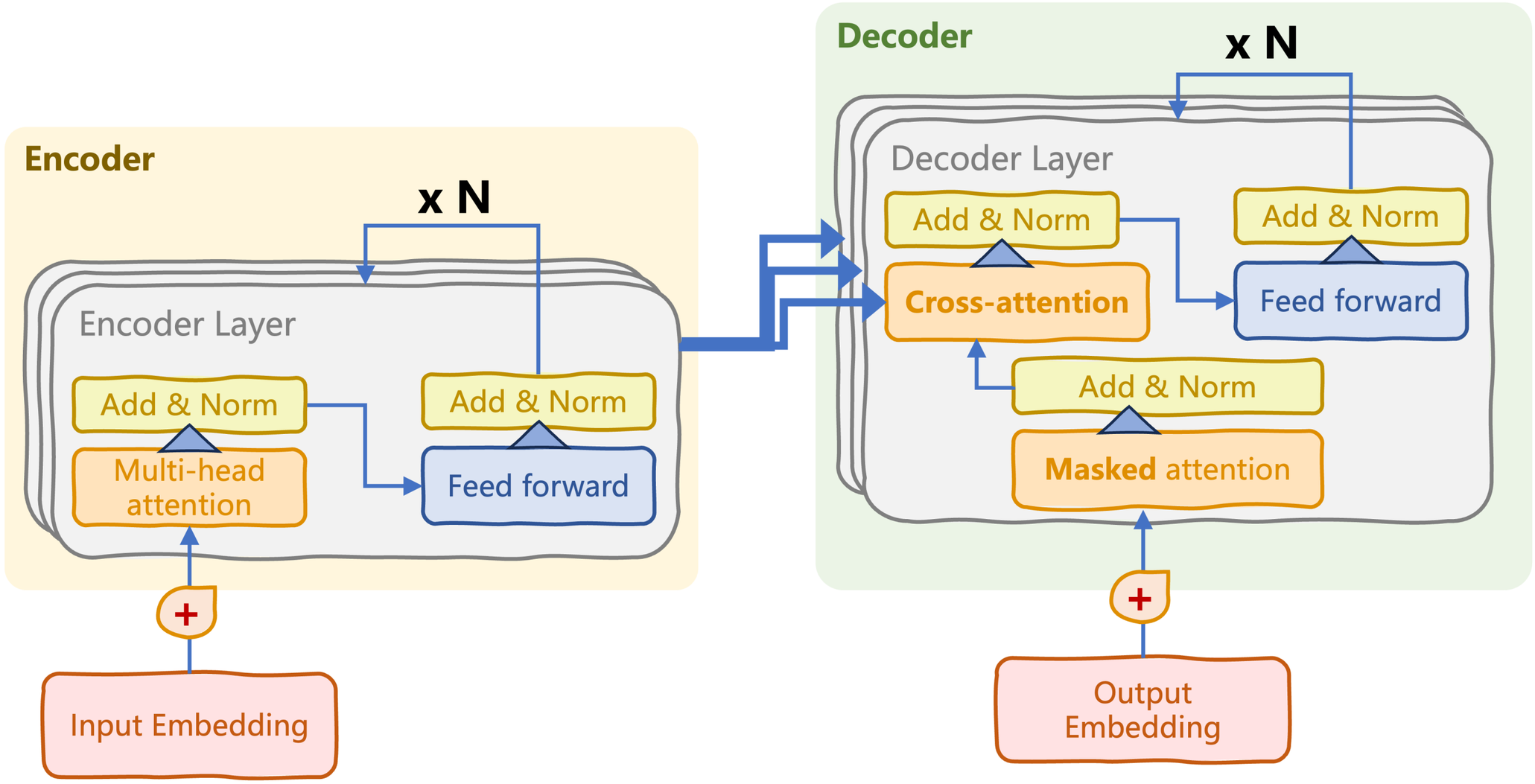
- `x` : **decoder information flow**, becomes cross-attention **query**
- `y` : **encoder output**, becomes cross-attention **key** and **values**

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(DecoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(
                            d_model, num_heads)
        self.cross_attn = MultiHeadAttention(
                            d_model, num_heads)
        ...

    def forward(self, x, y, causal_mask, cross_mask):
        self_attn_output = self.self_attn(x, x, x,
                                          causal_mask)
        x = self.norm1(x + self.dropout(self_attn_output))

        cross_attn_output = self.cross_attn(x, y, y,
                                            cross_mask)
        x = self.norm2(x + self.dropout(cross_attn_output))
        ...
```

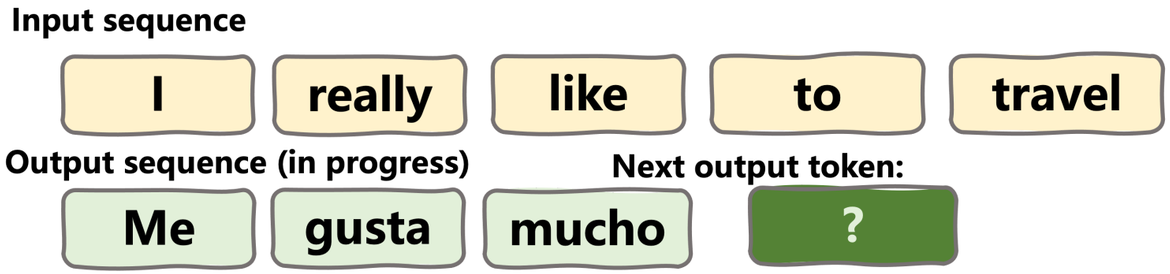
Encoder meets decoder



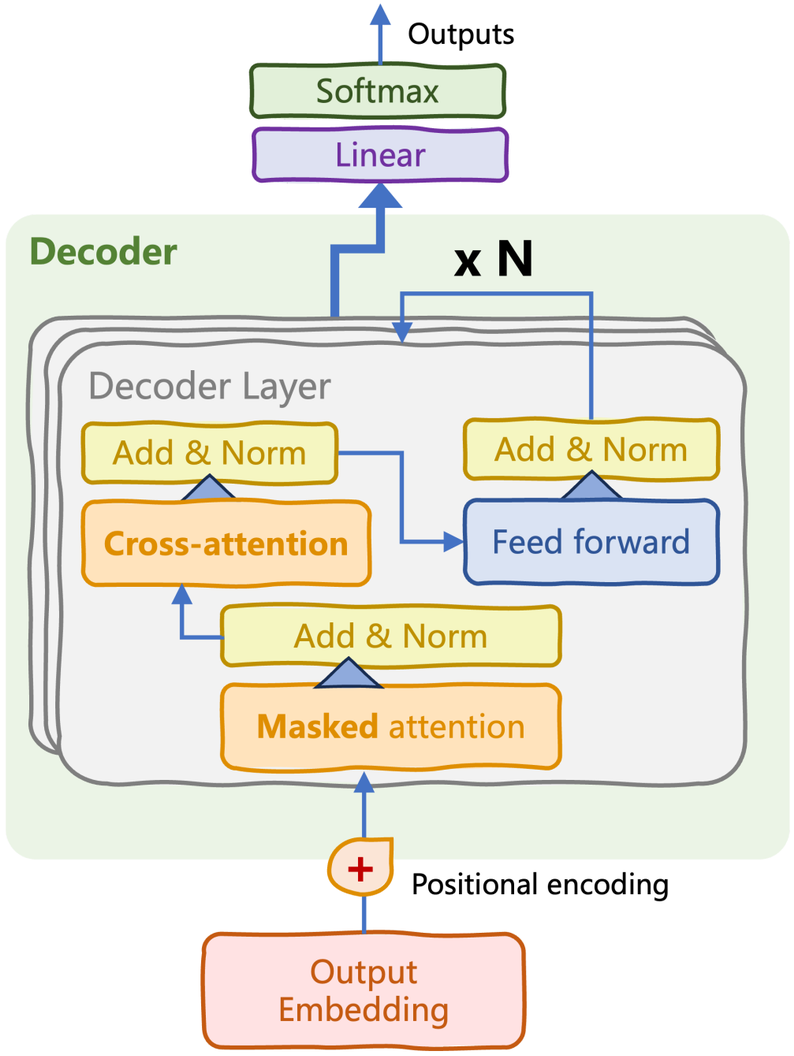
Encoder meets decoder



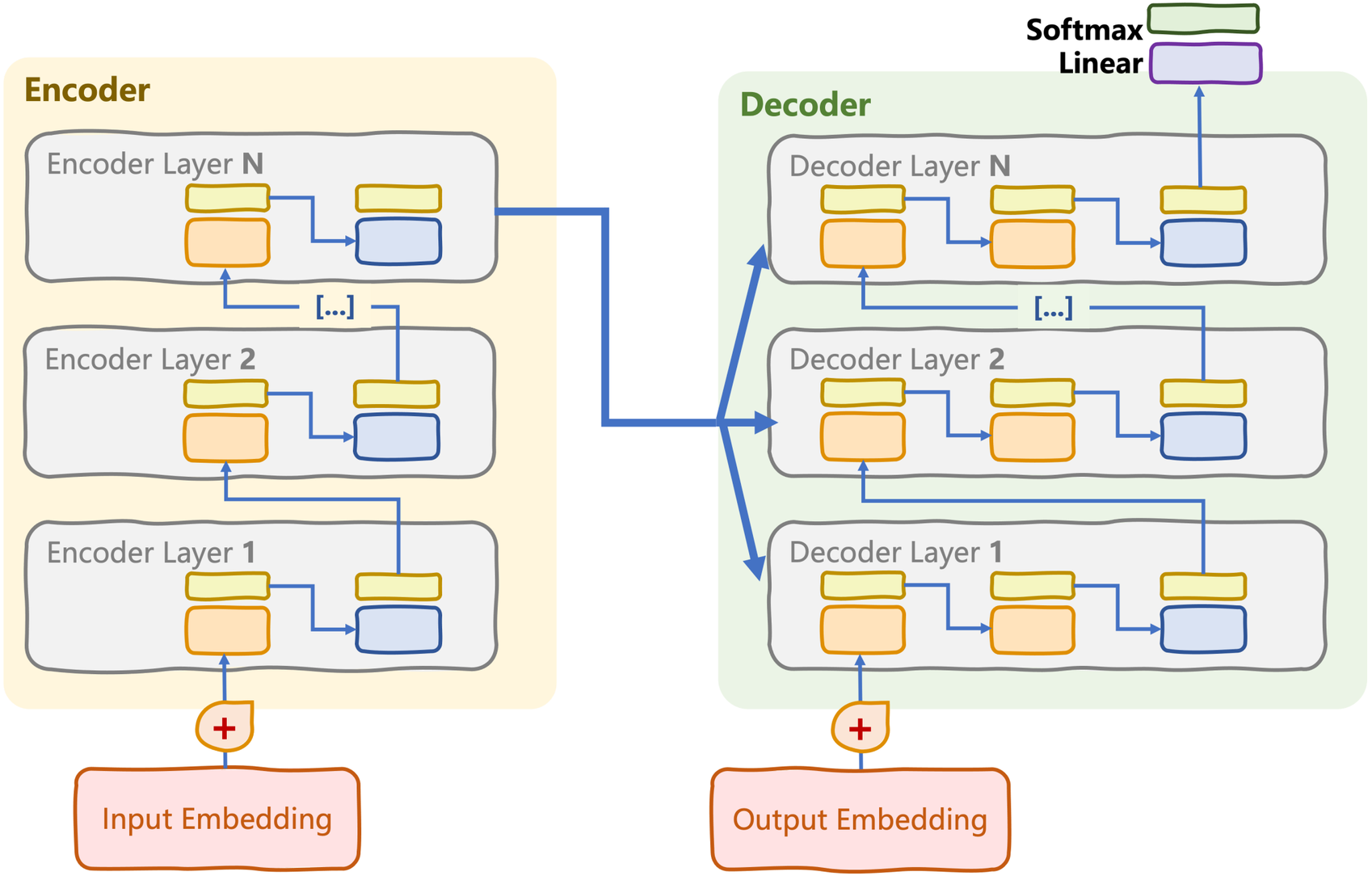
Transformer head



- jugar (*to play*): 0.03
- viajar (*to travel*): **0.96**
- dormir (*to sleep*): 0.01



Everything brought together!



Everything brought together!

```
class PositionalEncoding(nn.Module):
  ...
class MultiHeadAttention(nn.Module):
  ...
class FeedForwardSubLayer(nn.Module):
  ...
class EncoderLayer(nn.Module):
  ...
class DecoderLayer(nn.Module):
  ...
```

```
class TransformerEncoder(nn.Module):
  ...
class TransformerDecoder(nn.Module):
  ...
class ClassificationHead(nn.Module):
  ...
```

```
class Transformer(nn.Module):
    def __init__(self, vocab_size, d_model, num_heads,
                 num_layers, d_ff, max_seq_len, dropout):
        super(Transformer, self).__init__()

        self.encoder = TransformerEncoder(vocab_size,
            d_model, num_heads, num_layers, num_heads,
            d_ff, max_seq_len, dropout)
        self.decoder = TransformerDecoder(vocab_size,
            d_model, num_heads, num_layers, num_heads,
            d_ff, max_seq_len, dropout)

    def forward(self, src, src_mask, causal_mask):
        encoder_output = self.encoder(src, src_mask)
        decoder_output = self.decoder(src, encoder_output,
                                      causal_mask, mask)
        return decoder_output
```

Let's practice!

INTRODUCTION TO LLMS IN PYTHON