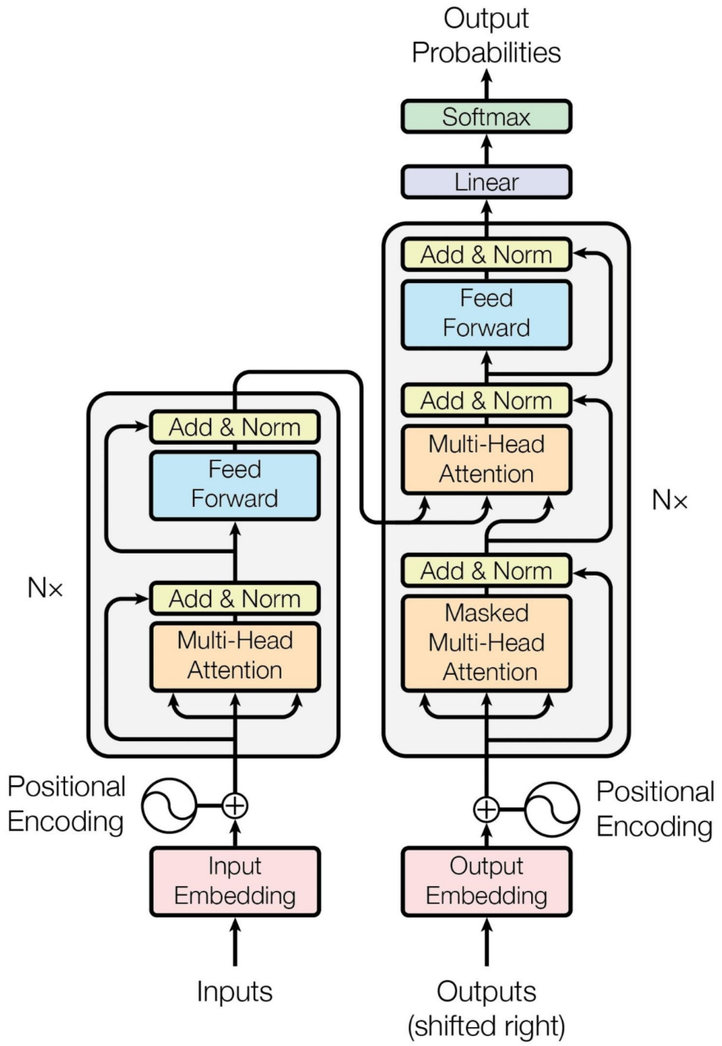
Encoder transformers

TRANSFORMER MODELS WITH PYTORCH

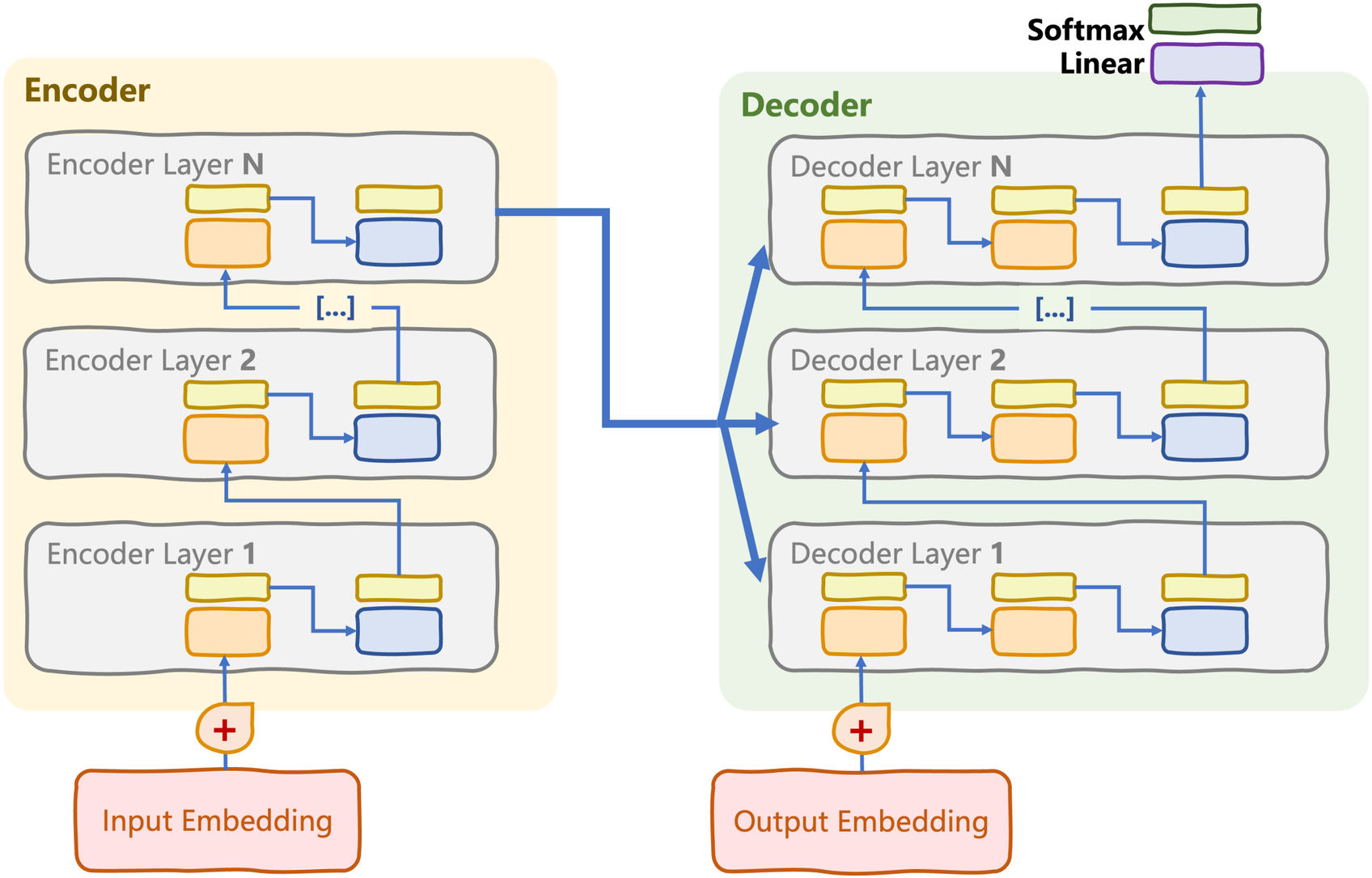
James Chapman
Curriculum Manager, DataCamp

datacamp

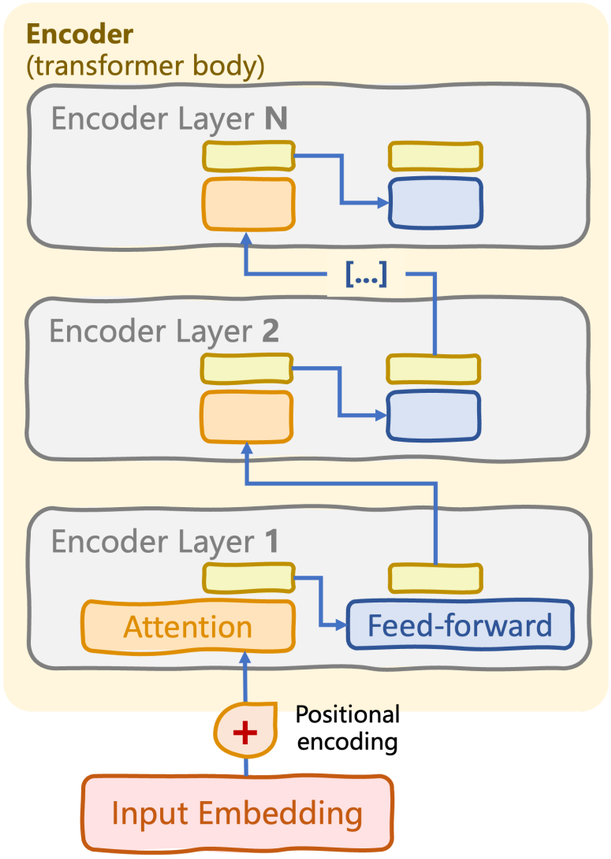
The original transformer



The original transformer



Encoder-only transformers

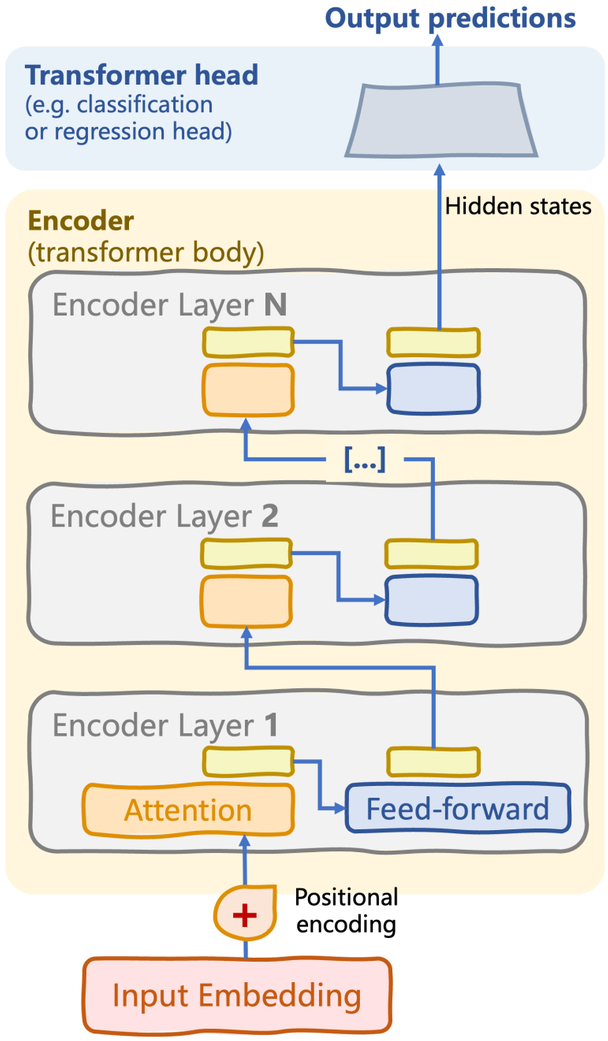


Transformer body: encoder stack with N encoder layers

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

Transformer head

Encoder-only transformers



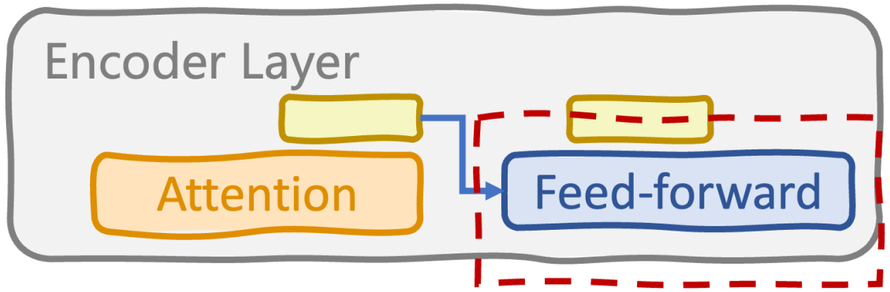
Transformer body: encoder stack with N encoder layers

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

Transformer head: process encoded inputs to produce output prediction

> **Supervised task:** classification, regression

Feed-forward sublayer in encoder layers




```
class FeedForwardSubLayer(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__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)))
```

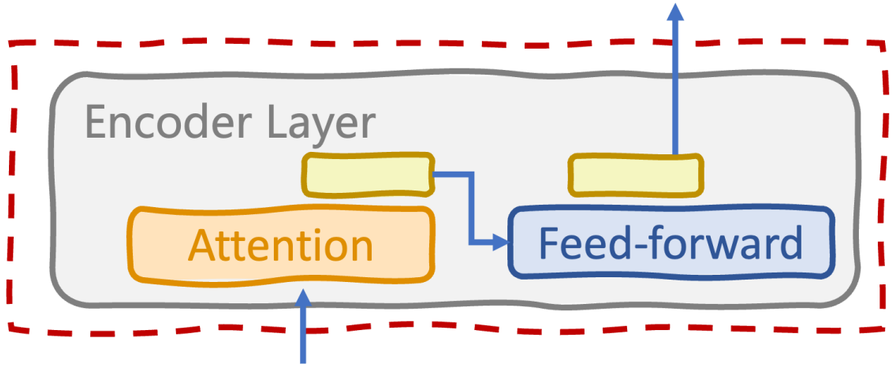
2 x fully connected + ReLU activation

- `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().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.ff_sublayer = 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, src_mask):
        attn_output = self.self_attn(x, x, x, src_mask)
        x = self.norm1(x + self.dropout(attn_output))
        ff_output = self.ff_sublayer(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x
```



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

`forward()`:

- `mask` prevents processing padding tokens

Masking the attention process

"I really like to travel"

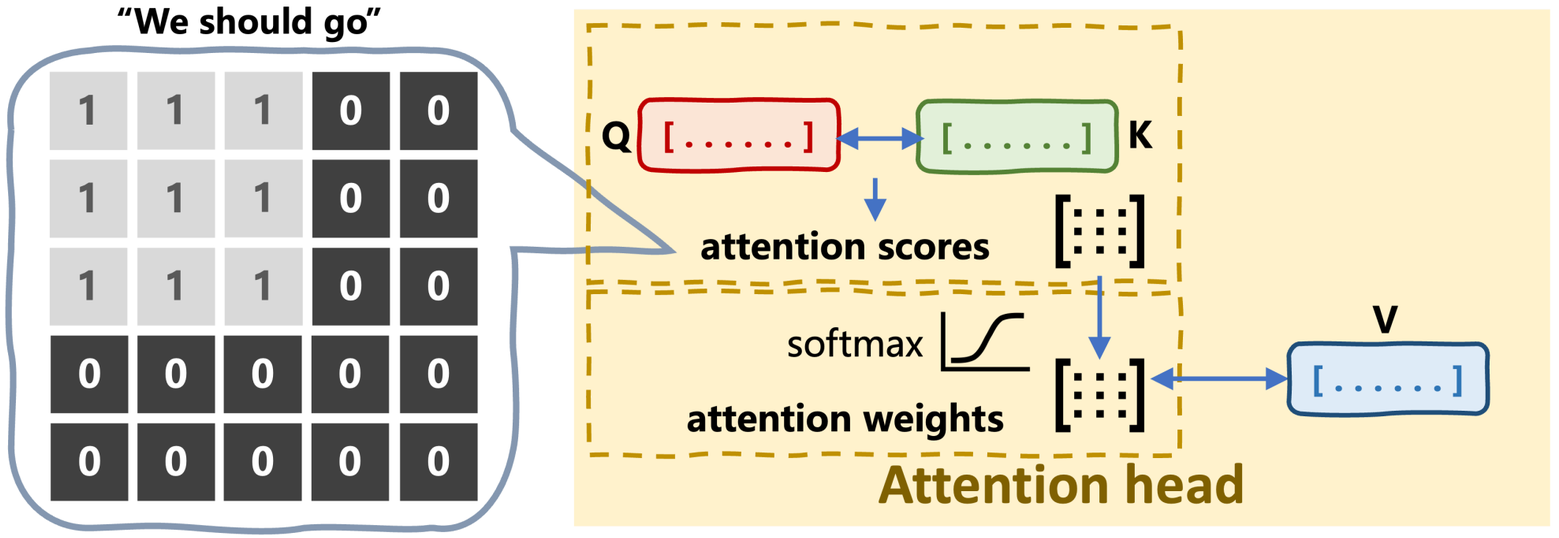
835	8246	94	302	7547

"Eastern Asia looks fascinating"

5839	236	492	28	0

"We should go"

12	641	46	0	0



Encoder transformer body

```
class TransformerEncoder(nn.Module):
    def __init__(self, vocab_size, d_model, num_layers, num_heads, d_ff, dropout, max_seq_length):
        super().__init__()
        self.embedding = InputEmbeddings(vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_seq_length)
        self.layers = nn.ModuleList(
          [EncoderLayer(d_model, num_heads, d_ff, dropout) for _ in range(num_layers)]
        )

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

Encoder transformer head

```
class ClassifierHead(nn.Module):
    def __init__(self, d_model, num_classes):
        super().__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().__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!

TRANSFORMER MODELS WITH PYTORCH

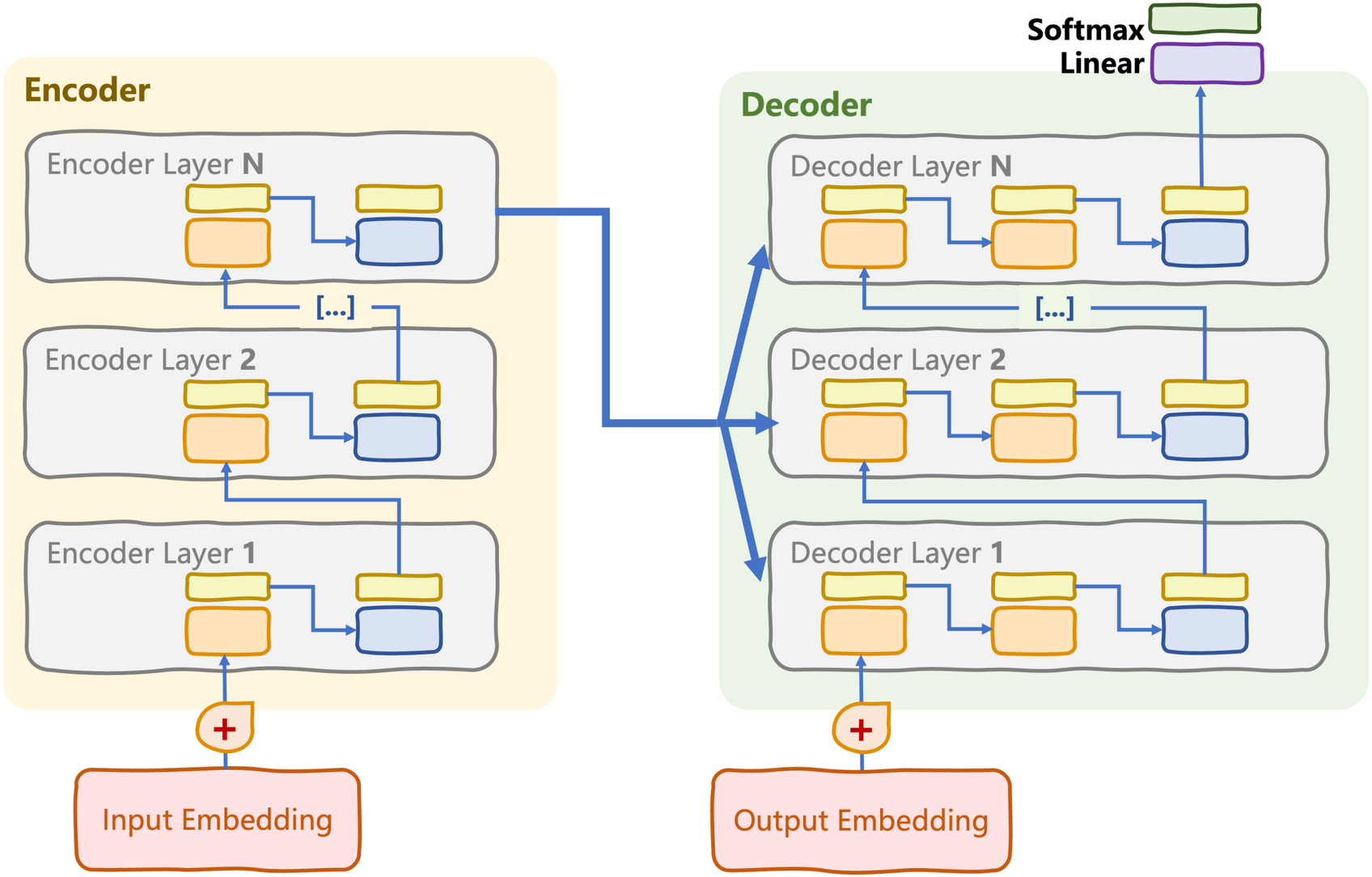
Decoder transformers

TRANSFORMER MODELS WITH PYTORCH

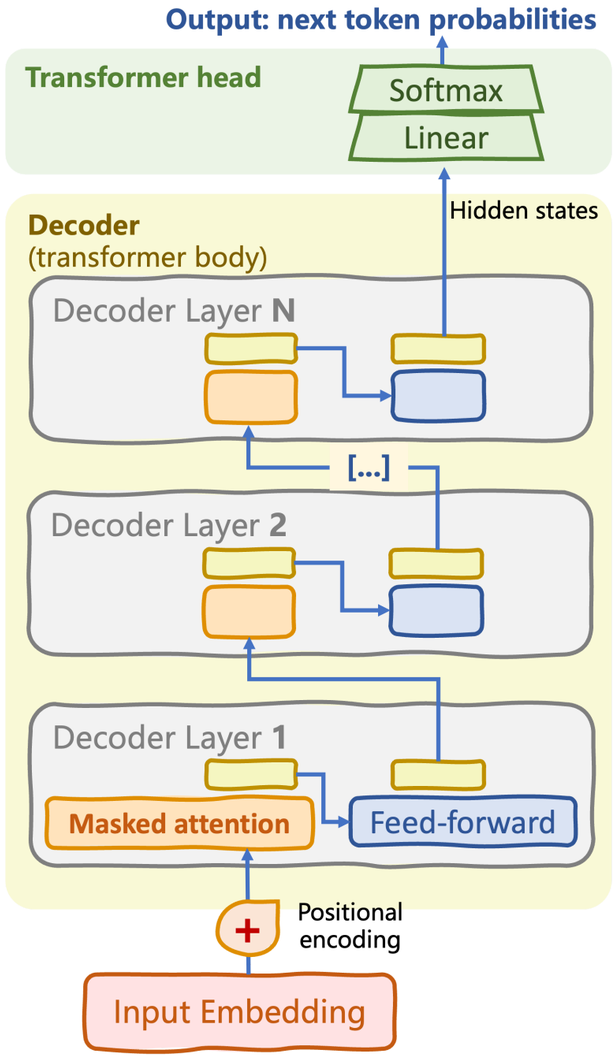
James Chapman
Curriculum Manager, DataCamp

datacamp

From original to decoder-only transformer

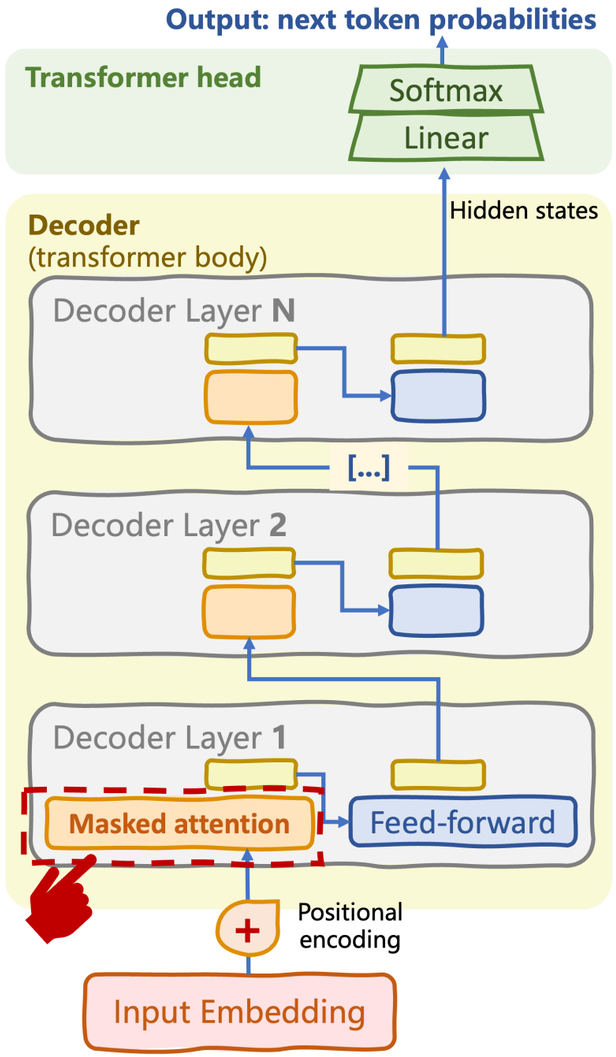


From original to decoder-only transformer



Autoregressive sequence generation: text generation and completion

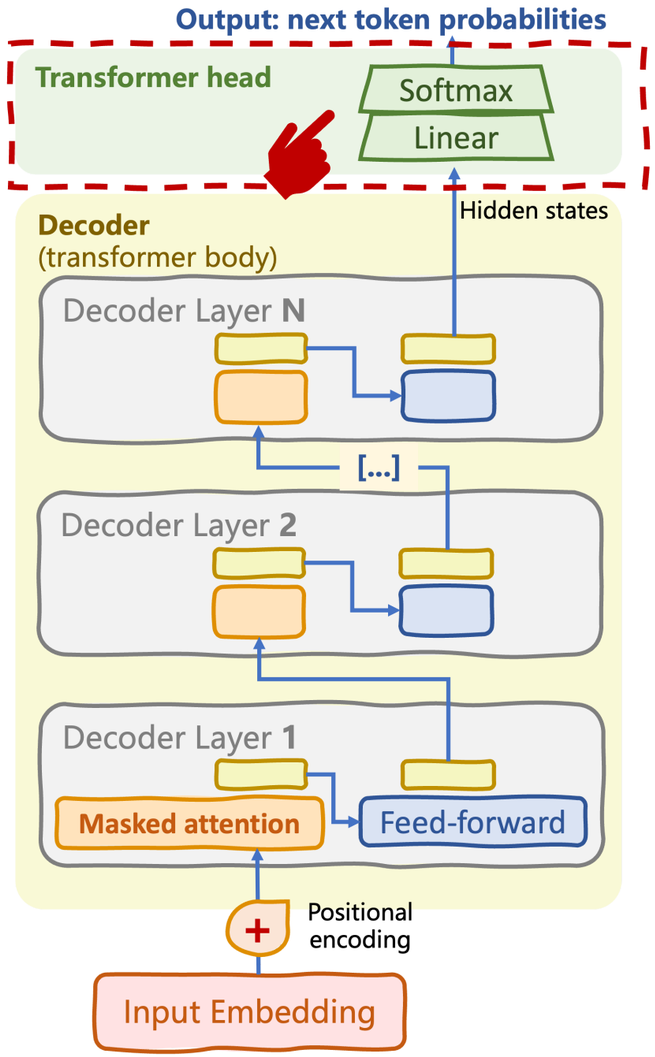
From original to decoder-only transformer



Autoregressive sequence generation: text generation and completion

> **Masked multi-head self-attention**
>
> - Hide later tokens in sequence

From original to decoder-only transformer



Autoregressive sequence generation: text generation and completion

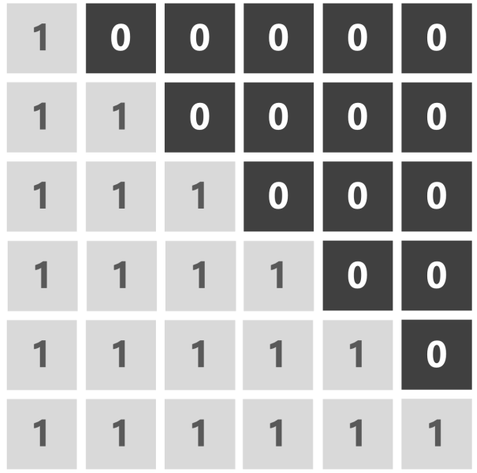
Masked multi-head self-attention

- Hide later tokens in sequence

Decoder-only transformer head

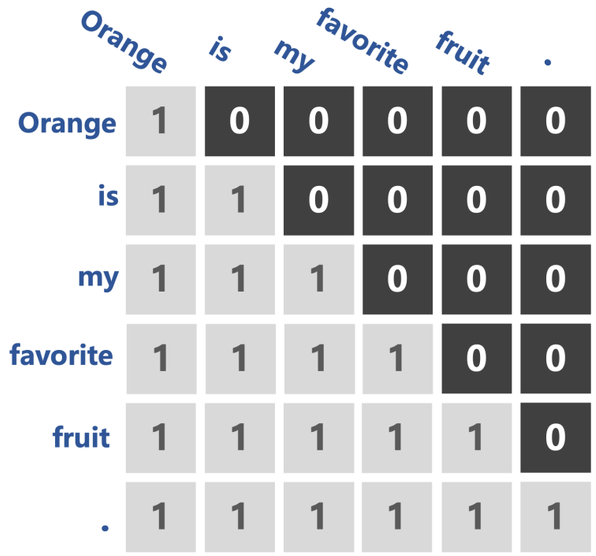
- Linear + Softmax over vocabulary
- Predict most *likely* next tokens

Masked self-attention/causal attention



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

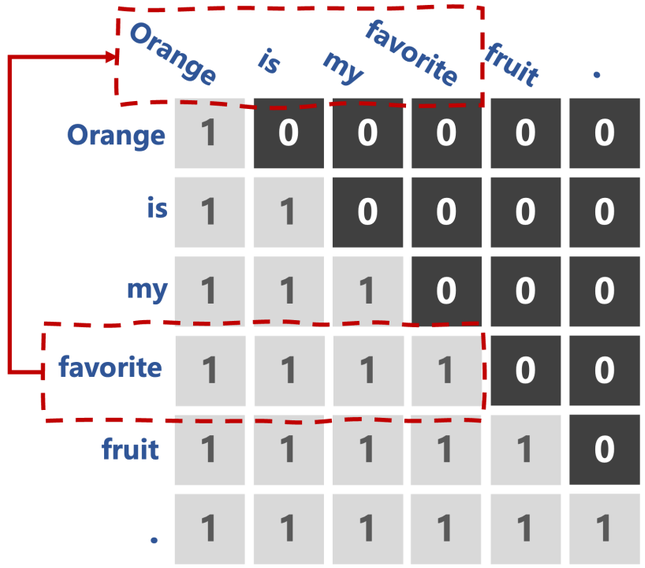
Masked self-attention/causal attention



	Orange	is	my	favorite	fruit	.
Orange	1	0	0	0	0	0
is	1	1	0	0	0	0
my	1	1	1	0	0	0
favorite	1	1	1	1	0	0
fruit	1	1	1	1	1	0
.	1	1	1	1	1	1

- Key to **autoregressive** or **causal** behavior
- Triangular (causal) attention mask
- Token only pays **attention to prior tokens** in the sequence

Masked self-attention/causal attention



- Key to **autoregressive** or **causal** behavior
- Triangular (causal) attention mask
- Token only pays **attention to prior tokens** in the sequence
  - *"**favorite**": "orange", "is", "my", "favorite"*
- Enforced causal attention: predict next word to generate, e.g., *"fruit"*

```
tgt_mask = (1 - torch.triu(
  torch.ones(1, seq_len, seq_len), diagonal=1)
).bool()
```

Decoder layer

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.ff_sublayer = 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, tgt_mask):
        attn_output = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(attn_output))
        ff_output = self.ff_sublayer(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x
```

Decoder transformer body and head

```
class TransformerDecoder(nn.Module):
    def __init__(self, vocab_size, d_model, num_layers, num_heads, d_ff, dropout, max_seq_length):
        super(TransformerDecoder, self).__init__()
        self.embedding = InputEmbeddings(vocab_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model, max_seq_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, tgt_mask):
        x = self.embedding(x)
        x = self.positional_encoding(x)
        for layer in self.layers:
            x = layer(x, tgt_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

Instantiating the decoder-only transformer

```
decoder = TransformerDecoder(vocab_size, d_model, num_layers, num_heads, d_ff, dropout, max_seq_length=seq_length)
output = decoder(input_sequence, tgt_mask)
```

```
tensor([[[ -9.4692,  -9.8429,  -9.3077,  ...,  -9.9523, -10.2669,  -9.7084],
         [ -9.1556,  -9.6133, -10.0923,  ...,  -9.3810,  -9.0420,  -9.1780],
         ...,
         [ -9.5327, -10.3534,  -9.8443,  ...,  -9.8170,  -8.8491,  -8.8322],
         [ -9.6086,  -9.6336, -10.1595,  ...,  -9.8550,  -9.9955,  -8.7121]],

        [[ -9.5865,  -8.0360,  -8.5056,  ...,  -9.9855,  -9.5677,  -9.0352],
         [ -9.7213,  -8.6451,  -8.3779,  ...,  -9.2994,  -9.2601,  -9.8509],
         ...,
         [ -9.0471,  -9.7410, -10.0160,  ..., -10.0195,  -9.4651,  -8.9605],
         [ -9.5767, -10.2692,  -8.8394,  ...,  -8.3458,  -9.1479, -10.0650]]],
       grad_fn=<LogSoftmaxBackward0>)
```

Let's practice!

TRANSFORMER MODELS WITH PYTORCH

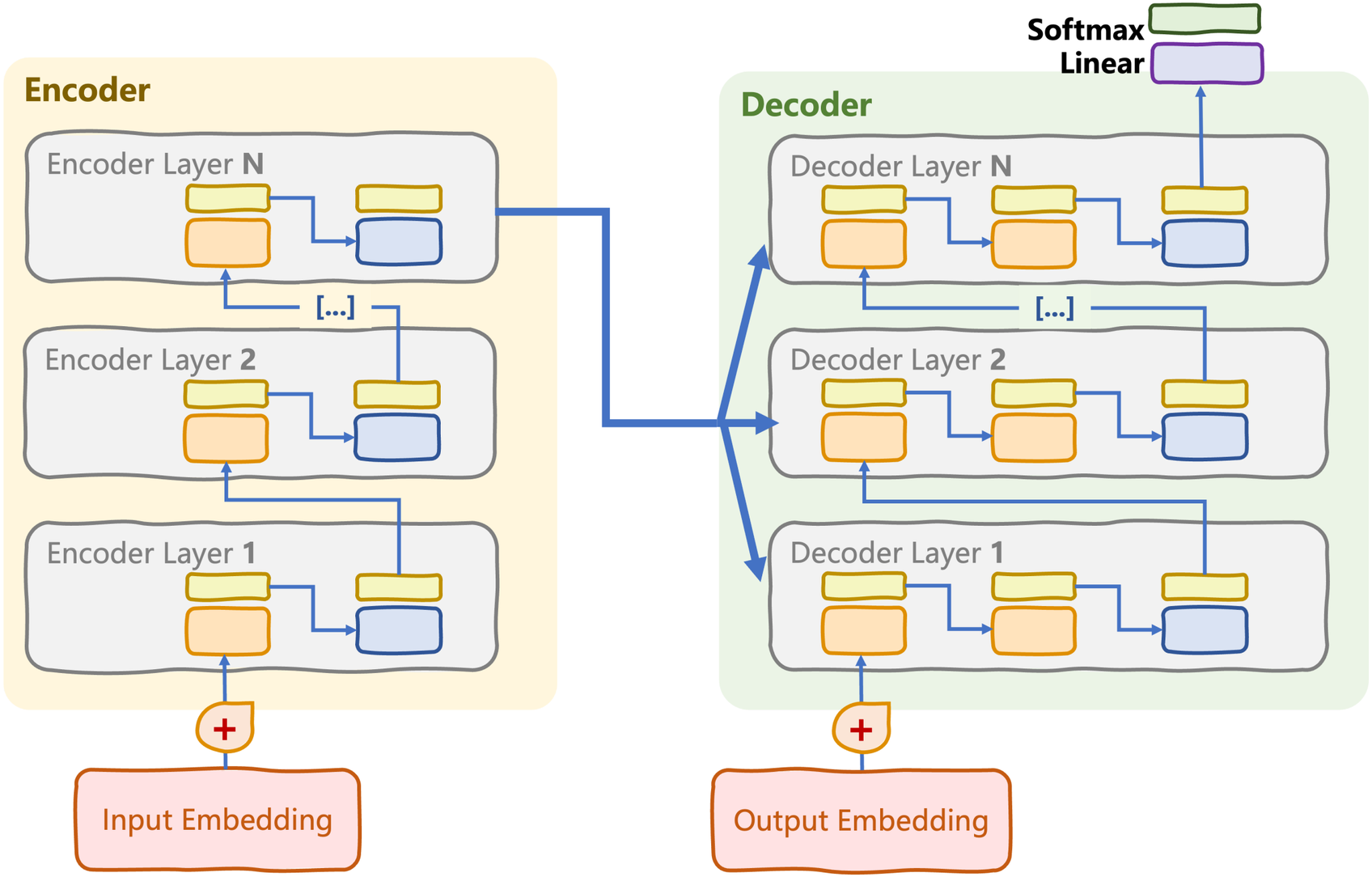
Encoder-decoder transformers

TRANSFORMER MODELS WITH PYTORCH

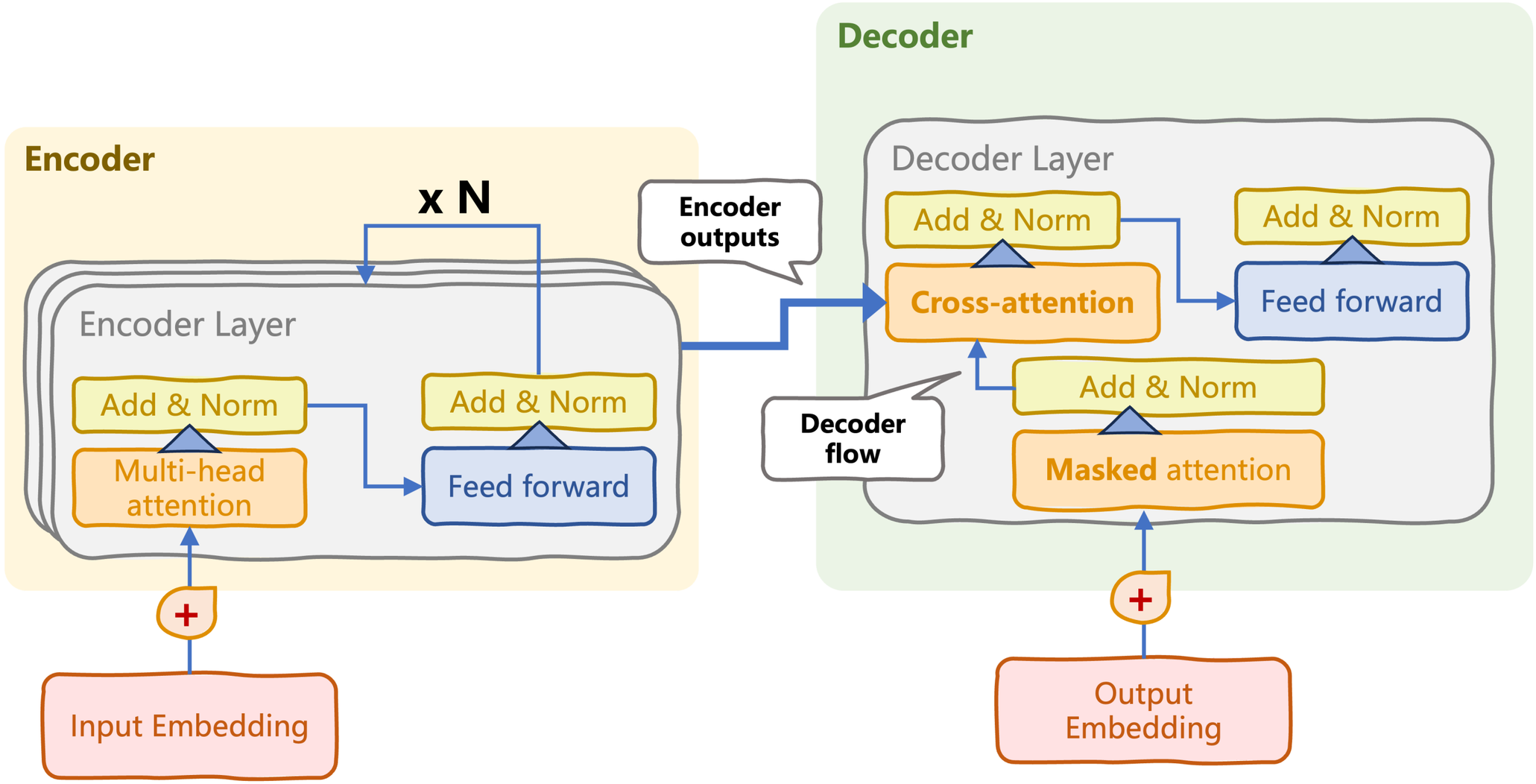
James Chapman
Curriculum Manager, DataCamp

datacamp

Encoder meets decoder

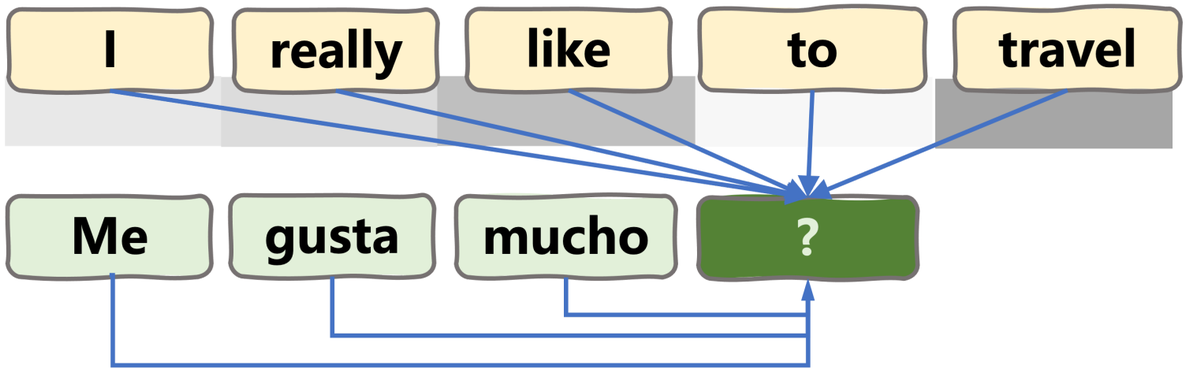


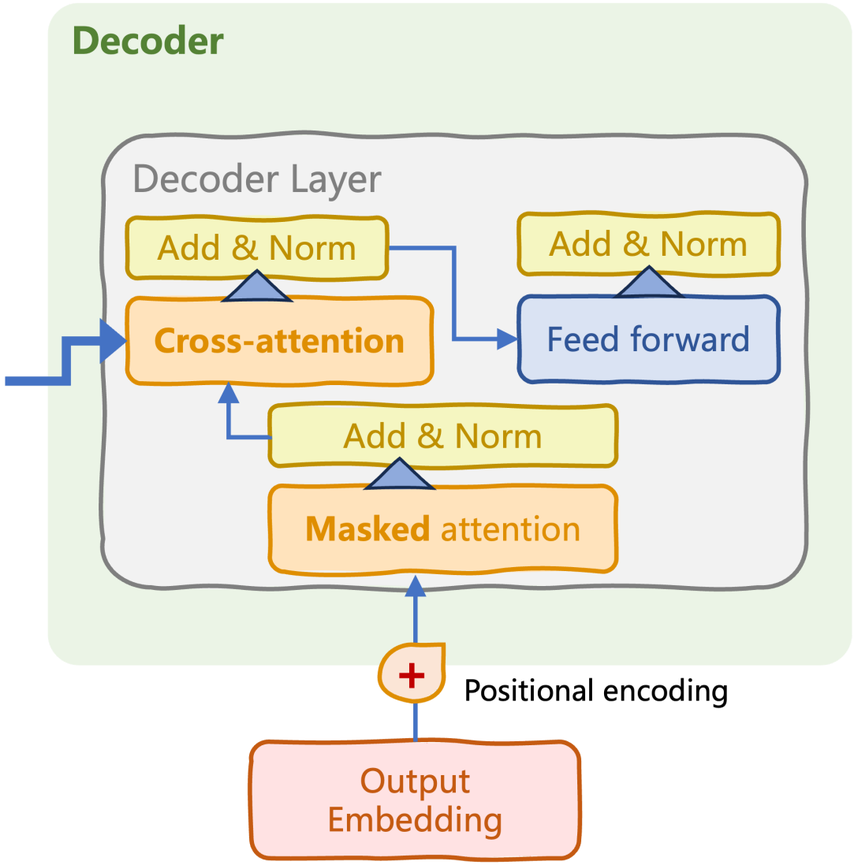
Encoder meets decoder



Cross-attention mechanism

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





Modifying the DecoderLayer

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

- `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().__init__()
        self.self_attn = MultiHeadAttention(
                            d_model, num_heads)
        self.cross_attn = MultiHeadAttention(
                            d_model, num_heads)
        ...

    def forward(self, x, y, tgt_mask, cross_mask):
        self_attn_output = self.self_attn(x, x, x,
                                          tgt_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))
        ...
```

Modifying DecoderTransformer

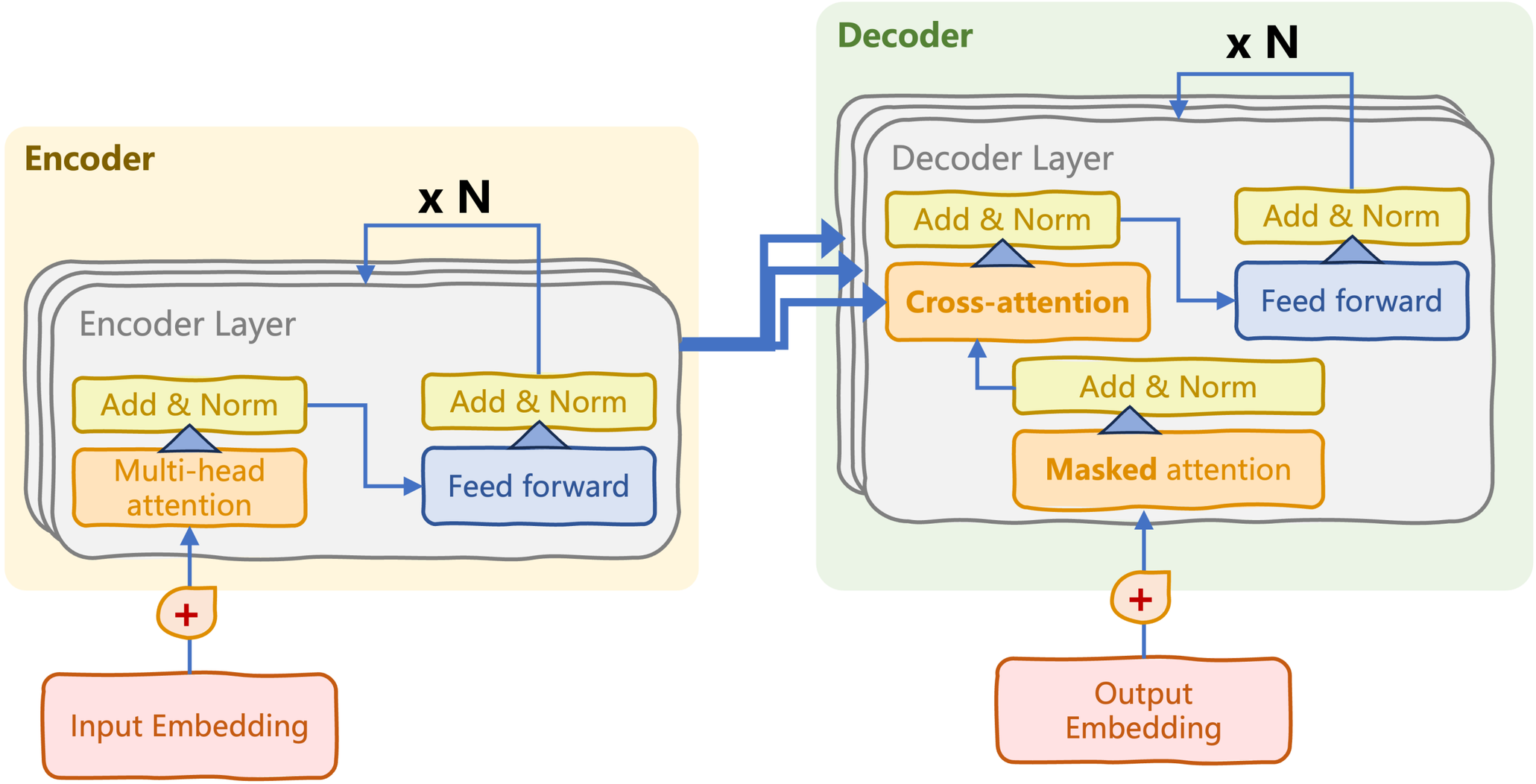
Decoder-only

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

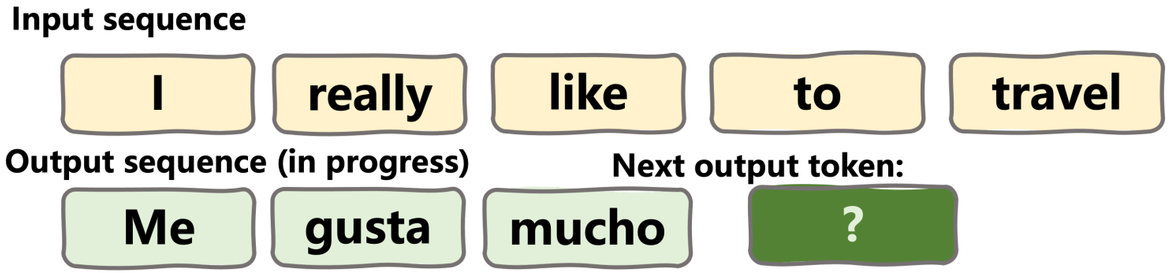
Encoder-decoder

```
class TransformerDecoder(nn.Module):
...
def forward(self, x, y, tgt_mask, cross_mask):
    x = self.embedding(x)
    x = self.positional_encoding(x)
    for layer in self.layers:
        x = layer(x, y, tgt_mask, cross_mask)
    x = self.fc(x)
    return F.log_softmax(x, dim=-1)
```

Encoder meets decoder

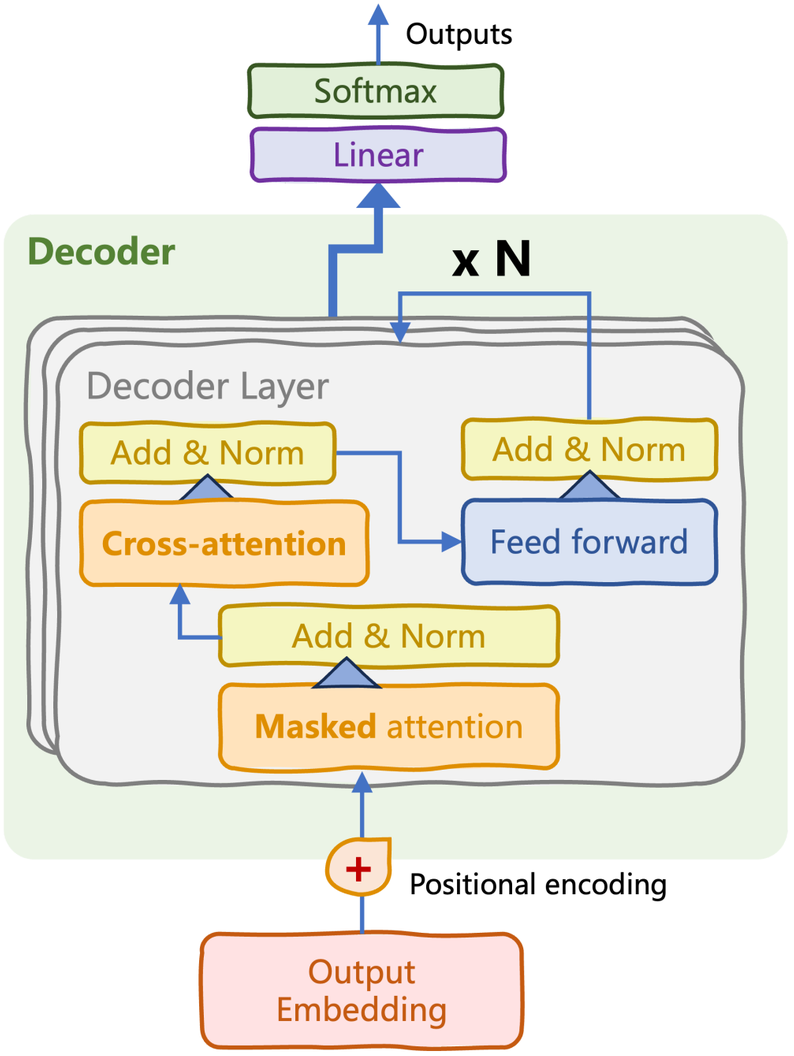


Transformer head

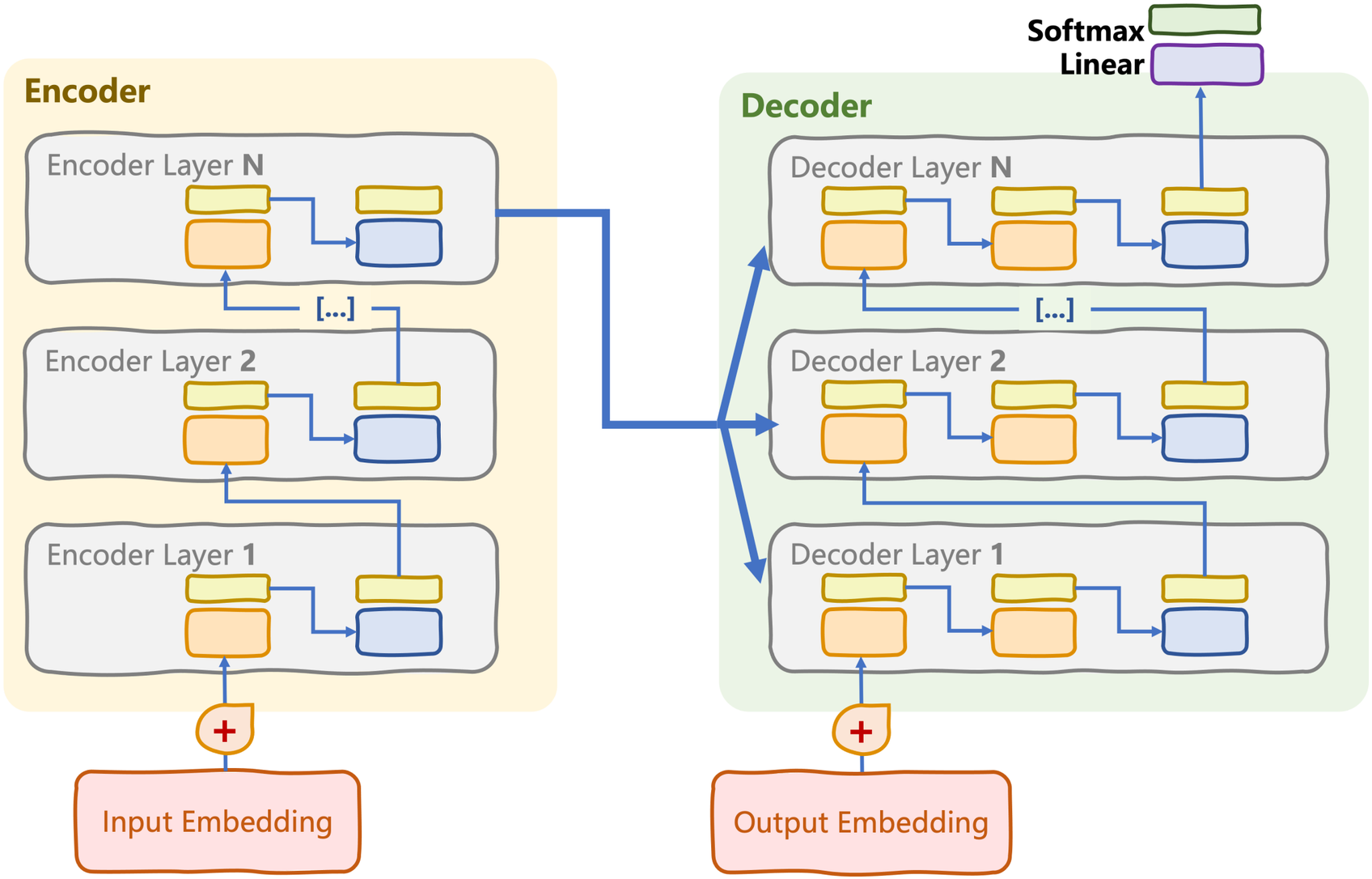


- jugar (*to play*): 0.03
- viajar (*to travel*): **0.96**
- dormir (*to sleep*): 0.01

> For other tasks, different *activations* may be required



Everything brought together!



Everything brought together!

```
class InputEmbeddings(nn.Module):
  ...
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().__init__()

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

    def forward(self, x, src_mask, tgt_mask, cross_mask):
        encoder_output = self.encoder(x, src_mask)
        decoder_output = self.decoder(x, encoder_output,
                                      tgt_mask, cross_mask)
        return decoder_output
```

Let's practice!

TRANSFORMER MODELS WITH PYTORCH

Congratulations!

TRANSFORMER MODELS WITH PYTORCH

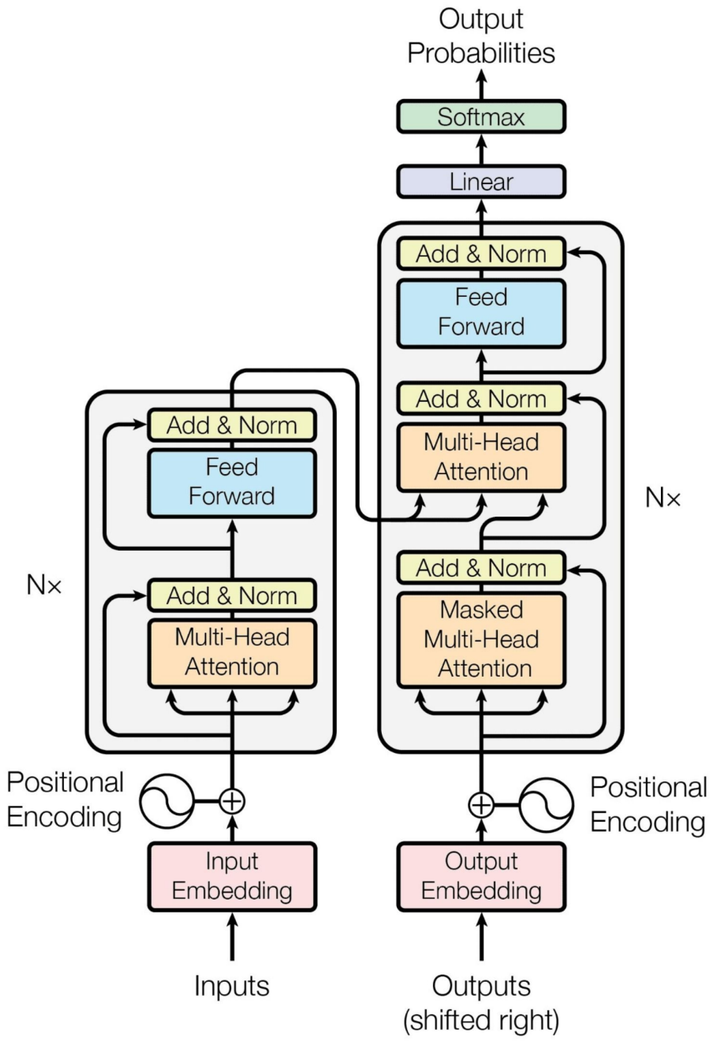
James Chapman
Curriculum Manager, DataCamp

datacamp

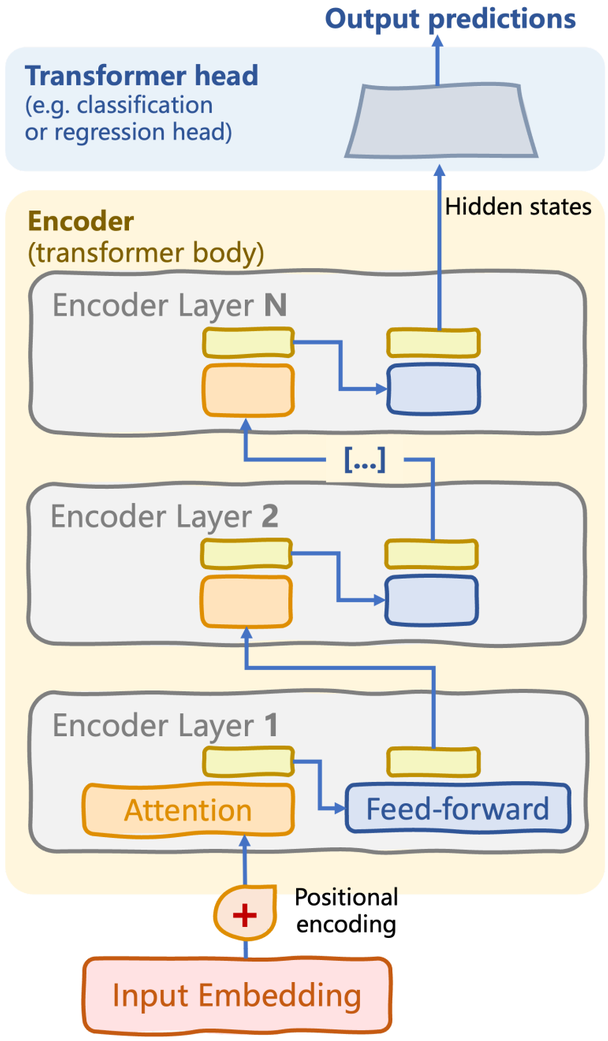
Chapter 1

```
model = nn.Transformer(
    d_model=1536,
    nhead=8,
    num_encoder_layers=6,
    num_decoder_layers=6
)
```

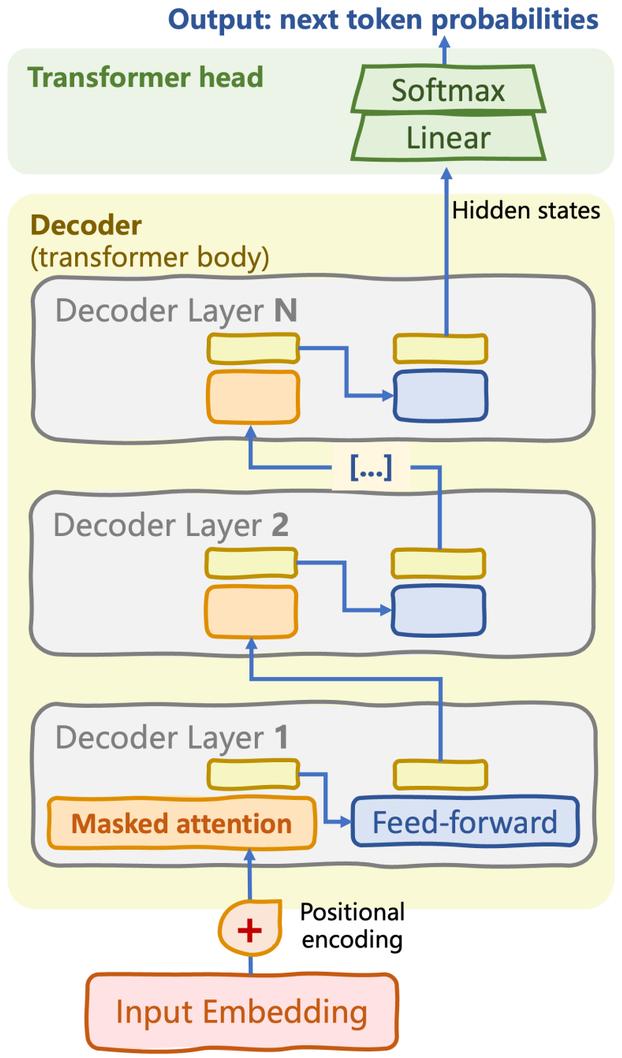
```
class InputEmbeddings(nn.Module): ...
class PositionalEncoding(nn.Module): ...
class MultiHeadAttention(nn.Module): ...
```



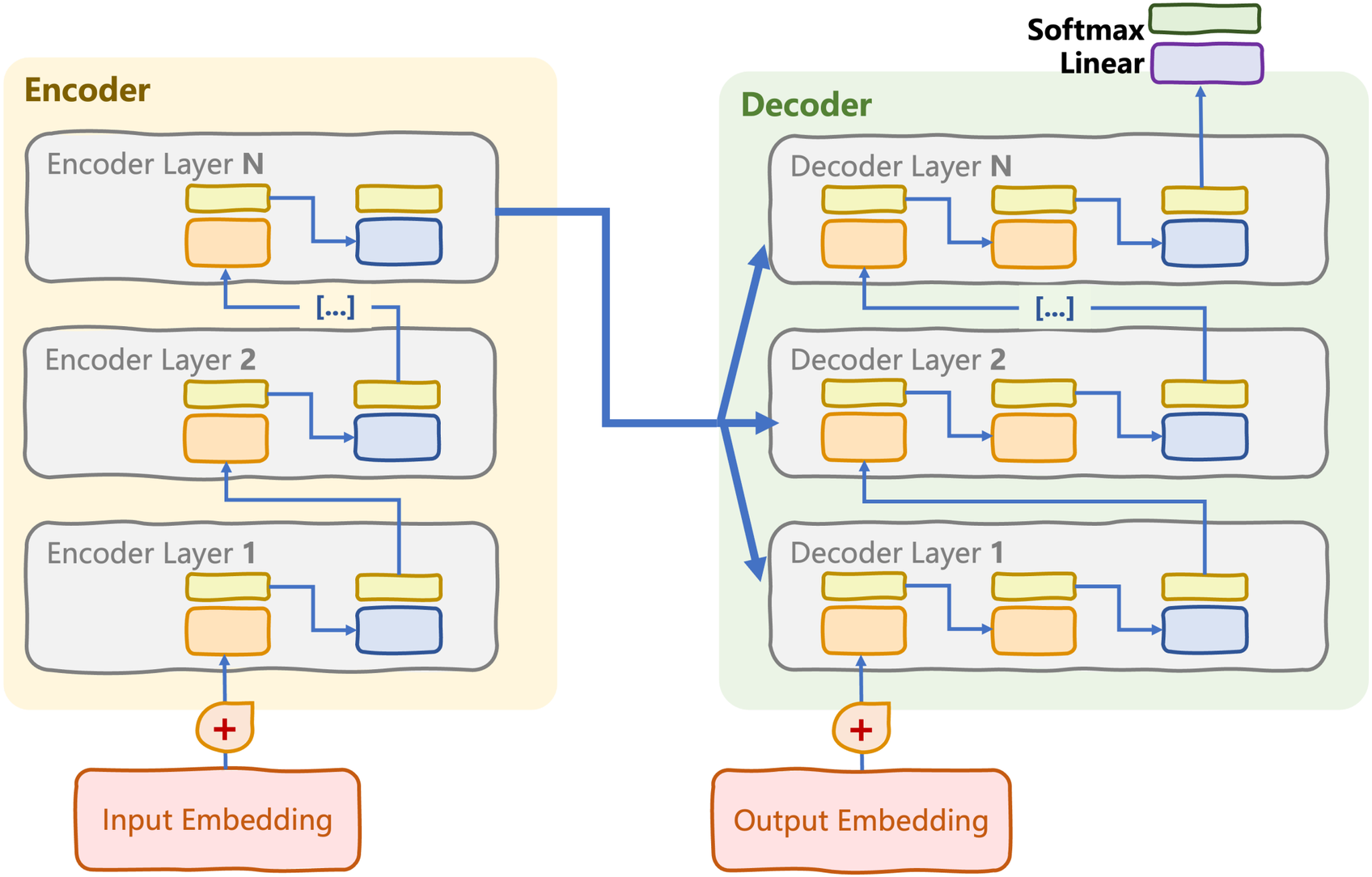
Encoder-only transformer



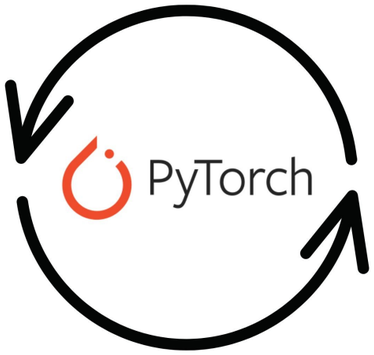
Decoder-only transformer

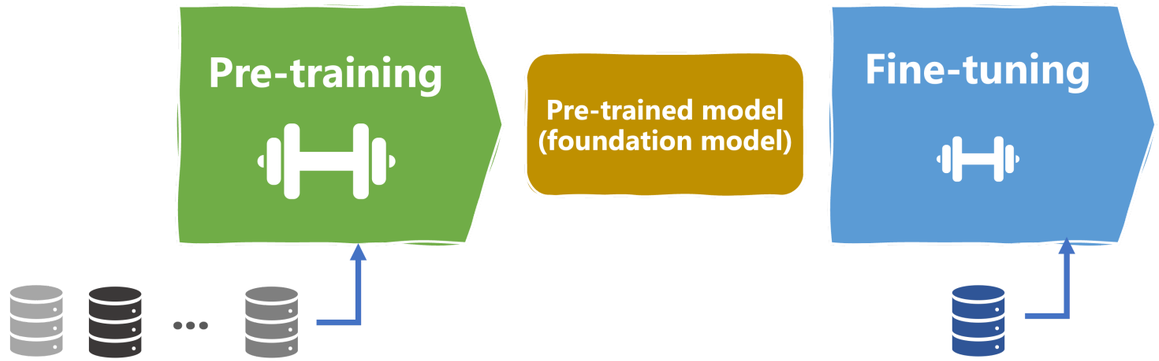


Chapter 2 - Encoder-decoder transformer



What next?





Training

- Distributed AI Model Training in Python

Additional Resources

- Attention Is All You Need

Pre-trained transformers

- Working with Hugging Face
- Introduction to LLMs in Python
- Fine-Tuning with Llama 3

Let's practice!

TRANSFORMER MODELS WITH PYTORCH