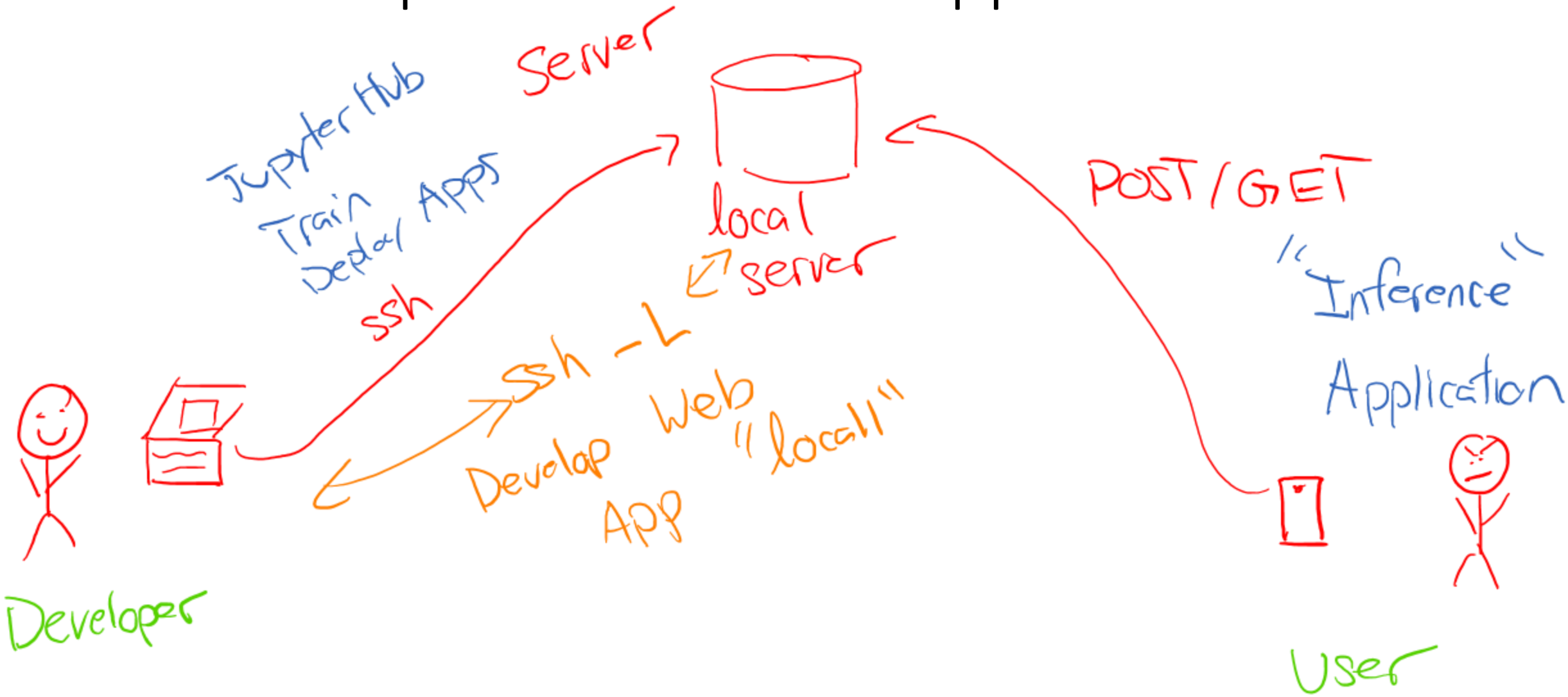
# Attention and Transformers

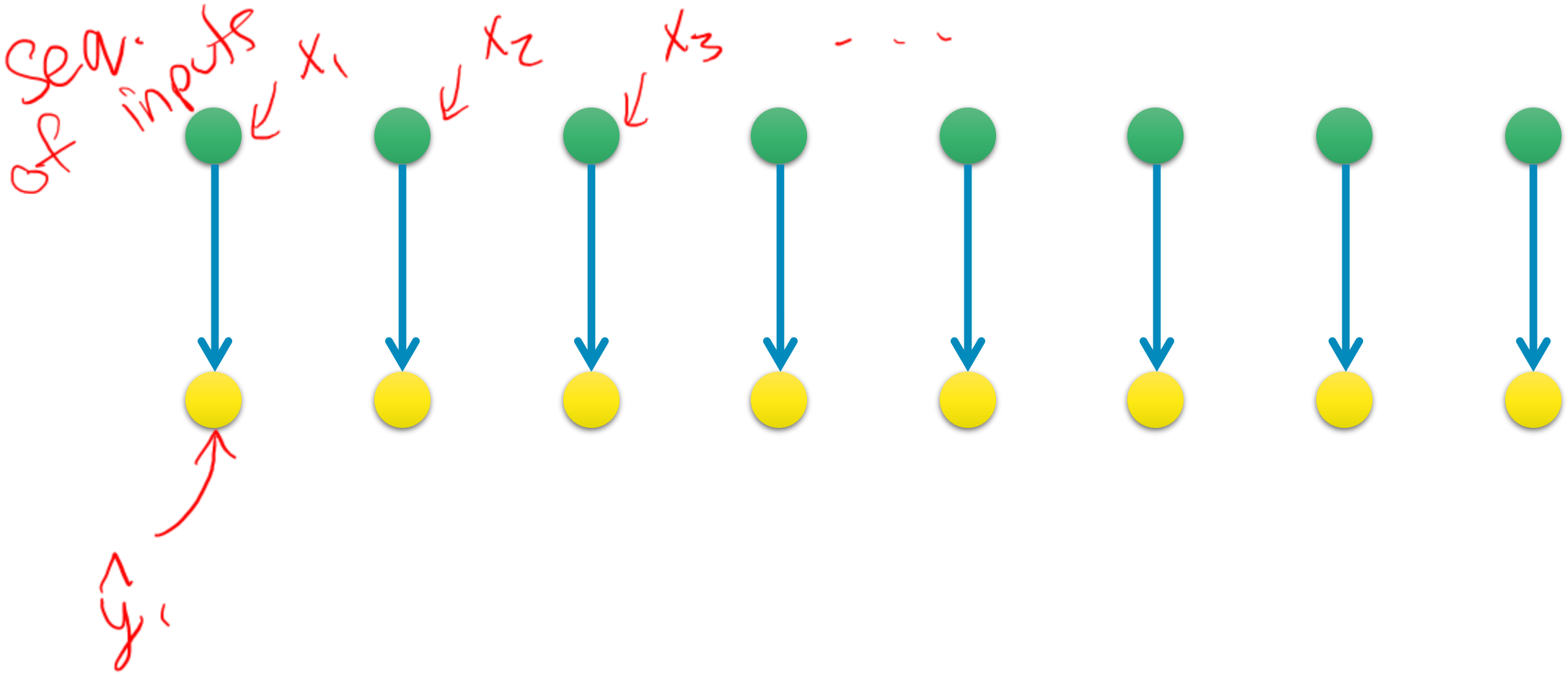
# Recap: Inference and Applications



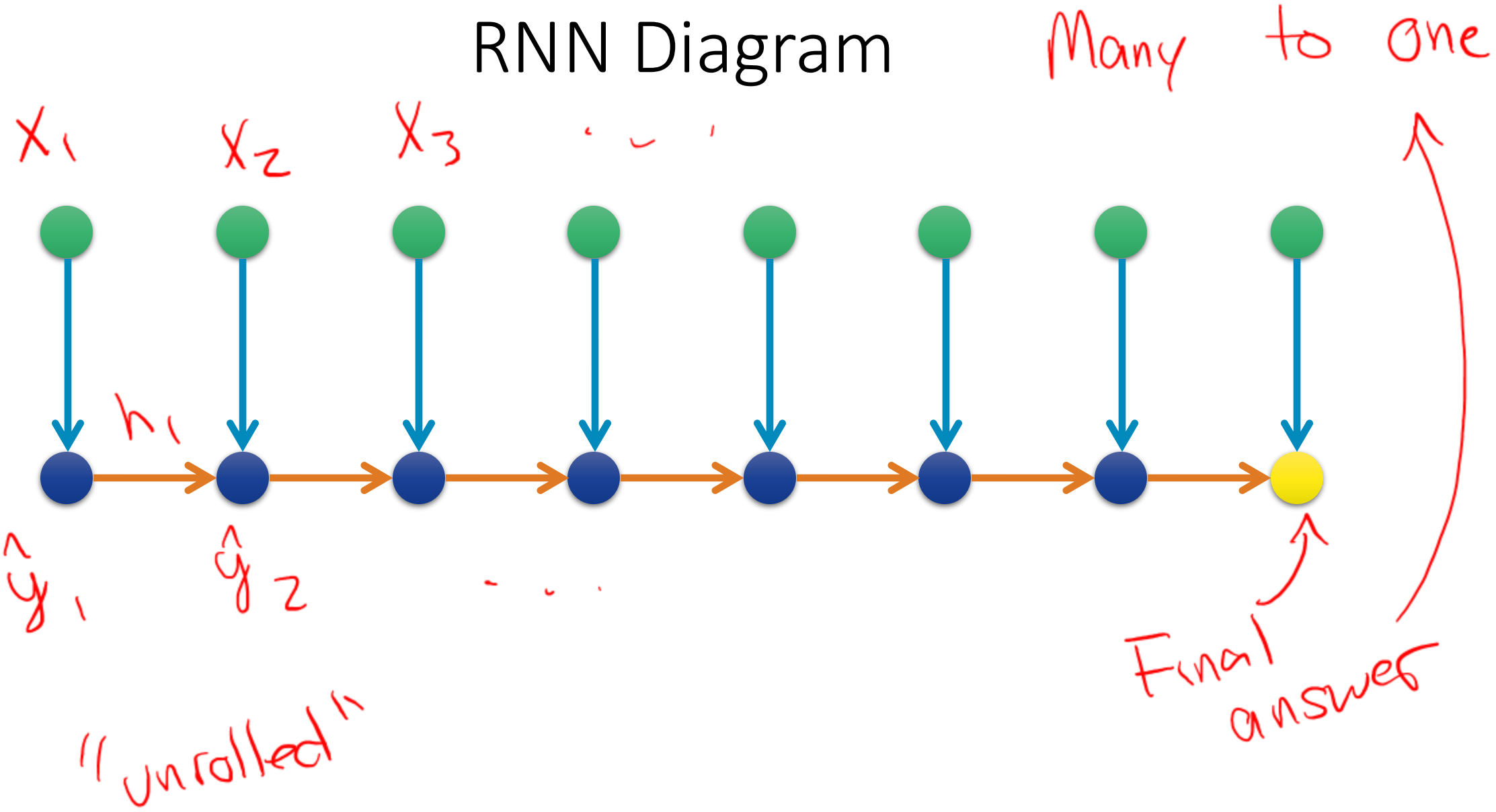
# Outline

- High-level comparison
  - Feed forward networks
  - Recurrent networks
  - Transformer networks
- Remember the RNN encoder/decoder architecture
- Motivate the need for attention
- Explore the transformer architecture

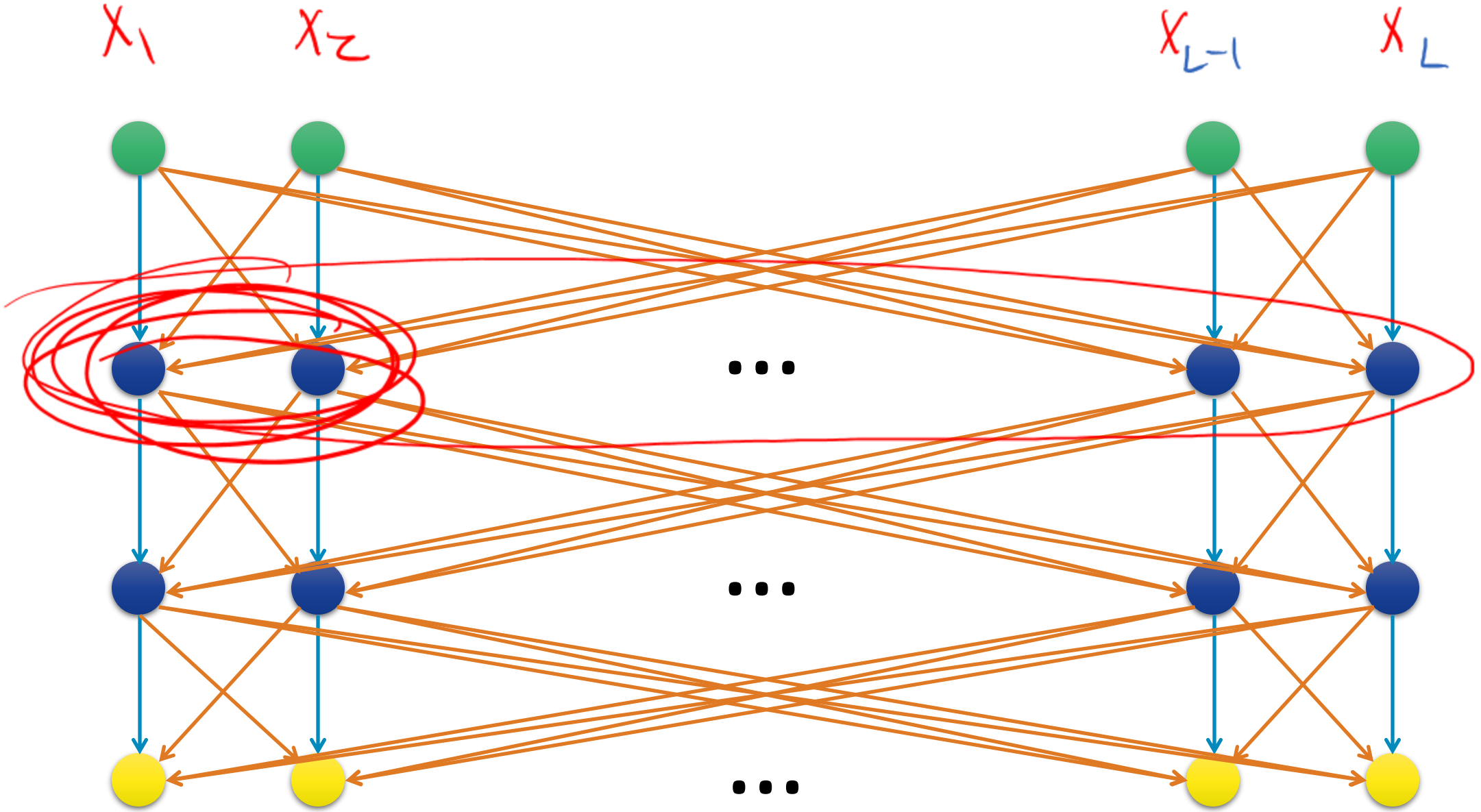
# Feed Forward NN Diagram



# RNN Diagram



# Transformer Diagram



# Outline

- High-level comparison
  - Feed forward networks
  - Recurrent networks
  - Transformer networks

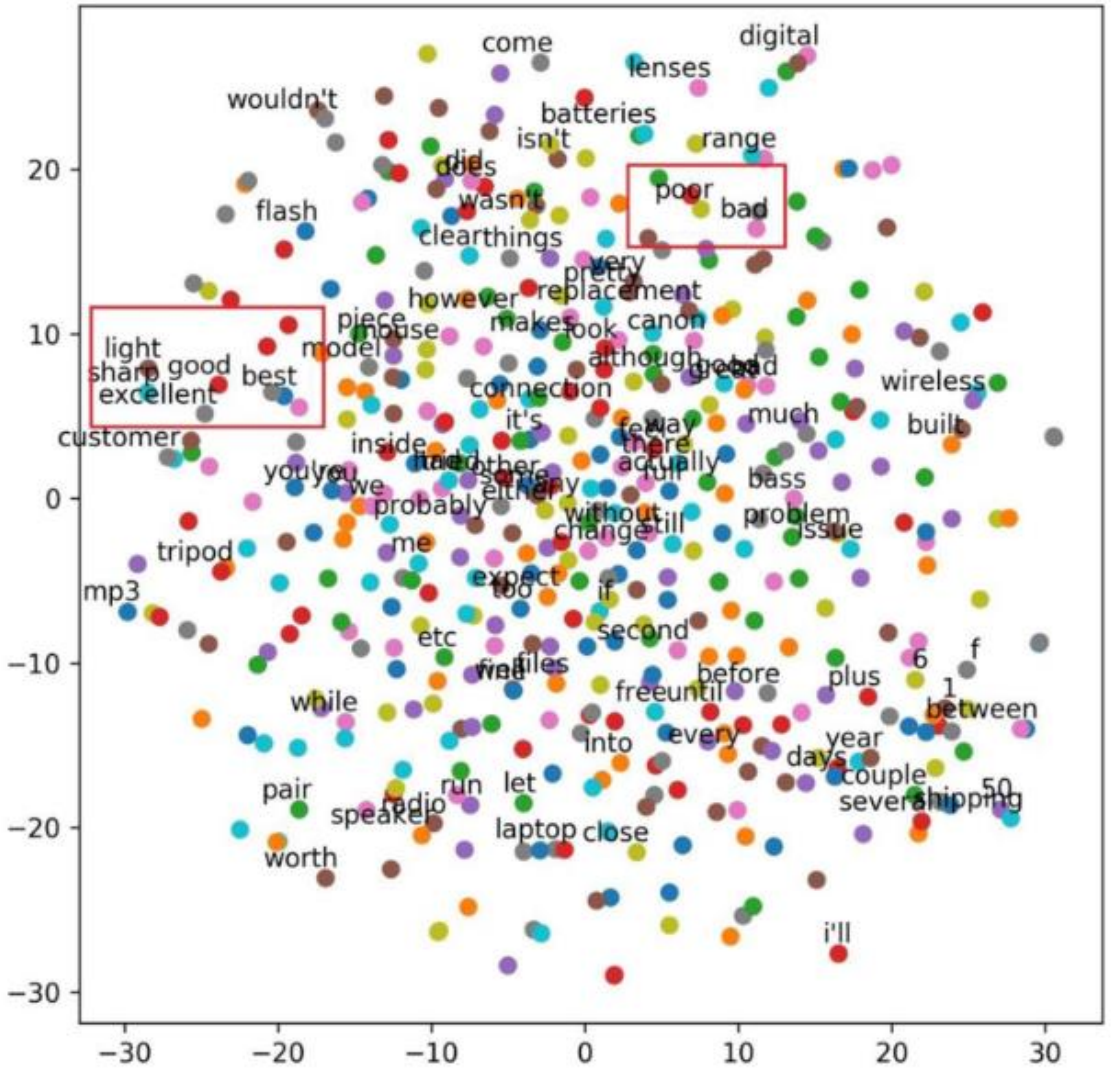
- Remember the RNN encoder/decoder architecture

- Motivate the need for attention

- Explore the transformer architecture

# Translation Encoder/Decoder RNN

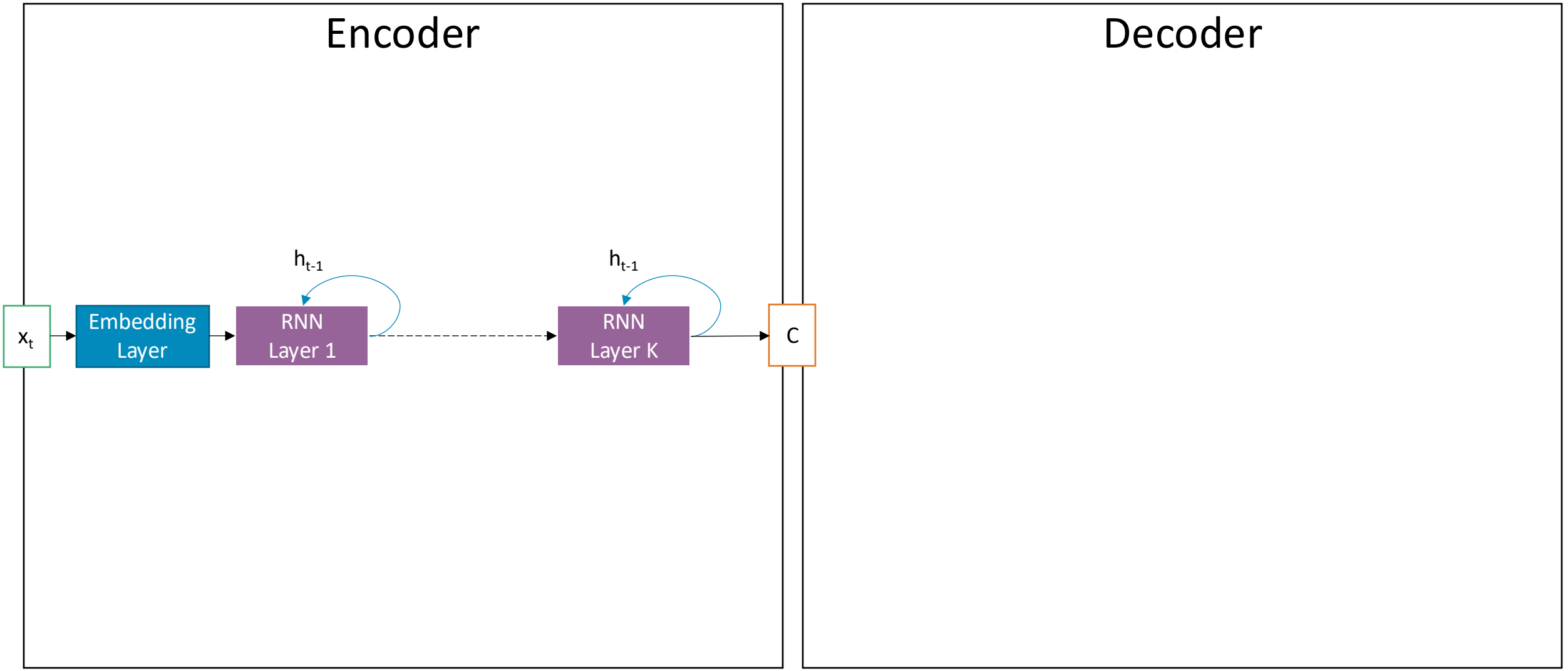
# Embedding Space




**Cross-domain sentiment aware word embeddings for review sentiment analysis**

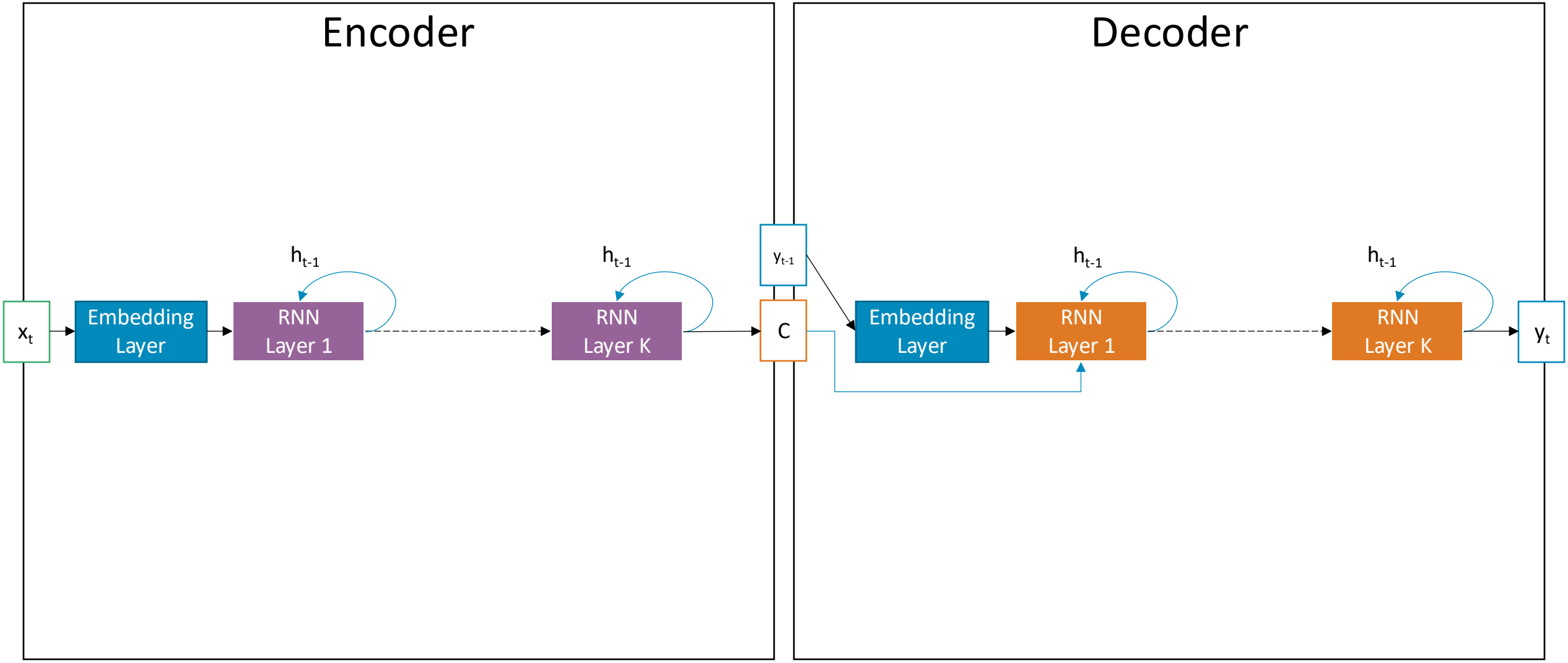
Jun Liu[1] · Shuang Zheng[1] · Guangxia Xu[1,2,3] · Mingwei Lin[4]

# Translation Encoder/Decoder RNN



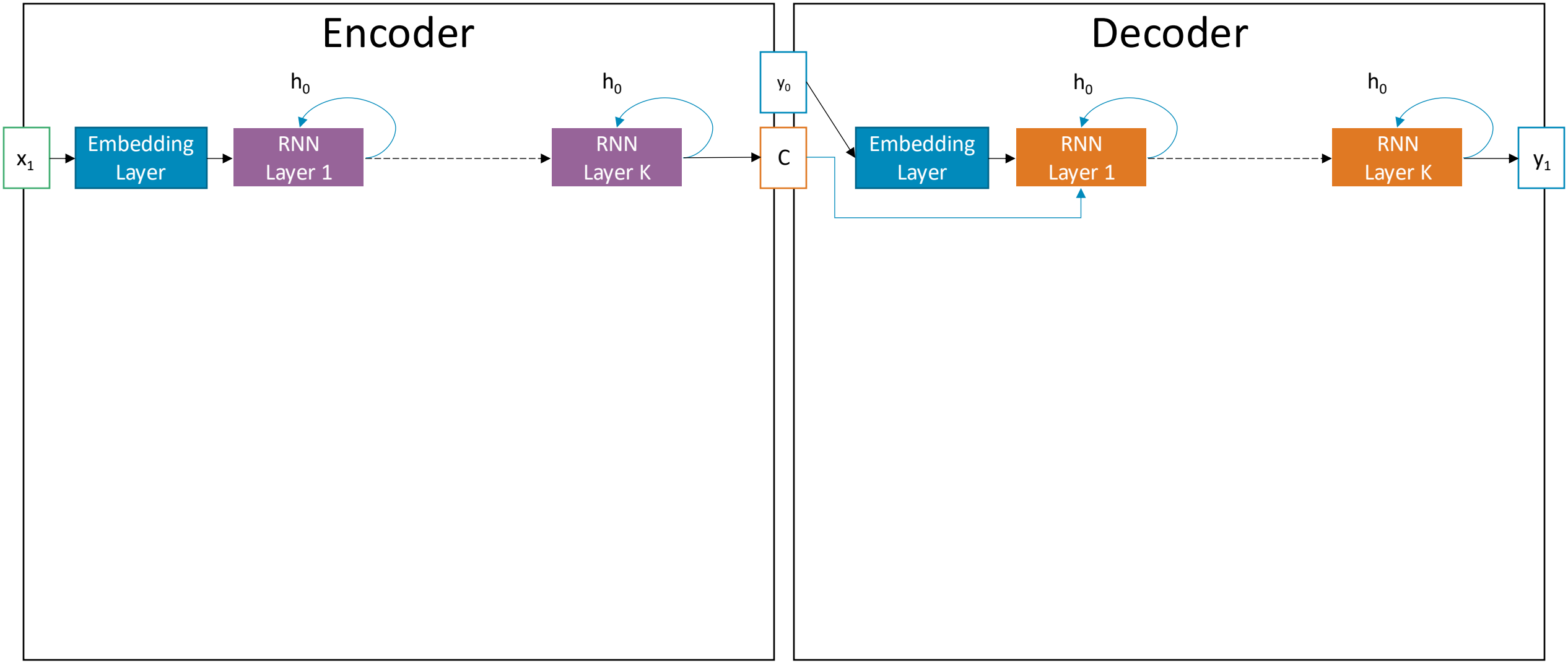

These RNN layers are not show as “unrolled” (we have multiple layers)

# Translation Encoder/Decoder RNN



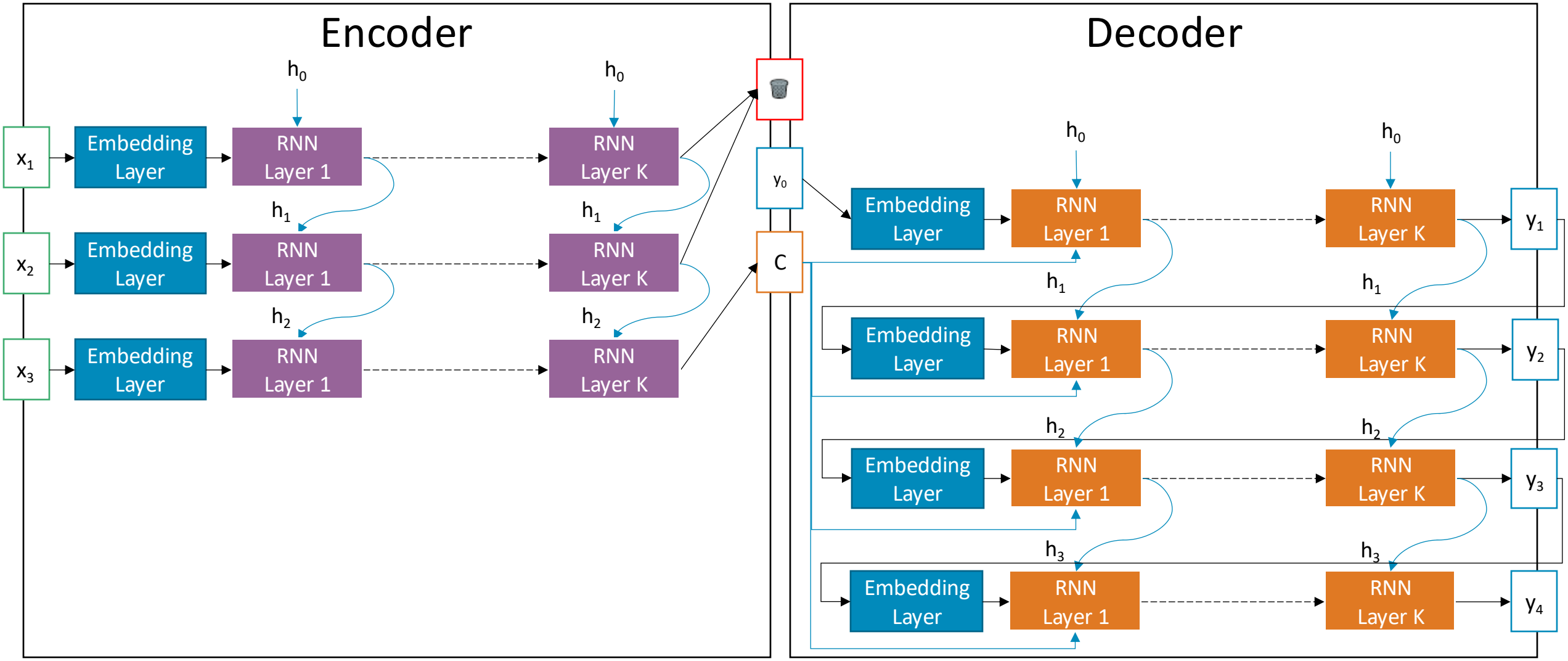

These RNN layers are not show as “unrolled” (we have multiple layers)

# Translation Encoder/Decoder RNN



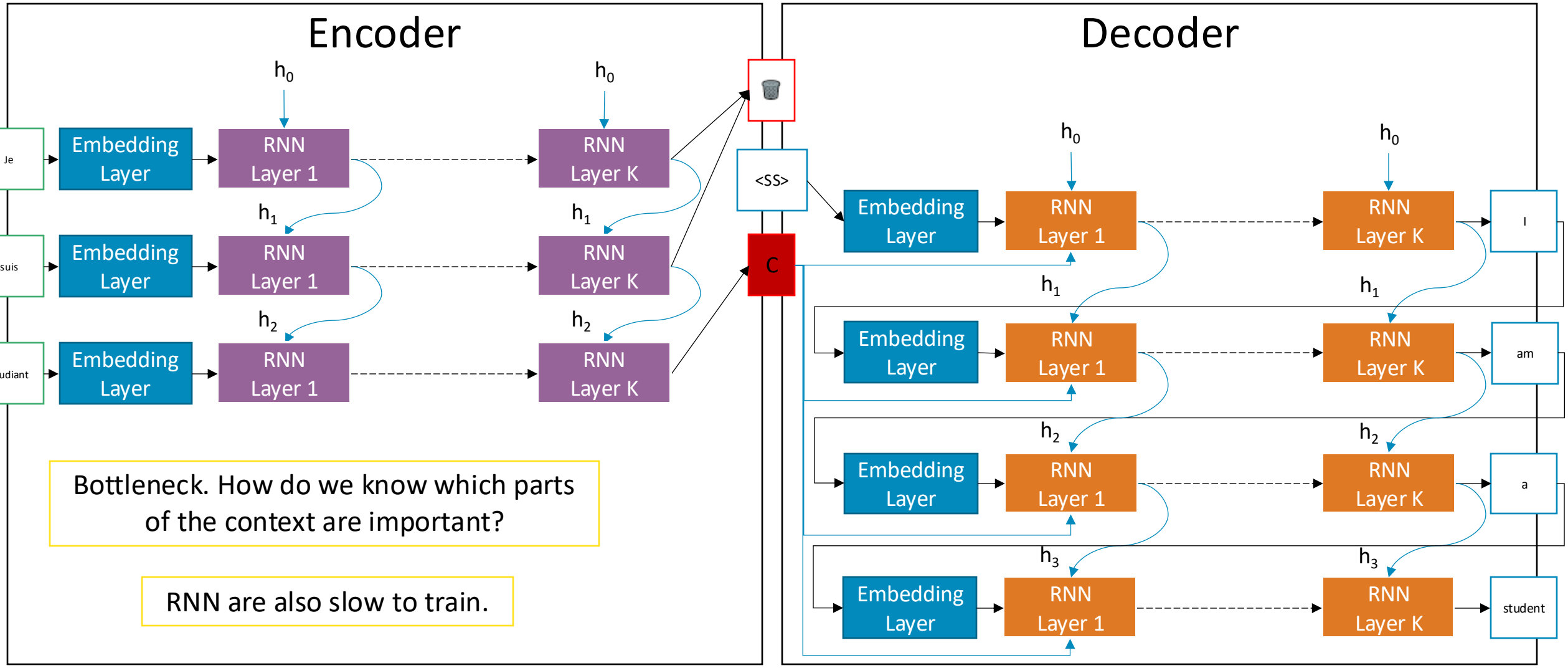
# Translation Encoder/Decoder RNN

Unrolled Version



# Translation Encoder/Decoder RNN

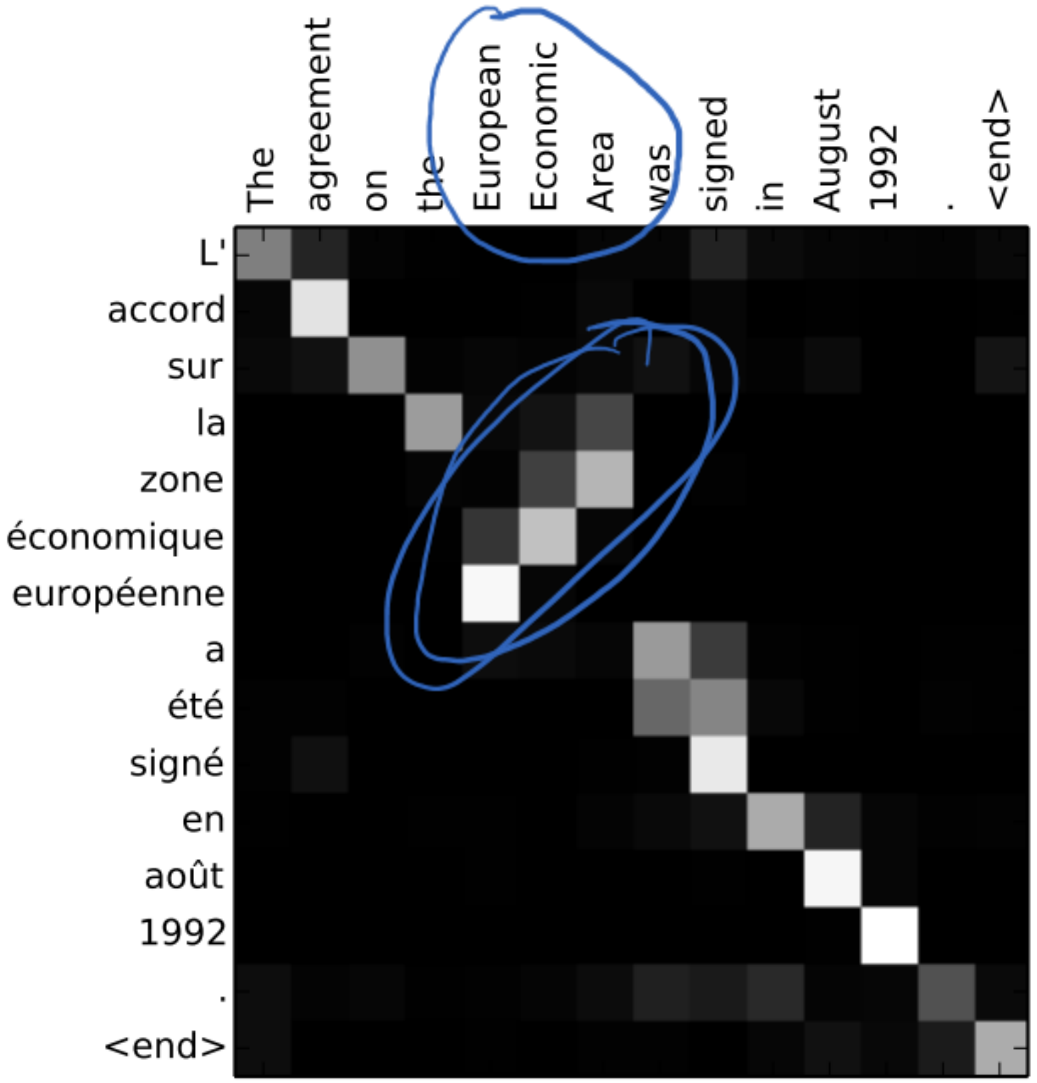
Unrolled Version



# Outline

- High-level comparison
  - Feed forward networks
  - Recurrent networks
  - Transformer networks
- Remember the RNN encoder/decoder architecture
- Motivate the need for attention
- Explore the transformer architecture

# Attention




The agreement on the European Economic Area was signed in August 1992.

L’ accord sur la zone économique européenne a été signé en août 1992.

[Submitted on 1 Sep 2014 (v1), last revised 19 May 2016 (this version, v7)]

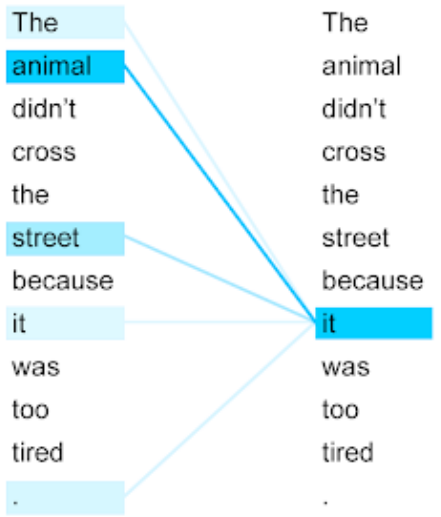
**Neural Machine Translation by Jointly Learning to Align and Translate**

Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio

# Coreference Resolution

The animal didn’t cross the street because it was too tired.

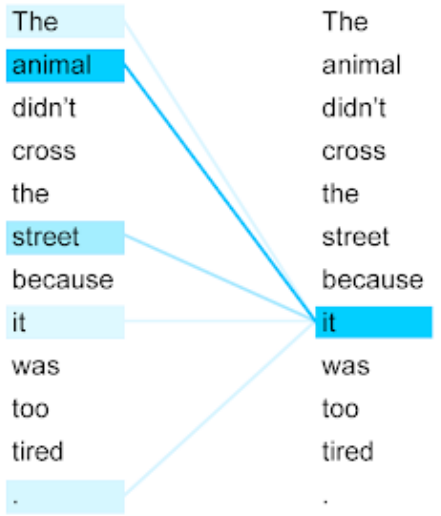
L’animal n’a pas traverse la rue parce qu’il était trop fatigue.



# Coreference Resolution

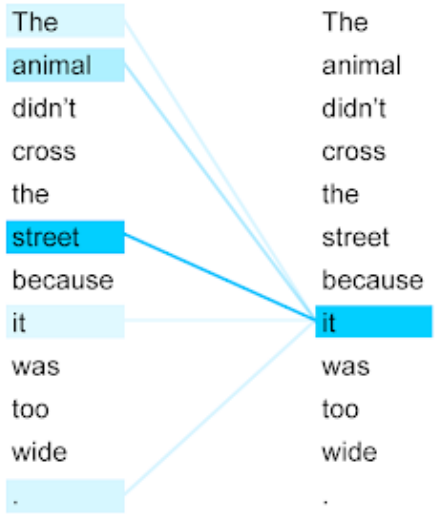
The animal didn't cross the street because it was too tired.

L'animal n'a pas traverse la rue parce qu'il était trop fatigue.



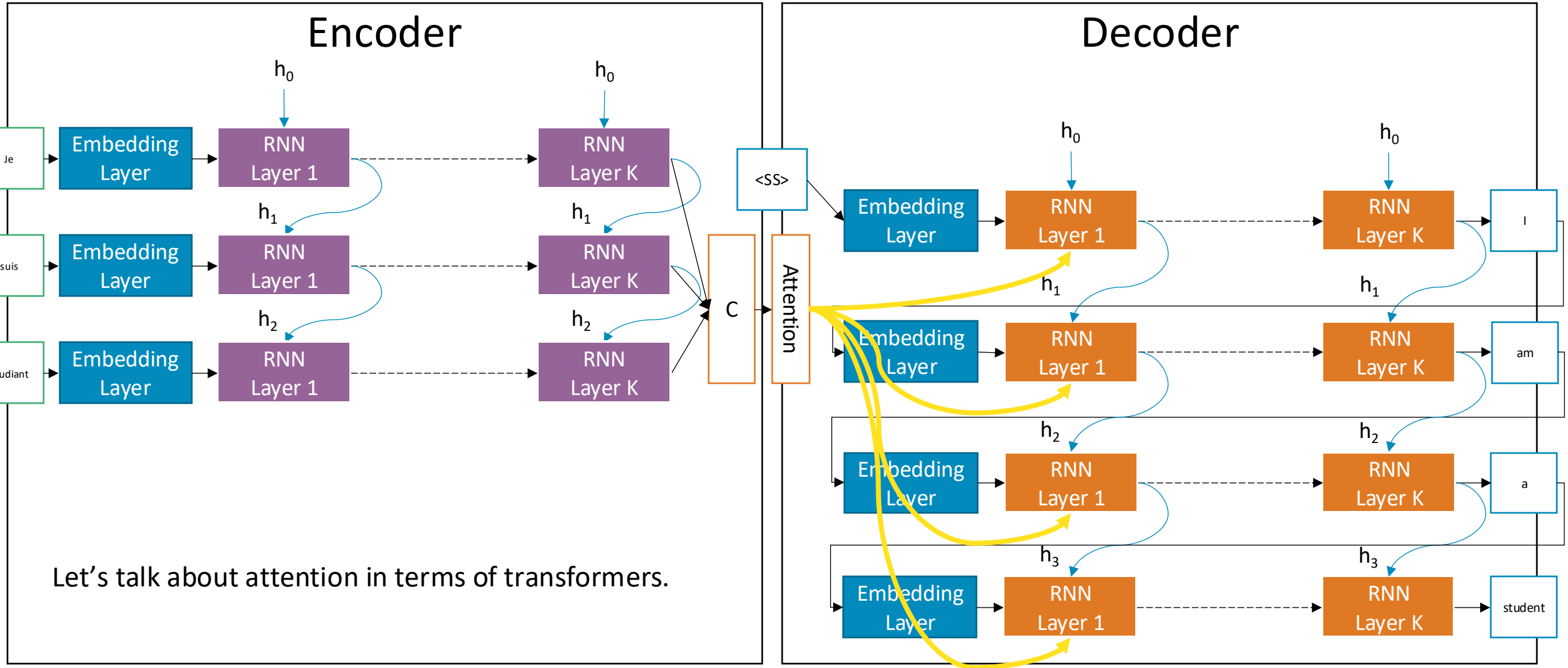

The animal didn't cross the street because it was too wide.

L'animal n'a pas traverse la rue parce qu'elle était trop large.



# Translation Encoder/Decoder RNN

Unrolled Version




Let's talk about attention in terms of transformers.

# Transformer Model Introduced In…

2017

**Attention Is All You Need**


**Ashish Vaswani*** Google Brain avaswani@google.com

**Noam Shazeer*** Google Brain noam@google.com

**Niki Parmar*** Google Research nikip@google.com

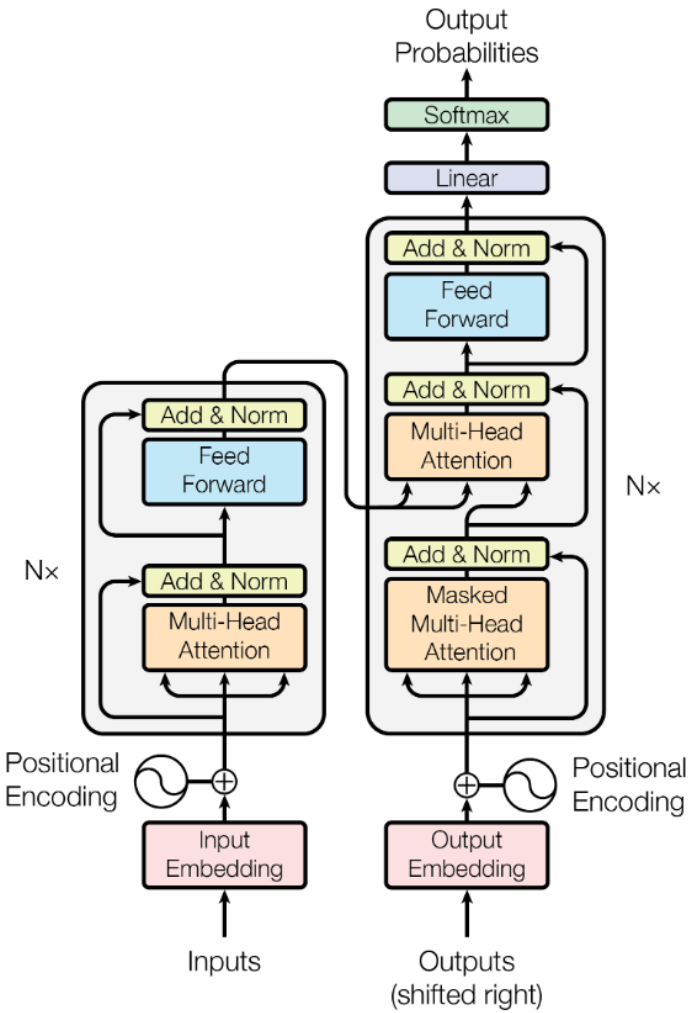
**Jakob Uszkoreit*** Google Research usz@google.com

**Llion Jones*** Google Research llion@google.com

**Aidan N. Gomez*** † University of Toronto aidan@cs.toronto.edu

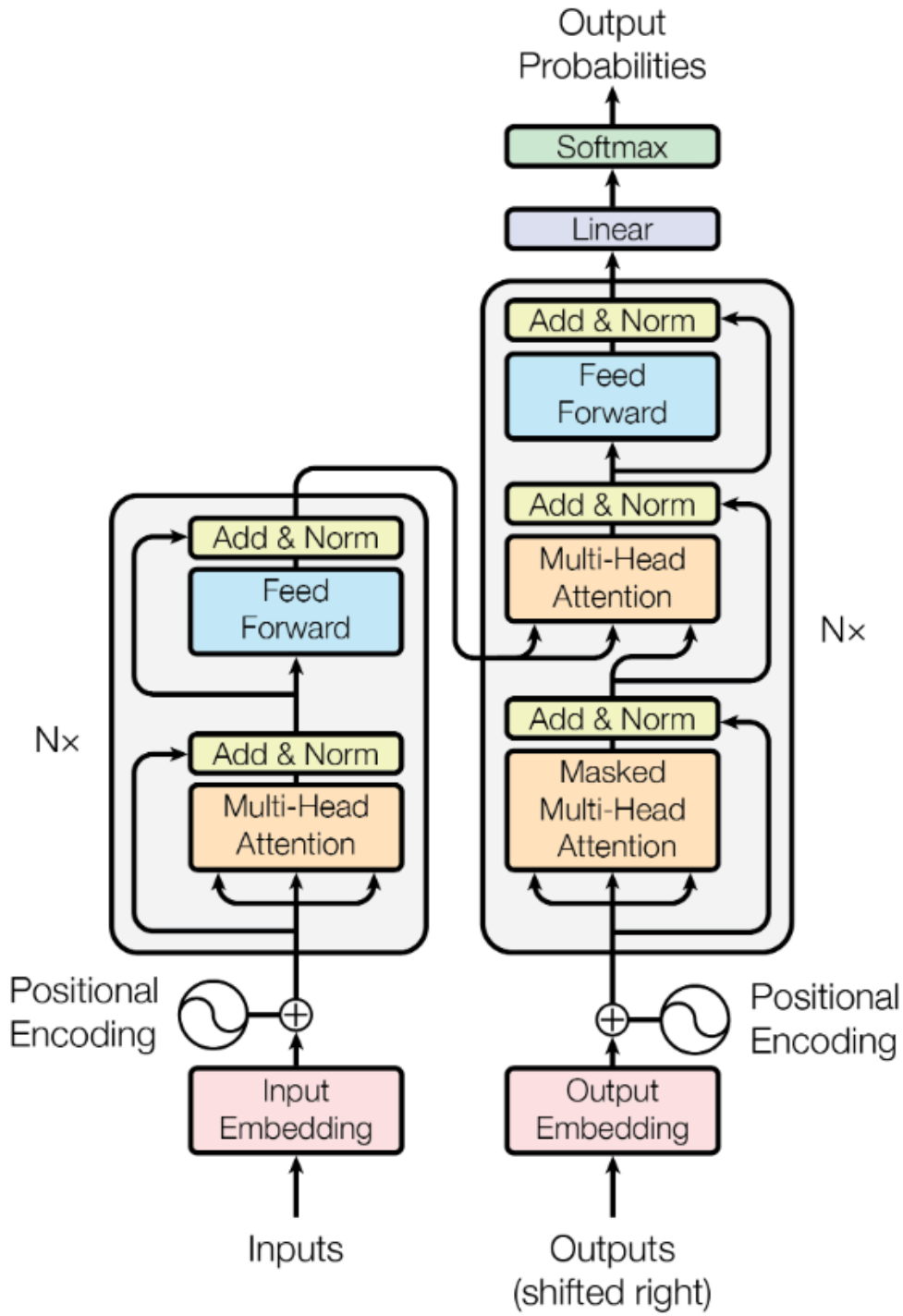
**Łukasz Kaiser*** Google Brain lukaszkaiser@google.com

**Illia Polosukhin*** ‡ illia.polosukhin@gmail.com


No recurrence. No convolutions.

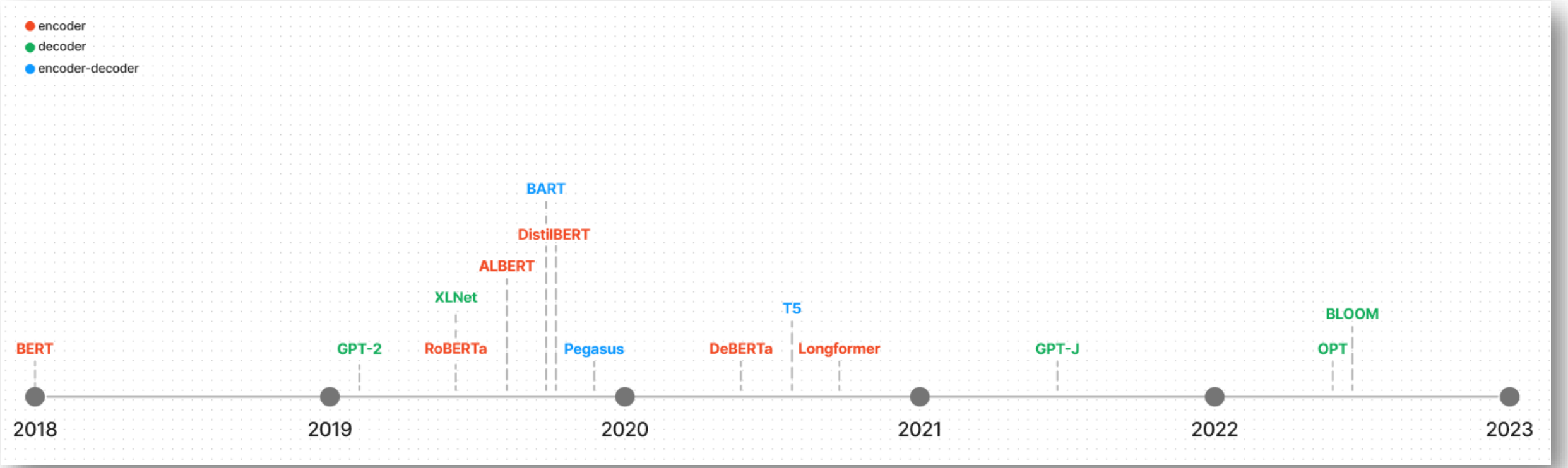
Better output. Faster to train.


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

# A Lot Happened All at Once

**Attention is All you Need**

Part of Advances in Neural Information Processing Systems 30 (NIPS 2017)




https://huggingface.co/docs/transformers/model_summary

# Common Transformer Applications

- Audio
  - Classification
  - Speech to text
- Vision
  - Classification
  - Object detection
  - Segmentation
  - Depth estimation
  - Captioning
- Text (NLP)
  - Classification
  - Question answering
  - Summarization
  - Translation
  - Named entity recognition
  - Language modeling
    - Masked (fill in the blank)
    - Generation
    - Completion

# Outline

- High-level comparison
  - Feed forward networks
  - Recurrent networks
  - Transformer networks
- Remember the RNN encoder/decoder architecture
- Motivate the need for attention
- Explore the transformer architecture
  - Start with an example: Dialogue completion
  - Start with word indices as input
  - Dig through the transformer architecture
    - Embeddings (take indices as input)
    - Encoder
    - Decoder

# PyTorch Transformer Layers

| | |
|---|---|
| nn.Transformer | A transformer model. |
| nn.TransformerEncoder | TransformerEncoder is a stack of N encoder layers. |
| nn.TransformerDecoder | TransformerDecoder is a stack of N decoder layers. |
| nn.TransformerEncoderLayer | TransformerEncoderLayer is made up of self-attn and feedforward network. |
| nn.TransformerDecoderLayer | TransformerDecoderLayer is made up of self-attn, multi-head-attn and feedforward network. |

# torch.nn.Transformer

Transformer: A Novel Neural Network Architecture for Language Understanding

```
class Transformer(Module):
    def __init__(
        self,
        d_model: int = 512,
        nhead: int = 8,
        num_encoder_layers: int = 6,
        num_decoder_layers: int = 6,
        dim_feedforward: int = 2048,
        dropout: float = 0.1,
        activation: Union[str, Callable[[Tensor], Tensor]] = F.relu,
        custom_encoder: Optional[Any] = None,
        custom_decoder: Optional[Any] = None,
        layer_norm_eps: float = 1e-5,
        batch_first: bool = False,
        norm_first: bool = False,
        bias: bool = True,
        device=None,
        dtype=None,
    ) -> None:
```

the number of expected features in the encoder/decoder inputs

the number of heads in the multiheadattention models

the dimension of the feedforward network model

Default input and output tensor shapes (seq, batch, feature)

Batch Size

seq length

```
transformer_model = nn.Transformer(nhead=16, num_encoder_layers=12)
src = torch.rand((10, 32, 512))
tgt = torch.rand((20, 32, 512))
out = transformer_model(src, tgt)
```

# Example Application: Dialogue Completer

TRAIN: *But that is not for them to decide. All we have to decide is what to do with the time that is given us.*

INPUT: *But that is not for them to decide. All we have*

OUTPUT: *to decide is what to do with the time that is given us.*

*"But that is not for them to decide. All we have to decide is what to do with the time that is given us."*

# Transformer Procedure

1. Tokenization (convert input into tokens with arbitrary unique identifiers)
2. Transform tokens into embeddings (each token has a learned “meaning”)
3. Augment embeddings into positional embeddings
4. For each positional embedding

# Words to Vocabulary Indices

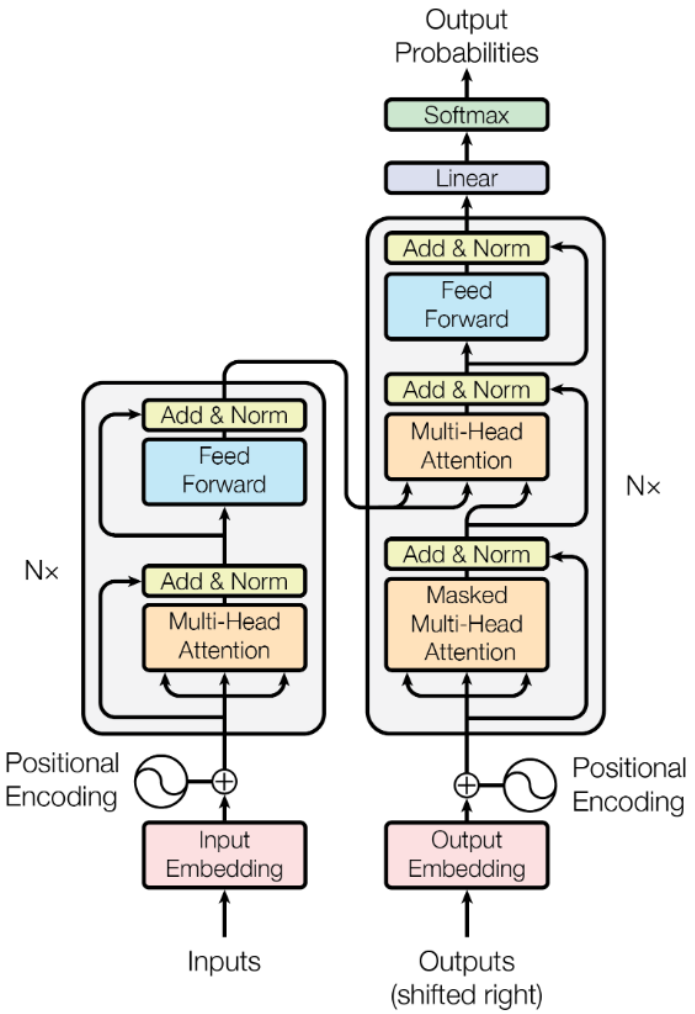
INPUT: <START>*But that is not for them to decide. All we have*<END>

OUTPUT: <START>*to decide is what to do with the time that is given us.*<END>

Input and output indices don’t always have to be the same.

# Constructing a Transformer

```
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )


    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)


    return model
```

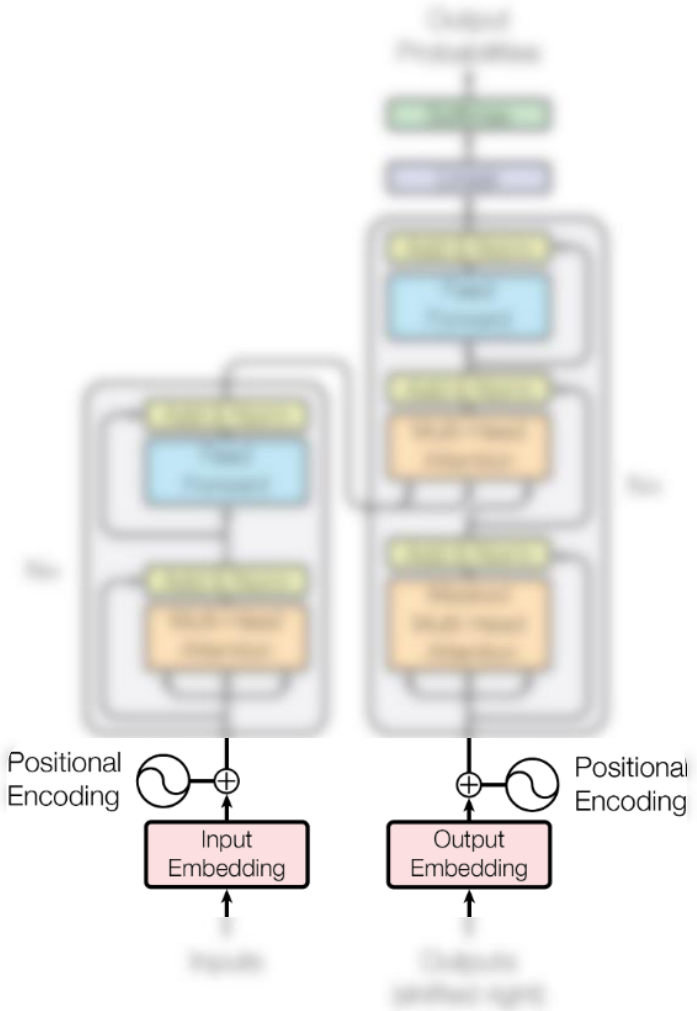



https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py

Adapted from: The Annotated Transformer

# Constructing a Transformer

```
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )


    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)


    return model
```

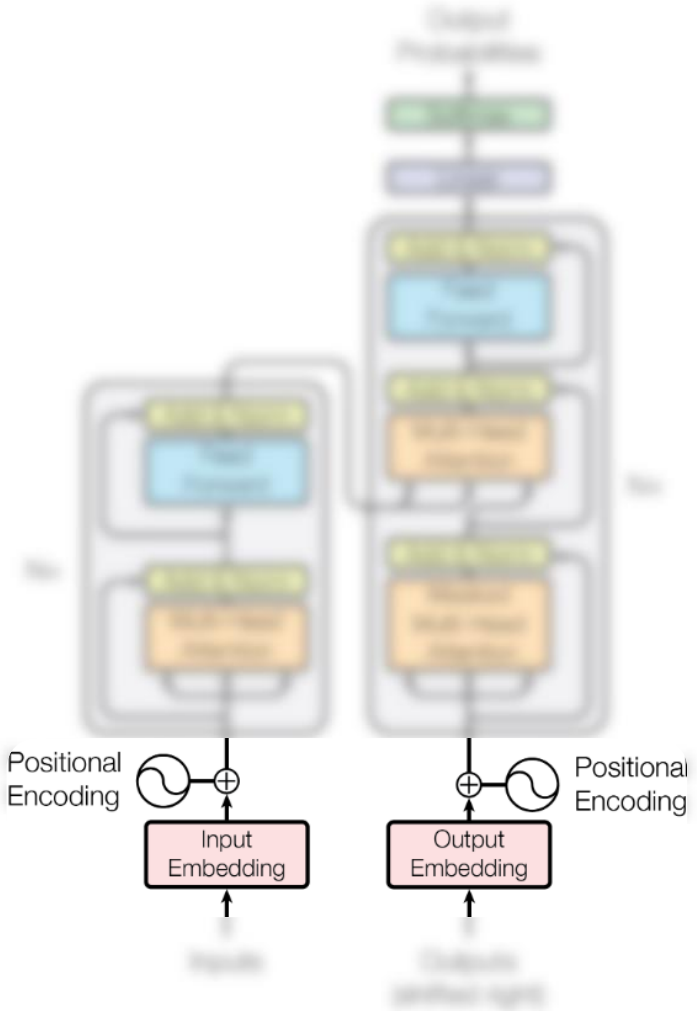



https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py

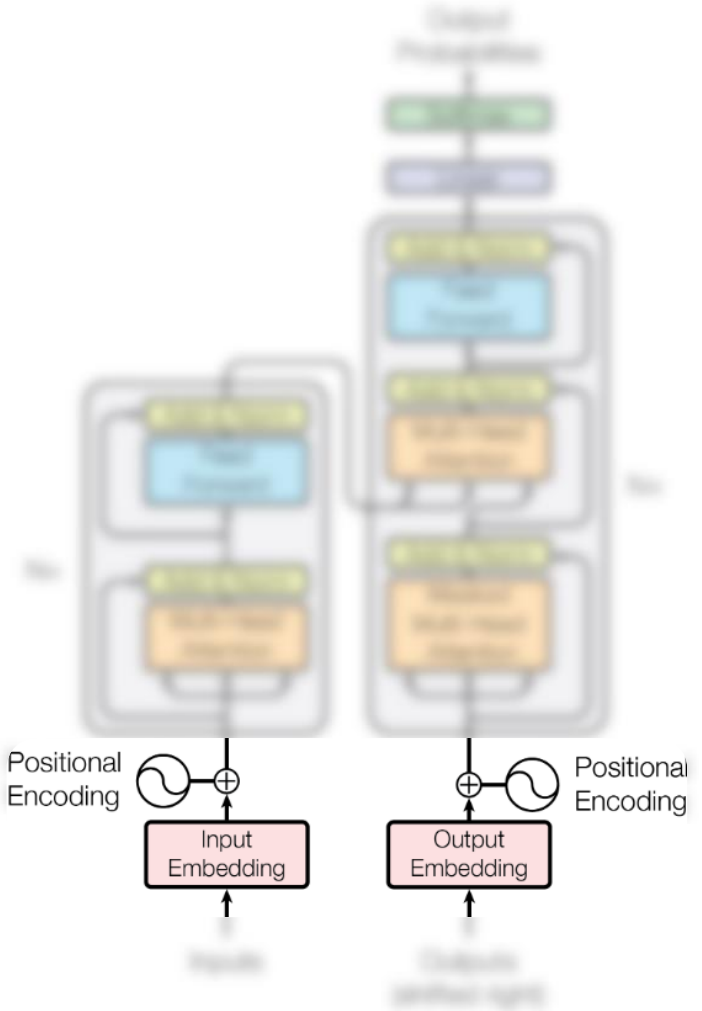
Adapted from: The Annotated Transformer

# PositionalEmbedding

```python
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout_prob: float, max_len: int):
        super().__init__()
        self.dropout = nn.Dropout(dropout_prob)

        # Compute the positional encodings once in log space
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * -(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: torch.Tensor) -> torch.Tensor:
        x = x + self.pe[:, : x.size(1)].requires_grad_(False)
        return self.dropout(x)


class PositionalEmbedding(nn.Module):
    "Scaled embedding followed by a positional encoding."

    def __init__(self, vocab_size: int, d_model: int, dropout_prob: float, max_len: int):
        super().__init__()
        self.embed = ScaledEmbedding(d_model, vocab_size)
        self.positional_encoding = PositionalEncoding(d_model, dropout_prob, max_len)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.positional_encoding(self.embed(x))
```



# PositionalEmbedding

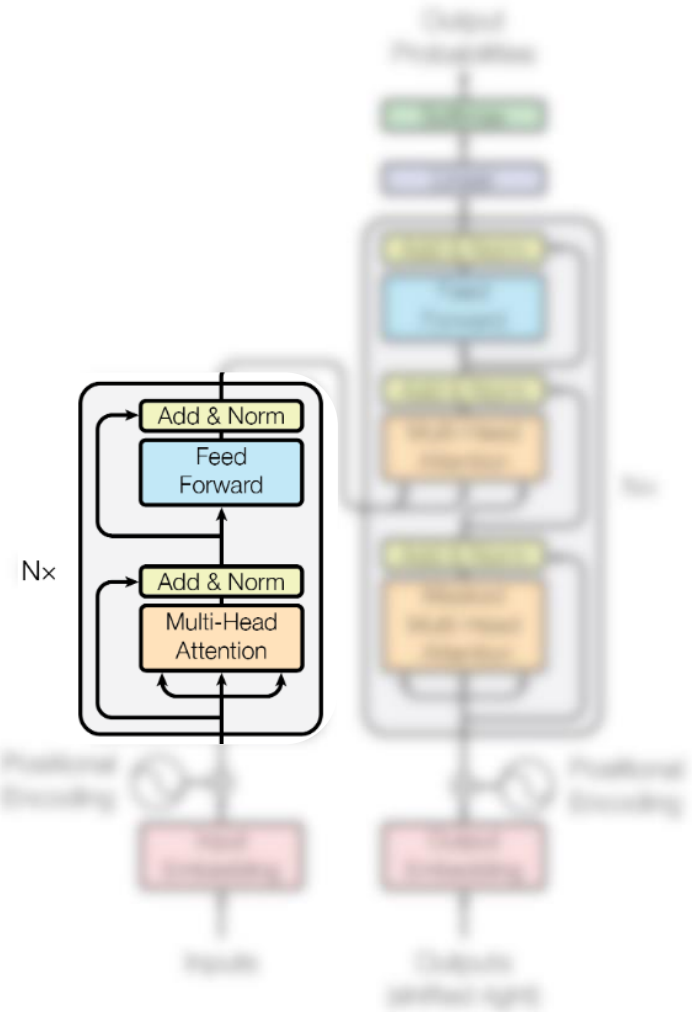



$$p_i = \sin\left(\frac{i}{10000^{\frac{2i}{d}}}\right) \forall \text{ even numbers}$$

$$p_i = \cos\left(\frac{i}{10000^{\frac{2i}{d}}}\right) \forall \text{ odd numbers}$$

# Constructing a Transformer

```
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )


    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)


    return model
```

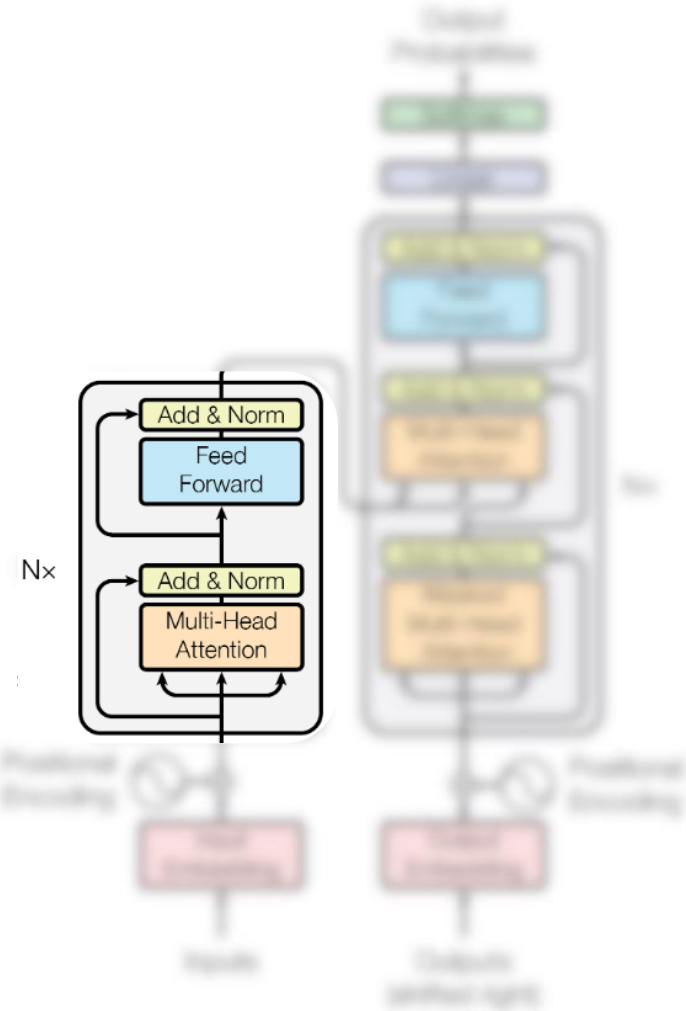



https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py

Adapted from: The Annotated Transformer

# Encoder

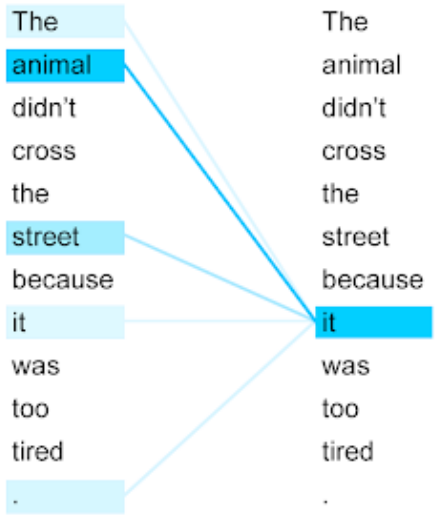
```python
class EncoderLayer(nn.Module):
    "An encoder layer is composed of self-attention and feed forward components."

    def __init__(self, d_model: int, num_head: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.self_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.feed_forward = PositionWiseFeedForward(d_model, ff_dim, dropout_prob)
        self.sublayer1 = SublayerConnection(d_model, dropout_prob)
        self.sublayer2 = SublayerConnection(d_model, dropout_prob)

    def forward(self, x: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:
        x = self.sublayer1(x, lambda x: self.self_attention(x, x, x, mask))
        return self.sublayer2(x, self.feed_forward)


class Encoder(nn.Module):
    "An encoder block comprising N encoder layers."

    def __init__(self, d_model: int, num_head: int, num_layers: int, ff_dim: int, dropout_pr
        super().__init__()

        self.layers = nn.ModuleList()
        for _ in range(num_layers):
            self.layers.append(EncoderLayer(d_model, num_head, ff_dim, dropout_prob))

        self.norm = nn.LayerNorm(d_model)

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



# Coreference Resolution

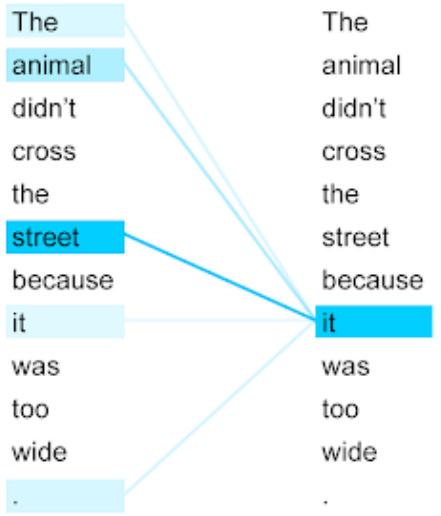
The animal didn’t cross the street because it was too tired.

L’animal n’a pas traverse la rue parce qu’il était trop fatigue.



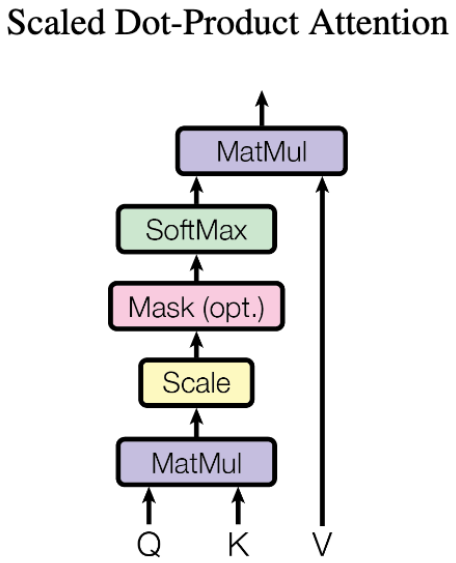

The animal didn’t cross the street because it was too wide.

L’animal n’a pas traverse la rue parce qu’elle était trop large.

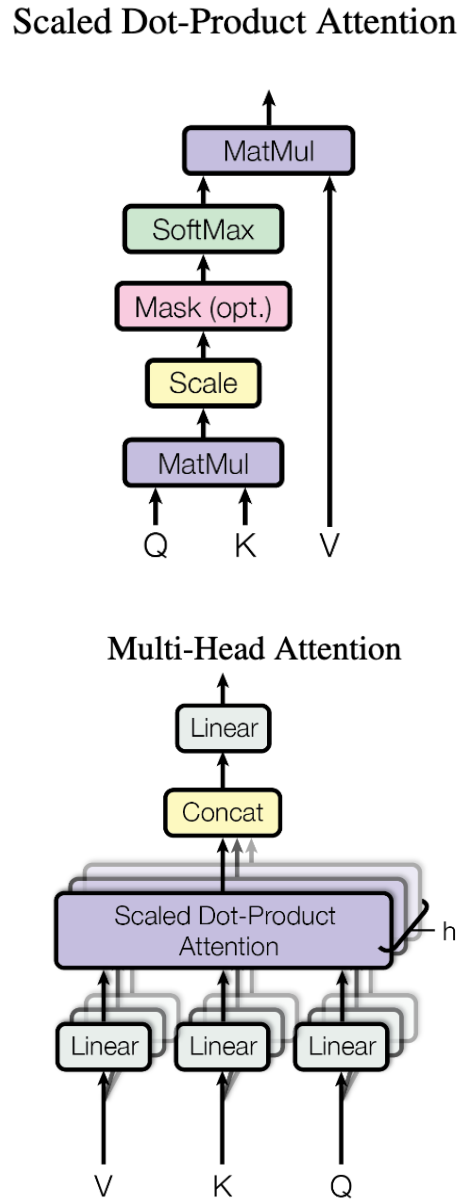


# Self-Attention

- Self-attention processes the entire sequence at once (not one at a time like an RNN)



# MultiHeadedAttention



```python
class MultiHeadedAttention(nn.Module):
    "Multi-Head Attention module."

    def __init__(self, d_model: int, num_head: int, dropout_prob: float):
        super().__init__()
        assert d_model % num_head == 0

        self.d_per_head = d_model // num_head
        self.num_head = num_head

        self.linear_layers = nn.ModuleList()
        for _ in range(4):
            self.linear_layers.append(nn.Linear(d_model, d_model))

        self.dropout = nn.Dropout(dropout_prob)

    def forward(
        self,
        query: torch.Tensor,
        key: torch.Tensor,
        value: torch.Tensor,
        mask: torch.Tensor | None = None,
    ) -> torch.Tensor:

        if mask is not None:
            mask = mask.unsqueeze(1)
```

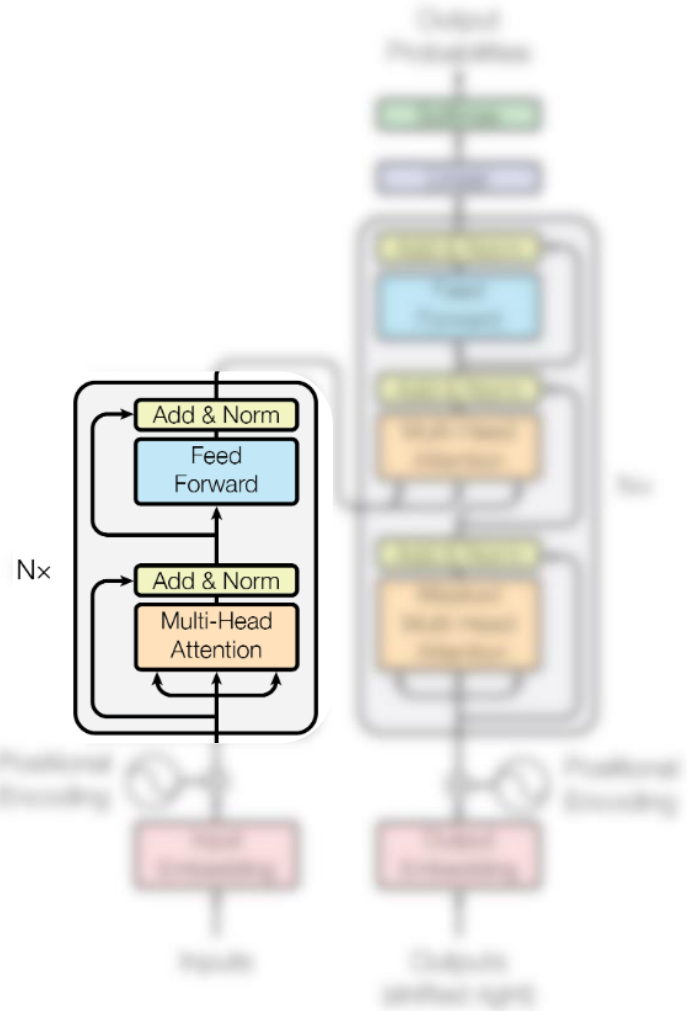
```
def forward(
    self,
    query: torch.Tensor,
    key: torch.Tensor,
    value: torch.Tensor,
    mask: torch.Tensor | None = None,
) -> torch.Tensor:

    if mask is not None:
        mask = mask.unsqueeze(1)
    num_batches = query.size(0)

    # (1) Do all the linear projections in batch from d_model => h x d_k
    query, key, value = [
        lin(x).view(num_batches, -1, self.num_head, self.d_per_head).transpose(1, 2)
        for lin, x in zip(self.linear_layers, (query, key, value))
    ]

    # (2) Apply attention on all the projected vectors in batch.
    # "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    attention = scores.softmax(dim=-1)
    if self.dropout is not None:
```

```
# (2) Apply attention on all the projected vectors in batch.
# "Compute 'Scaled Dot Product Attention'"
d_k = query.size(-1)
scores = torch.matmul(query, key.transpose(-2, -1)) / sqrt(d_k)
if mask is not None:
    scores = scores.masked_fill(mask == 0, -1e9)
attention = scores.softmax(dim=-1)
if self.dropout is not None:
    attention = self.dropout(attention)
x = torch.matmul(attention, value)

# (3) "Concat" using a view and apply a final linear.
x = (
    x.transpose(1, 2)
    .contiguous()
    .view(num_batches, -1, self.num_head * self.d_per_head)
)
del query
del key
del value
return self.linear_layers[-1](x)
```

# PositionWiseFeedForward

```python
class PositionWiseFeedForward(nn.Module):
    "Implements feedforward neural network equation."

    def __init__(self, d_model: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.w1 = nn.Linear(d_model, ff_dim)
        self.w2 = nn.Linear(ff_dim, d_model)
        self.dropout = nn.Dropout(dropout_prob)


    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.w2(self.dropout(self.w1(x).relu()))
```

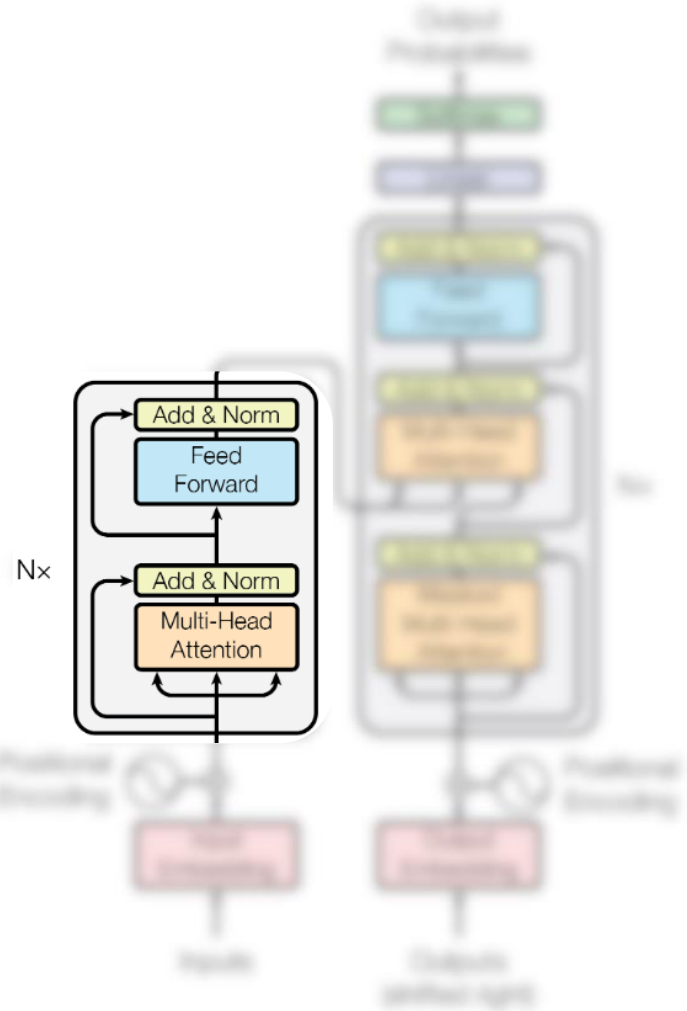
```python
class EncoderLayer(nn.Module):
    "An encoder layer is composed of self-attention and feed forward components."

    def __init__(self, d_model: int, num_head: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.self_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.feed_forward = PositionWiseFeedForward(d_model, ff_dim, dropout_prob)
        self.sublayer1 = SublayerConnection(d_model, dropout_prob)
        self.sublayer2 = SublayerConnection(d_model, dropout_prob)

    def forward(self, x: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:
        x = self.sublayer1(x, lambda x: self.self_attention(x, x, x, mask))
        return self.sublayer2(x, self.feed_forward)
```



# SublayerConnection

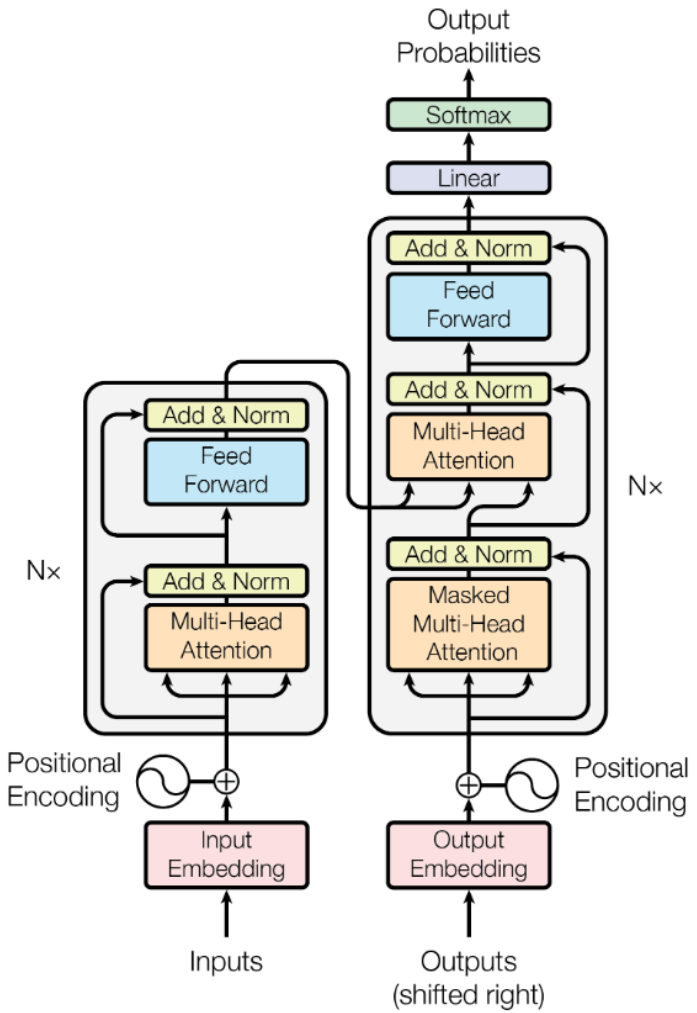
```python
class SublayerConnection(nn.Module):
    "A residual connection followed by a LayerNorm."


    def __init__(self, d_model: int, dropout_prob: float):
        super().__init__()
        self.layernorm = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout_prob)


    def forward(self, x: torch.Tensor, sublayer: nn.Module) -> torch.Tensor:
        return x + self.dropout(sublayer(self.layernorm(x)))
```

```python
class EncoderLayer(nn.Module):
    "An encoder layer is composed of self-attention and feed forward components."

    def __init__(self, d_model: int, num_head: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.self_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.feed_forward = PositionWiseFeedForward(d_model, ff_dim, dropout_prob)
        self.sublayer1 = SublayerConnection(d_model, dropout_prob)
        self.sublayer2 = SublayerConnection(d_model, dropout_prob)

    def forward(self, x: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:
        x = self.sublayer1(x, lambda x: self.self_attention(x, x, x, mask))
        return self.sublayer2(x, self.feed_forward)
```



# Constructing a Transformer

```
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )


    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)


    return model
```

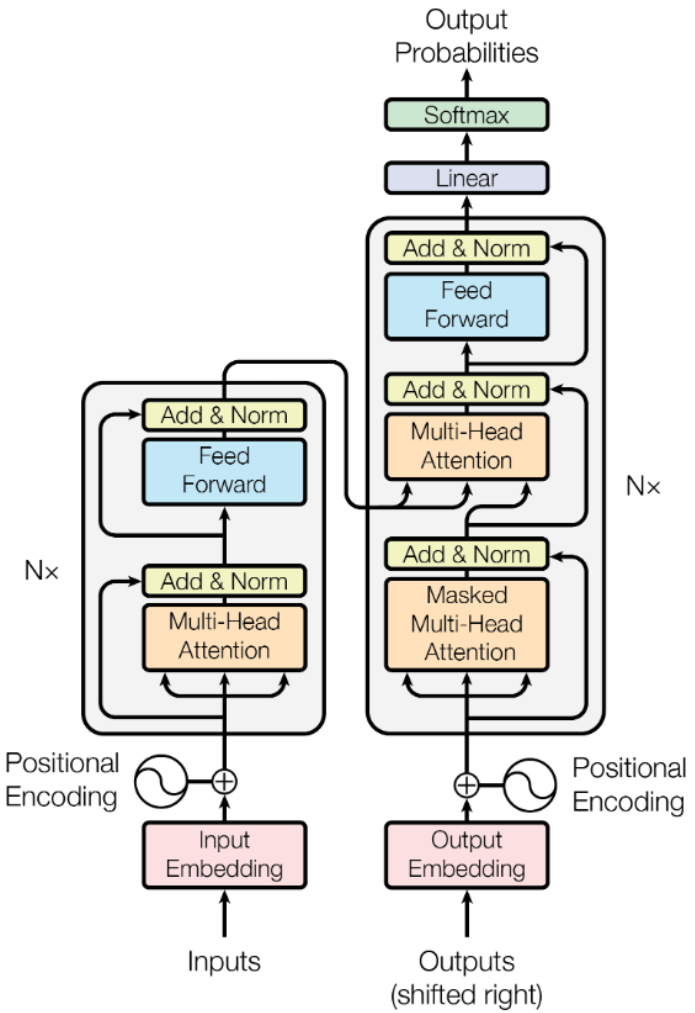



https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py

Adapted from: The Annotated Transformer

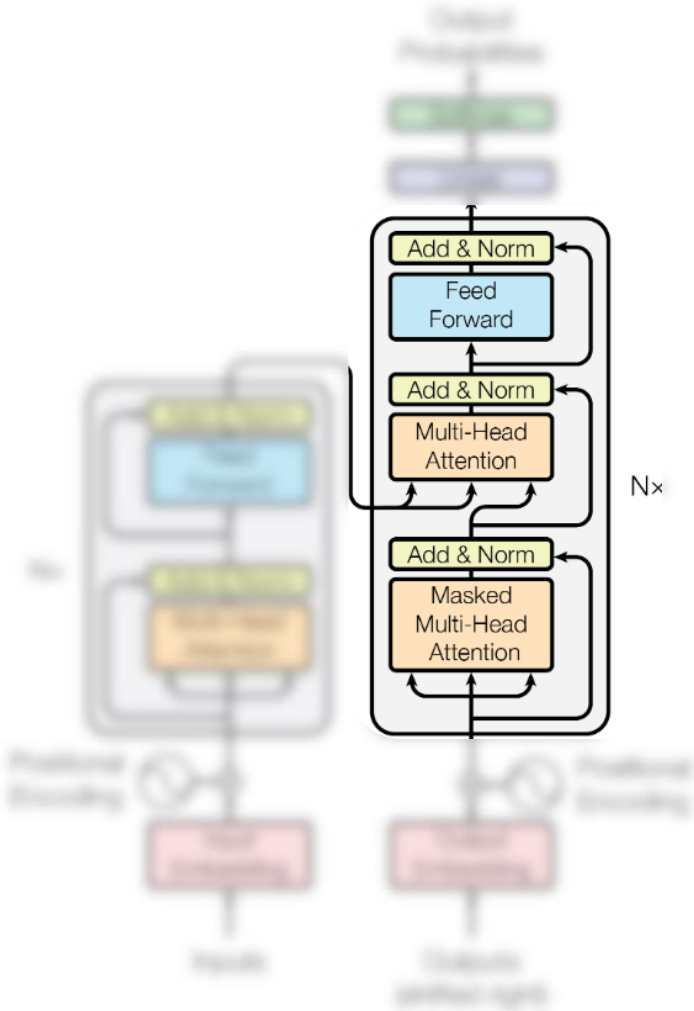
# Decoder

```python
class DecoderLayer(nn.Module):
    "A decode layer comprises self and source attention and feed forward components."
    def __init__(self, d_model: int, num_head: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.self_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.src_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.feed_forward = PositionWiseFeedForward(d_model, ff_dim, dropout_prob)
        self.sublayer1 = SublayerConnection(d_model, dropout_prob)
        self.sublayer2 = SublayerConnection(d_model, dropout_prob)
        self.sublayer3 = SublayerConnection(d_model, dropout_prob)

    def forward(self, x: torch.Tensor, memory: torch.Tensor, src_mask: torch.Tensor, tgt_ma
        m = memory
        x = self.sublayer1(x, lambda x: self.self_attention(x, x, x, tgt_mask))
        x = self.sublayer2(x, lambda x: self.src_attention(x, m, m, src_mask))
        return self.sublayer3(x, self.feed_forward)

class Decoder(nn.Module):
    "A decoder block comprising N layers with masking."
    def __init__(self, d_model: int, num_head: int, num_layers: int, ff_dim: int, dropout_
        super().__init__()
        self.layers = nn.ModuleList()
        for _ in range(num_layers):
            self.layers.append(DecoderLayer(d_model, num_head, ff_dim, dropout_prob))

        self.norm = nn.LayerNorm(d_model)

    def forward(self, x: torch.Tensor, memory: torch.Tensor, src_mask: torch.Tensor, tgt_ma
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```



# Decoder

```python
class DecoderLayer(nn.Module):
    "A decode layer comprises self and source attention and feed forward components."
    def __init__(self, d_model: int, num_head: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.self_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.src_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.feed_forward = PositionWiseFeedForward(d_model, ff_dim, dropout_prob)
        self.sublayer1 = SublayerConnection(d_model, dropout_prob)
        self.sublayer2 = SublayerConnection(d_model, dropout_prob)
        self.sublayer3 = SublayerConnection(d_model, dropout_prob)

    def forward(self, x: torch.Tensor, memory: torch.Tensor, src_mask: torch.Tensor, tgt_ma
        m = memory
        x = self.sublayer1(x, lambda x: self.self_attention(x, x, x, tgt_mask))
        x = self.sublayer2(x, lambda x: self.src_attention(x, m, m, src_mask))
        return self.sublayer3(x, self.feed_forward)

class Decoder(nn.Module):
    "A decoder block comprising N layers with masking."
    def __init__(self, d_model: int, num_head: int, num_layers: int, ff_dim: int, dropout_
        super().__init__()
        self.layers = nn.ModuleList()
        for _ in range(num_layers):
            self.layers.append(DecoderLayer(d_model, num_head, ff_dim, dropout_prob))

        self.norm = nn.LayerNorm(d_model)

    def forward(self, x: torch.Tensor, memory: torch.Tensor, src_mask: torch.Tensor, tgt_ma
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```



# Example Application: Dialogue Completer

TRAIN: *But that is not for them to decide. All we have to decide is what to do with the time that is given us.*

INPUT: *But that is not for them to decide. All we have*

OUTPUT: *to decide is what to do with the time that is given us.*

*"But that is not for them to decide. All we have to decide is what to do with the time that is given us."*

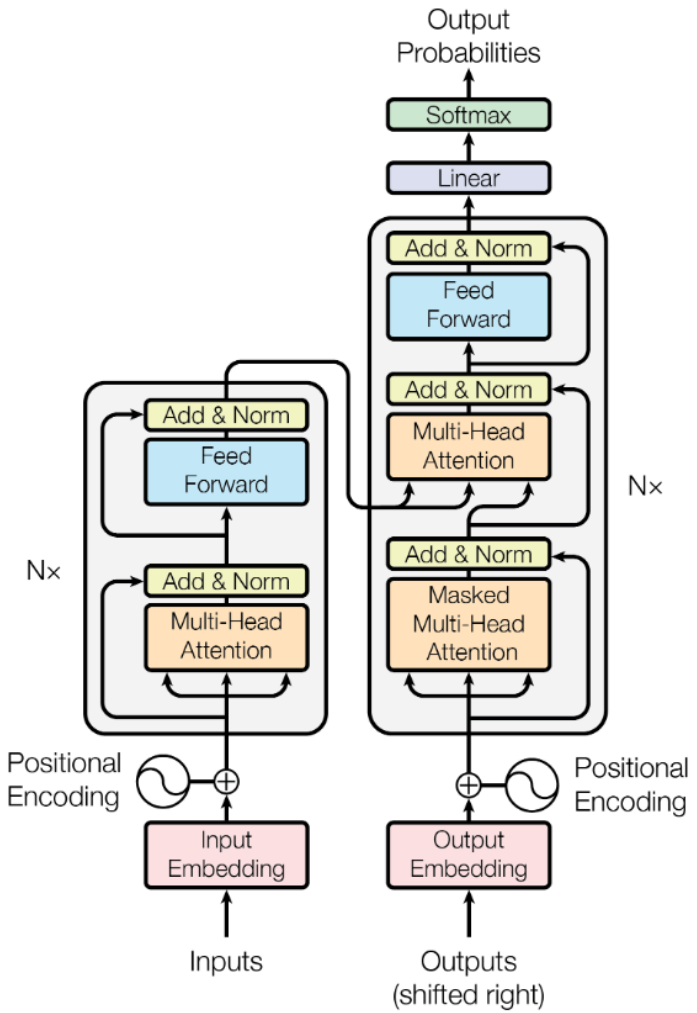
# Masking

INPUT: <START>*But that is not for them to decide. All we have*<END>

OUTPUT: <START>*to decide is what to do with the time that is given us.*<END>

# Constructing a Transformer

```
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )


    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)


    return model
```

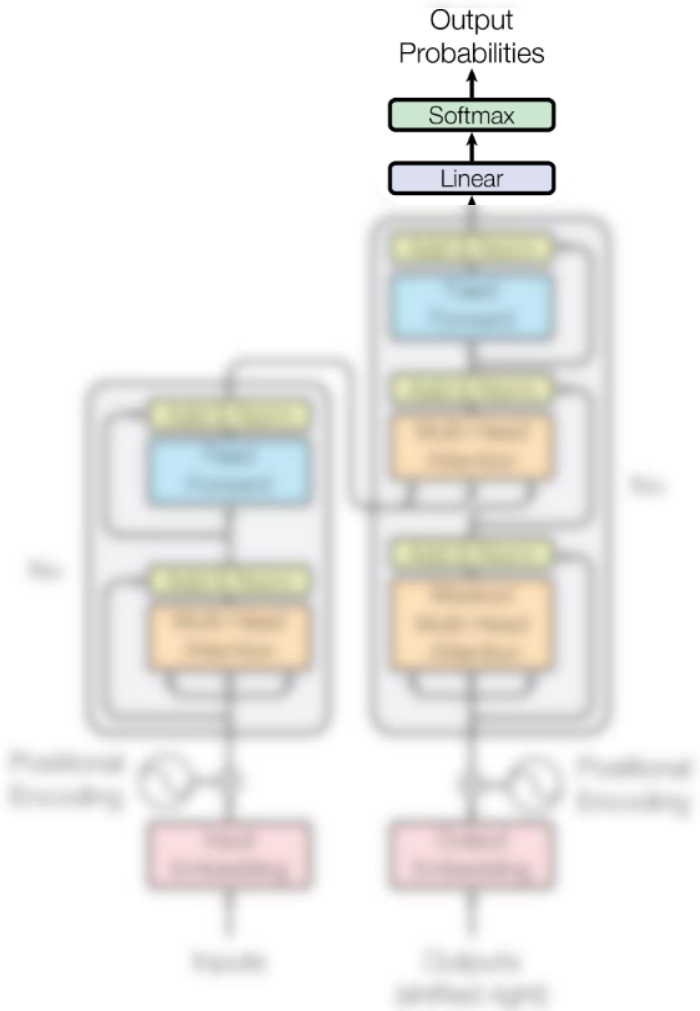



https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py

Adapted from: The Annotated Transformer

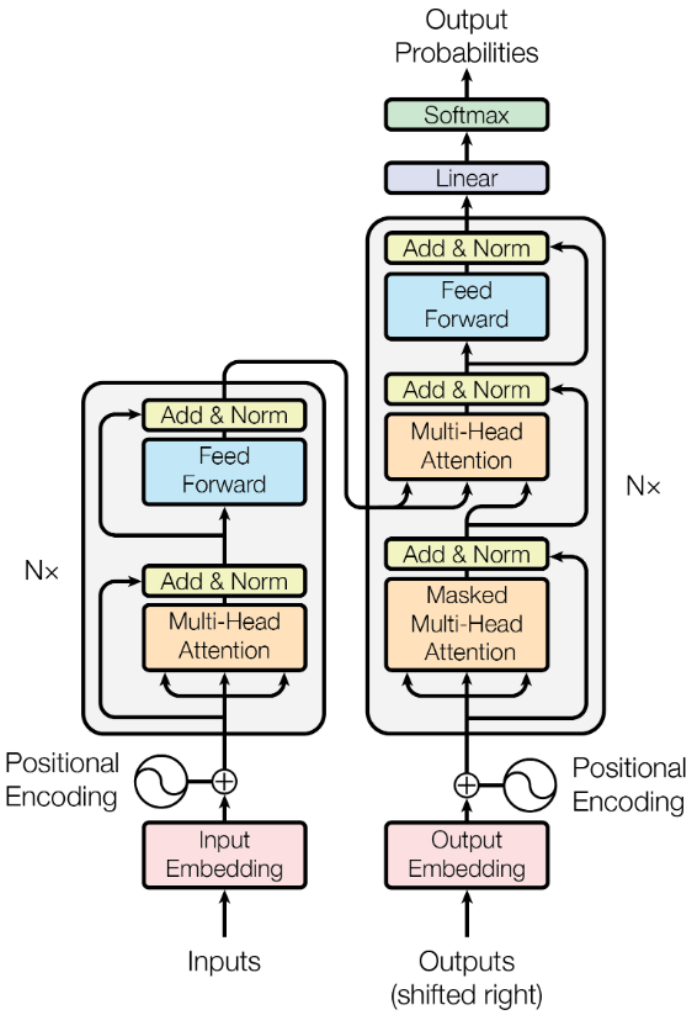
# Generator

```python
class Generator(nn.Module):
    "Define standard linear + softmax generation step."

    def __init__(self, d_model: int, vocab_size: int):
        super().__init__()
        self.linear = nn.Linear(d_model, vocab_size)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return log_softmax(self.linear(x), dim=-1)
```



# Constructing a Transformer

```
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )


    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)


    return model
```




https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py

Adapted from: The Annotated Transformer

# Summary

- Recurrent neural networks were the best performing for sequence tasks.
- They have an advantage of arbitrary input and output sequence lengths.
- Transformers have fixed input and output sequence lengths (with padding).
- But, transformers outperform RNNs on (probably) all sequence tasks.
- Transformers do not maintain any internal memory.
- Most (if not all) large language models (LLMs) are built on transformers.

# Resources

- Visualizing A Neural Machine Translation Model
- Transformer: A Novel Neural Network Architecture for Language Understanding
- A Visual Guide to Transformers Neural Networks: step by step explanation - YouTube
- L19.4.1 Using Attention Without the RNN -- A Basic Form of Self-Attention - YouTube
- karpathy/nanoGPT: The simplest, fastest repository for training/finetuning medium-sized GPTs.
- karpathy/minGPT: A minimal PyTorch re-implementation of the OpenAI GPT (Generative Pretrained Transformer) training
- karpathy/minGPT: A minimal PyTorch re-implementation of the OpenAI GPT (Generative Pretrained Transformer) training
- The Annotated Transformer