CSE 493/599
May 11 Lecture

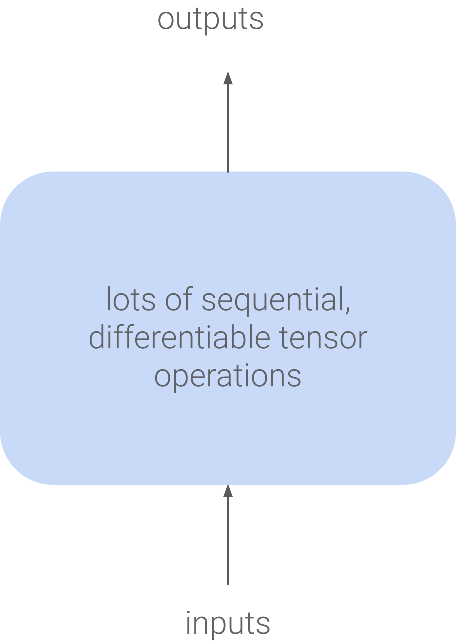
From zero to LLaMA

Overview

- RNNs
- Attention
- Transformers
- LLaMA

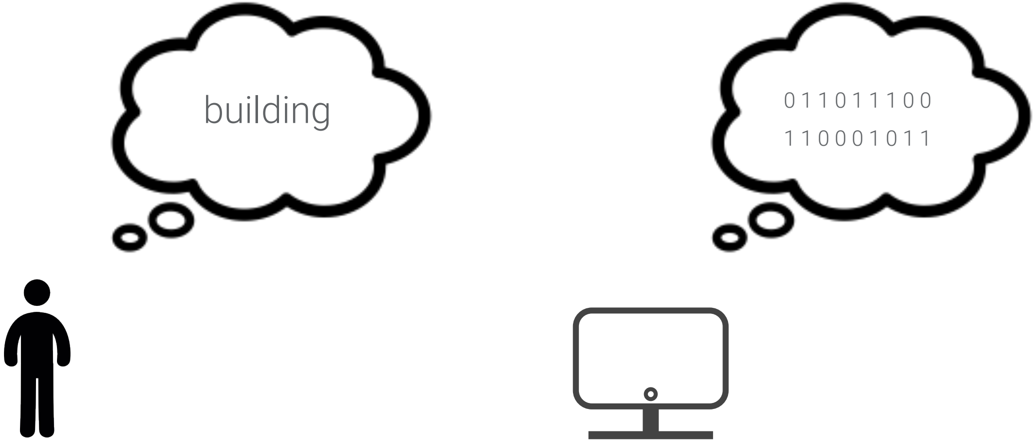
Overview

https://xkcd.com/1838/



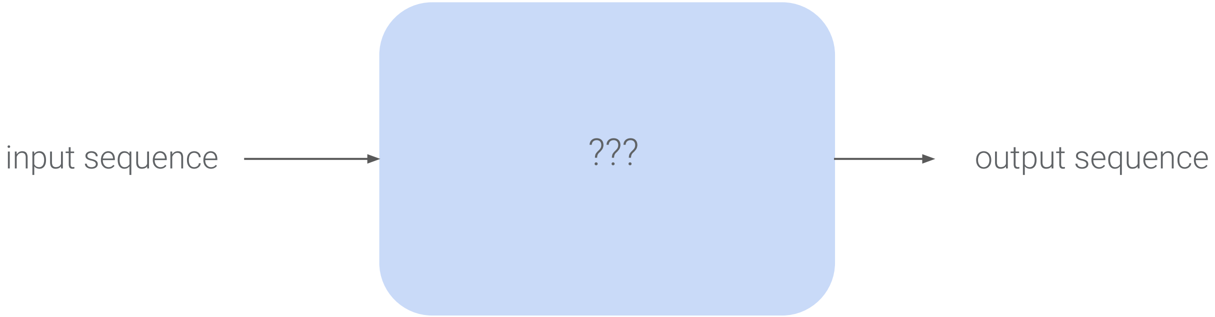
Two questions

How can we make numeric representations out of words?



Two questions

What sorts of models are better suited for processing sequential data?



Word Embeddings

Word embeddings

One-hot encodings

How to represent words?

vocab size

a →	**1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . . .
ability →	0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . . .
able →	0 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . . .
about →	0 0 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . . .
above →	0 0 0 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . . .
acarus →	0 0 0 0 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 0 . . .
.	.
.	.
.	.

Word embeddings

Embeddings: learned latent representations of words

How to represent words?

embedding size (<< vocab size)

a	→	.3452 .7162 .1827 .9382 .9182 . . .
ability	→	.1234 .8172 .6473 .5630 .0263 . . .
able	→	.1263 .8054 .5632 .5589 .0374 . . .
about	→	.7364 .2039 .2831 .2837 .1923 . . .
above	→	.9283 .0023 .0065 .2938 .5472 . . .
acarus	→	.1938 .2938 .0293 .5647 .2348 . . .
⋮		⋮

Token embeddings

Embeddings: learned latent representations of tokens

Decreased vocab size: words not in reduced dictionary will be split

How to represent tokens?

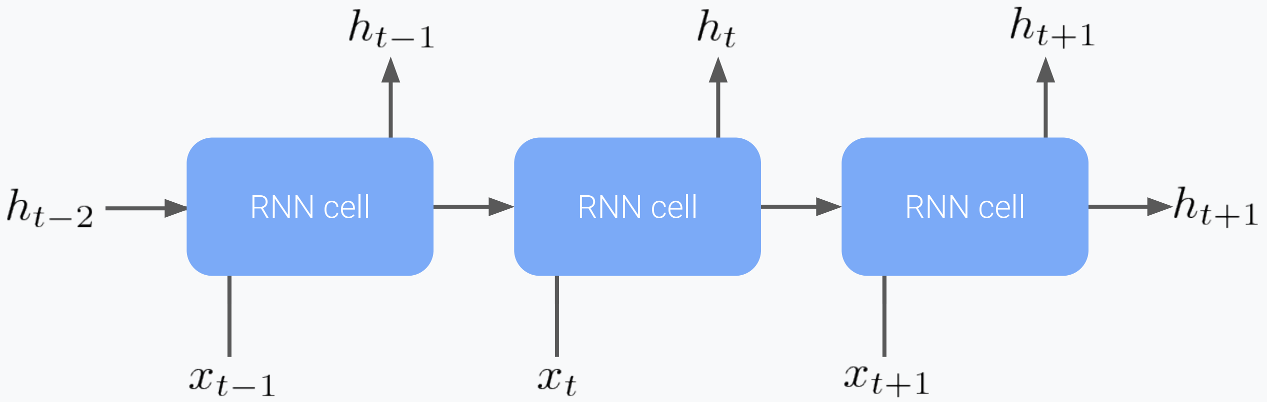
embedding size (<< vocab size)

a	.3452 .7162 .1827 .9382 .9182 . . .
ability	.1234 .8172 .6473 .5630 .0263 . . .
able	.1263 .8054 .5632 .5589 .0374 . . .
about	.7364 .2039 .2831 .2837 .1923 . . .
above	.9283 .0023 .0065 .2938 .5472 . . .
ac	.2754 .9572 .5810 .8513 .7412 . . .
ar	.7012 .7851 .4169 .0876 .9651 . . .
us	.1752 .9270 .0923 .7422 .1014 . . .
•	•
•	•
•	•

Recurrent Neural Networks

RNNs

Computations over
sequences of arbitrary length



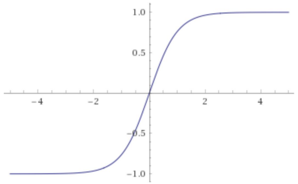
RNNs

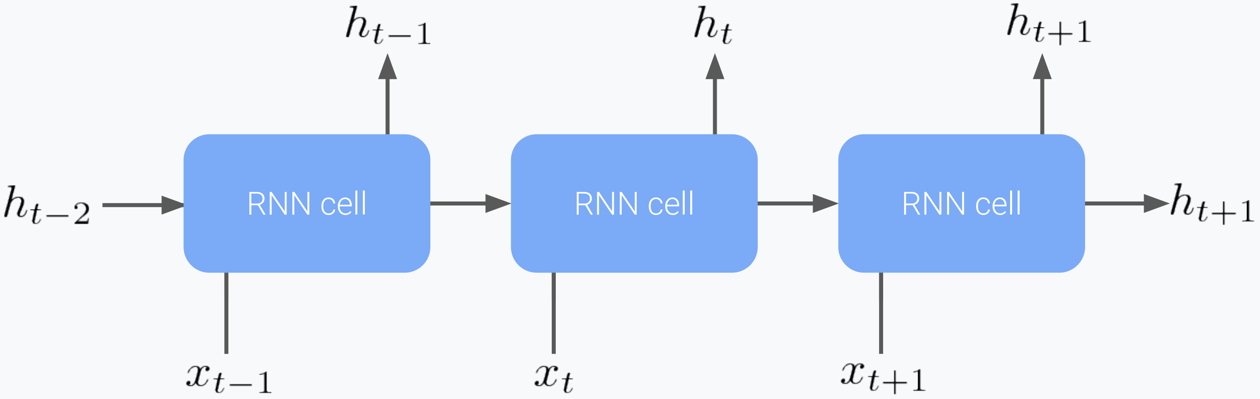
Computations over

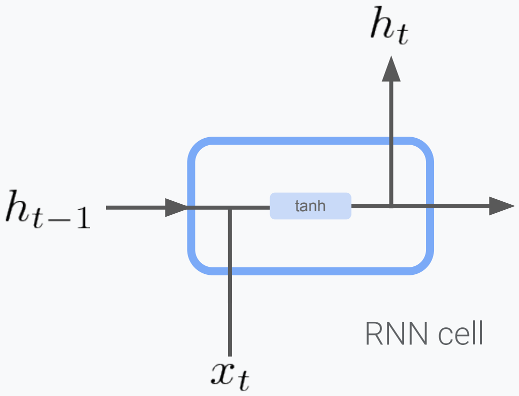
sequences of arbitrary length

= dense layer

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$



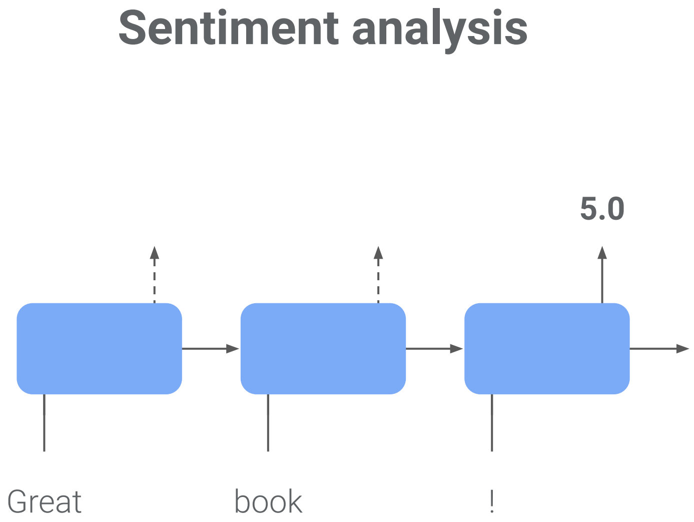




$$h_t = \tanh(W[h_{t-1}, x_t] + b)$$

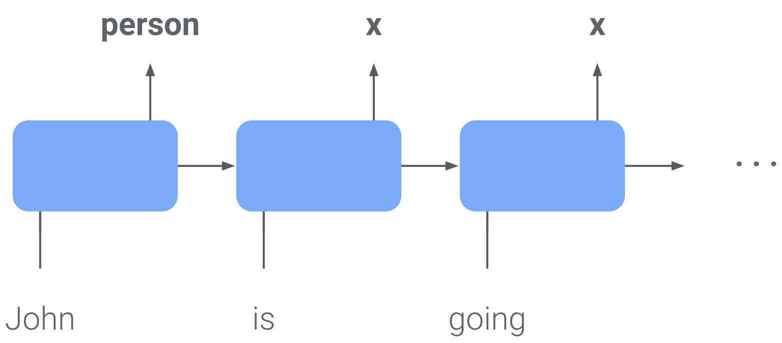
NOTE: W and b are shared across time.

RNNs are versatile!



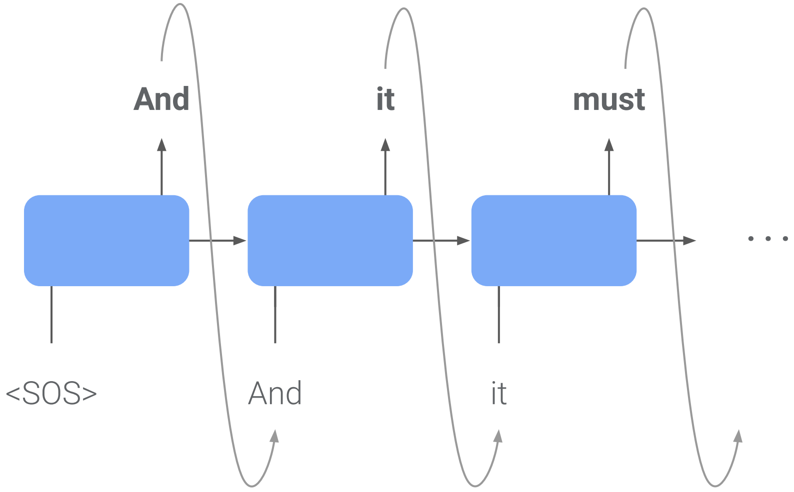
RNNs are versatile!

Named-entity recognition



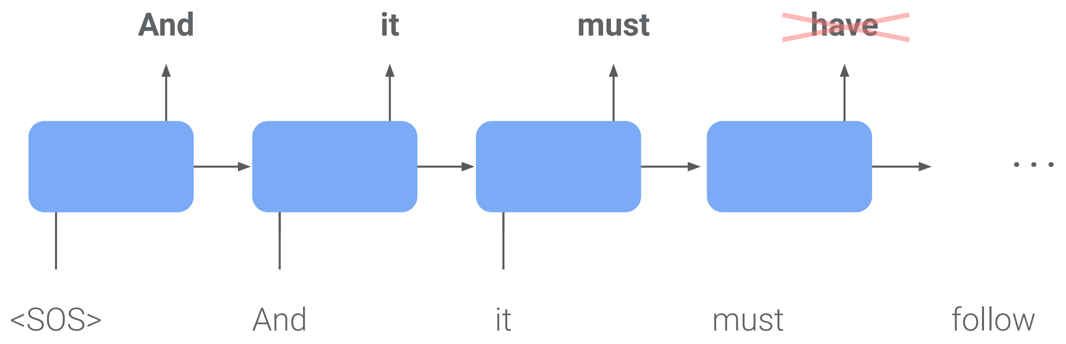
RNNs are versatile!

Language Models

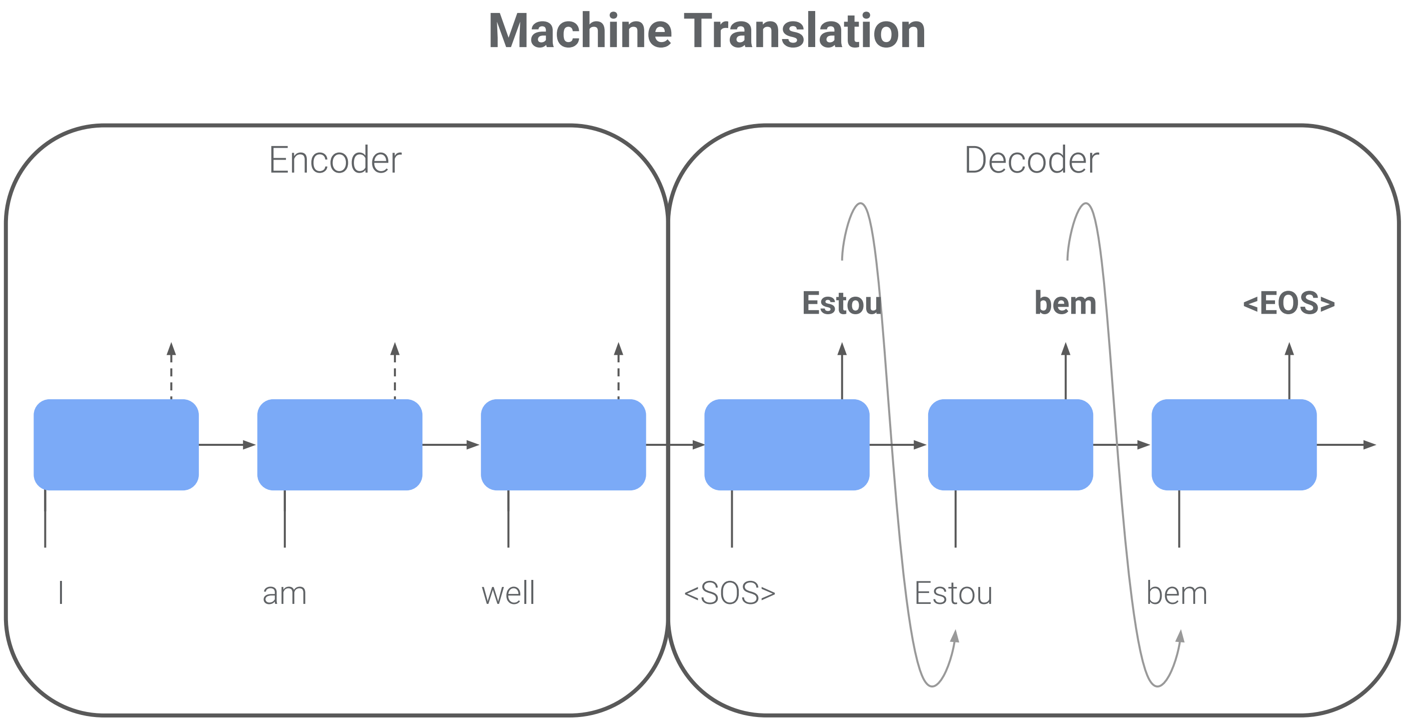


RNNs are versatile!

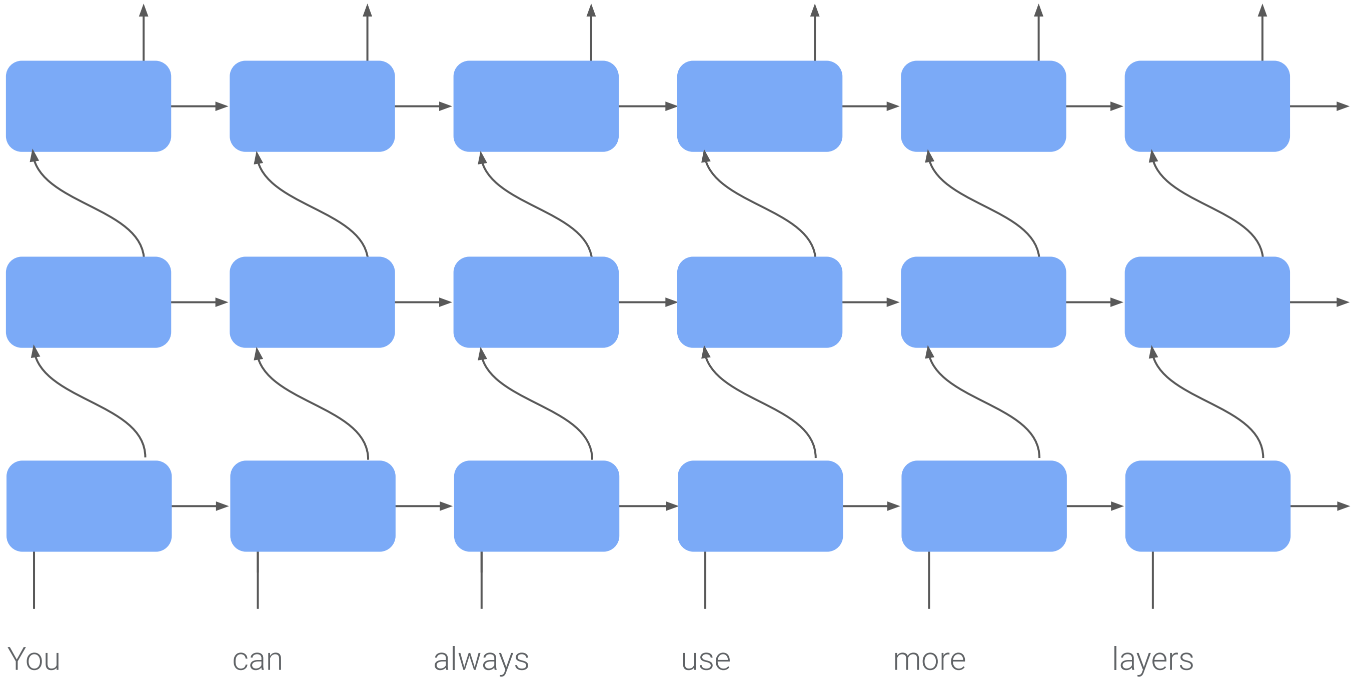
Language Models with Teacher forcing



RNNs are versatile!

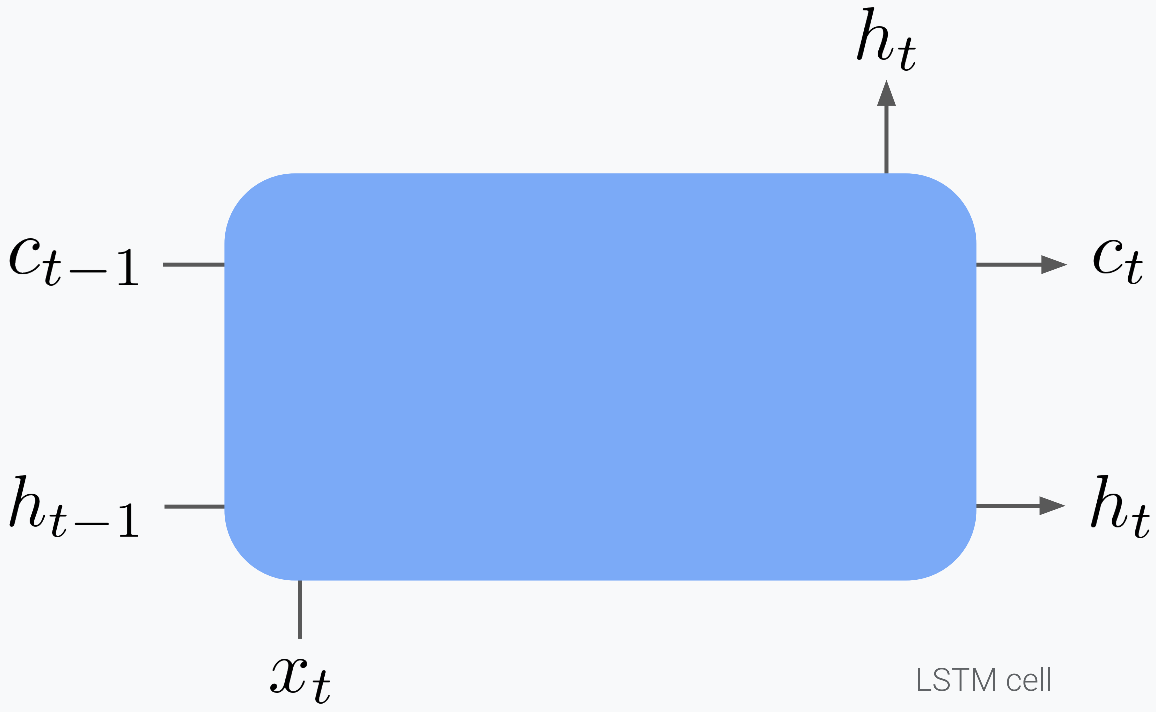


Going deep



LSTMs

Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation, 1997.



LSTM cell

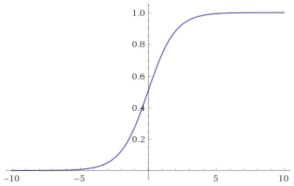
LSTMs

Addressing vanishing and exploding gradients

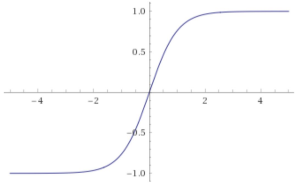
= dense layer

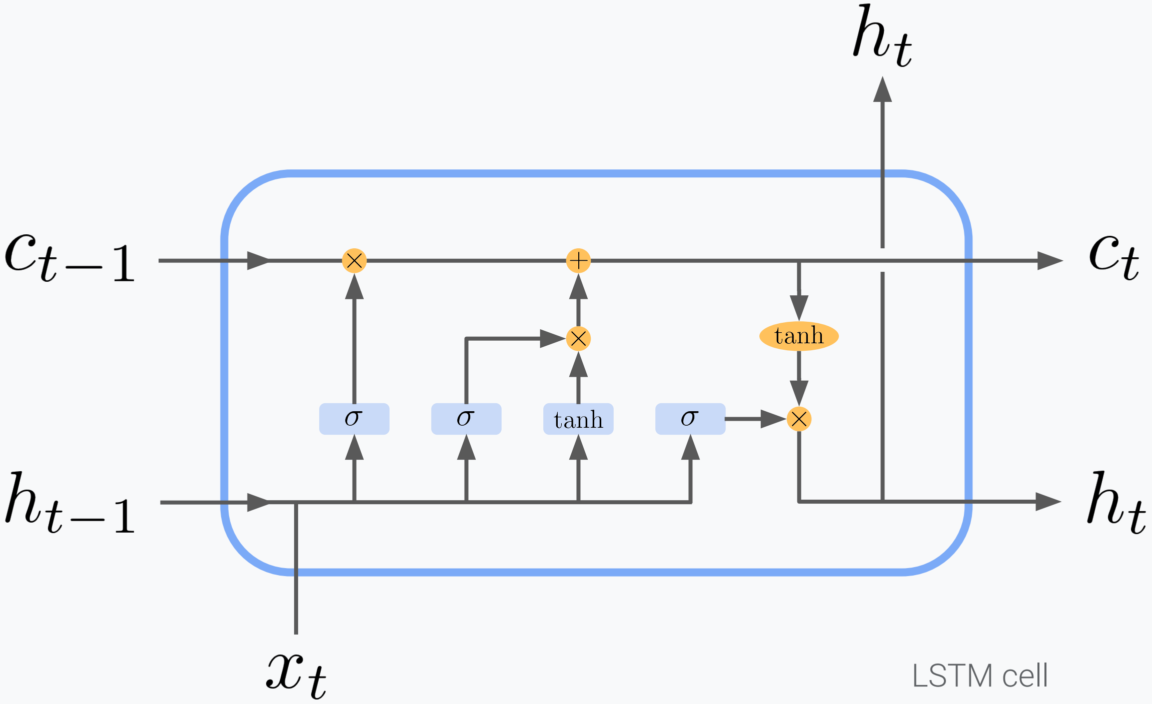
= pointwise operation

$$\sigma(x) = \frac{e^x}{e^x + 1}$$



$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

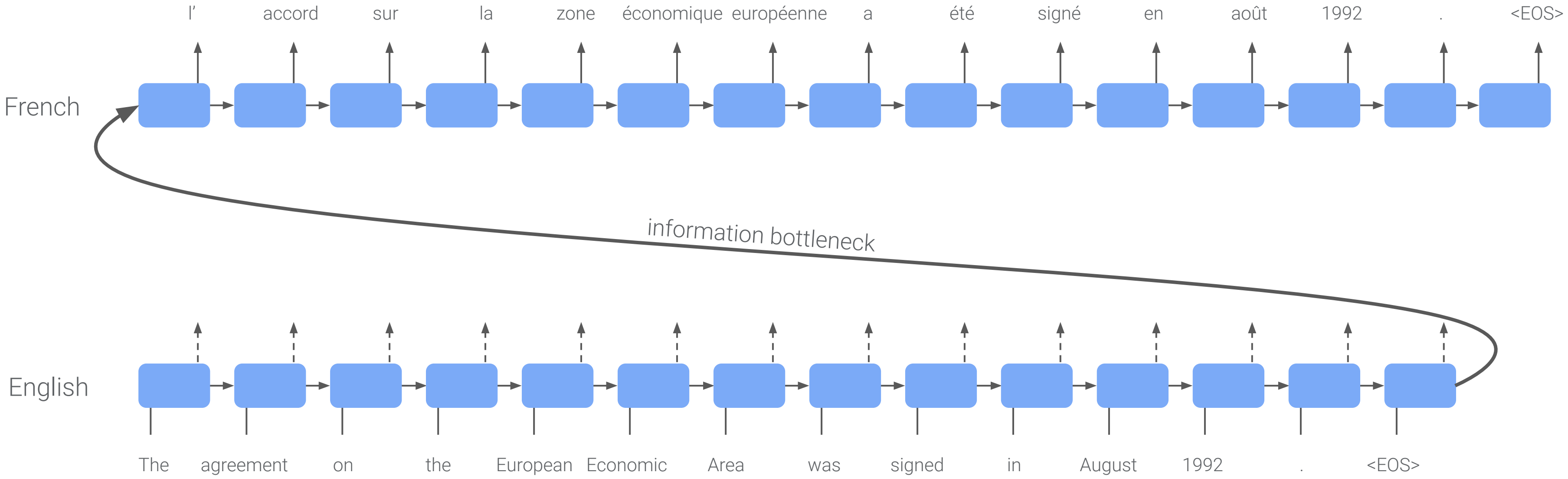




LSTM cell

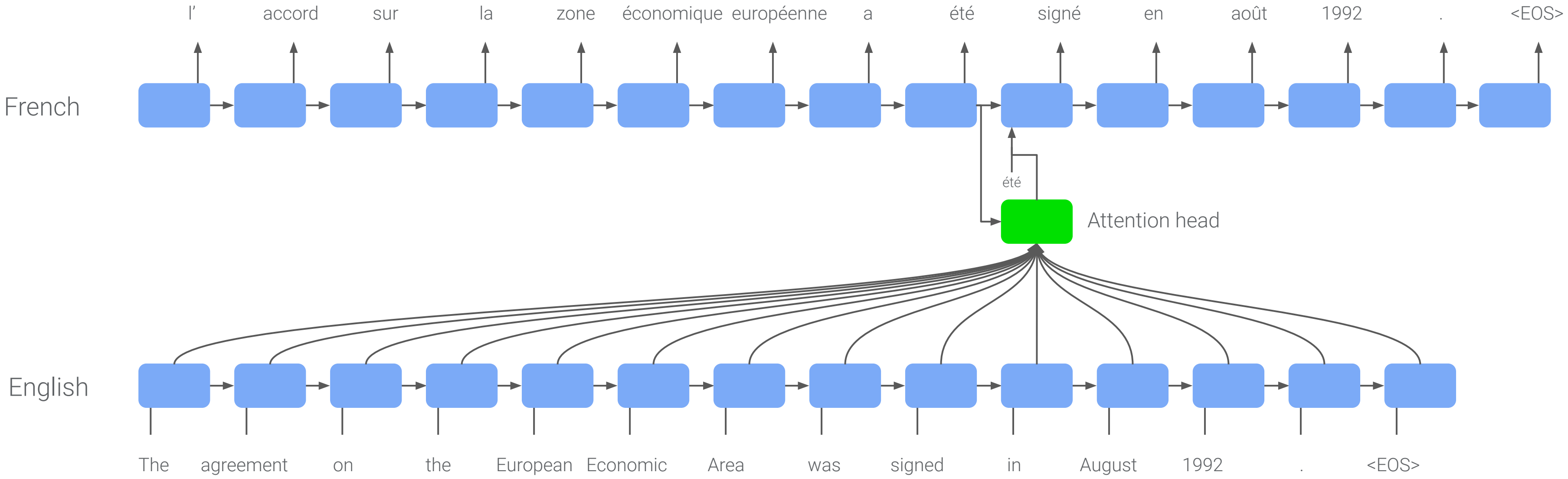
Attention

The encoder-decoder bottleneck



Example derived from Bahdanau, et al. 2014 (https://arxiv.org/pdf/1409.0473.pdf)

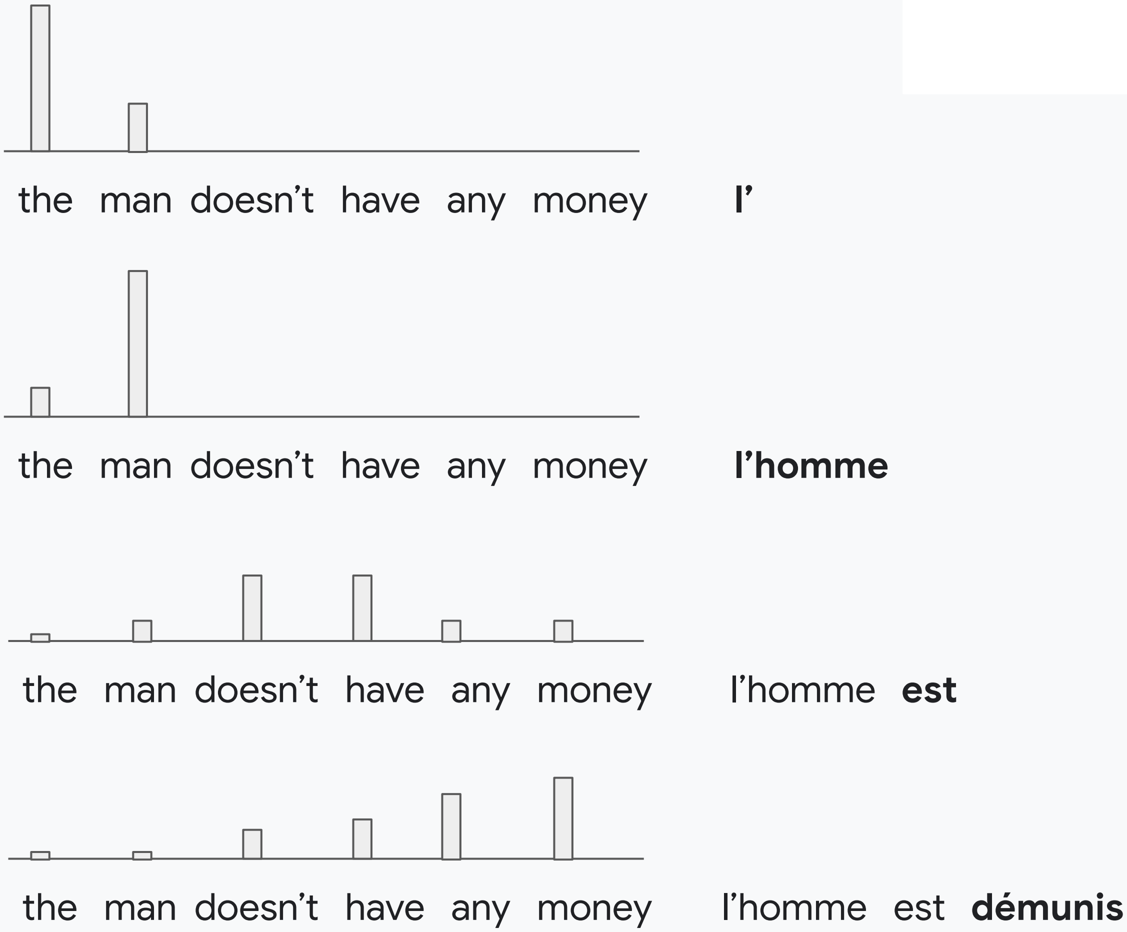
Attention



Example derived from Bahdanau, et al. 2014 (https://arxiv.org/pdf/1409.0473.pdf)

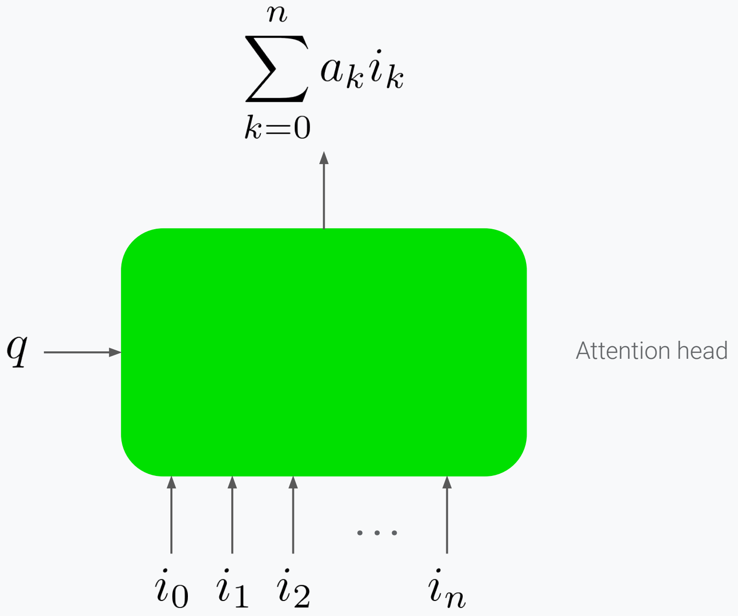
Attention mechanisms

Intuition on attention weights



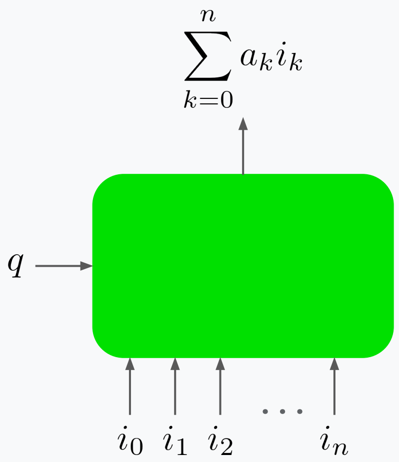
Luong attention

Thang Luong et al.
Effective approaches to attention-based neural machine translation. 2015



Luong attention

Thang Luong et al.
Effective approaches to
attention-based neural
machine translation. 2015



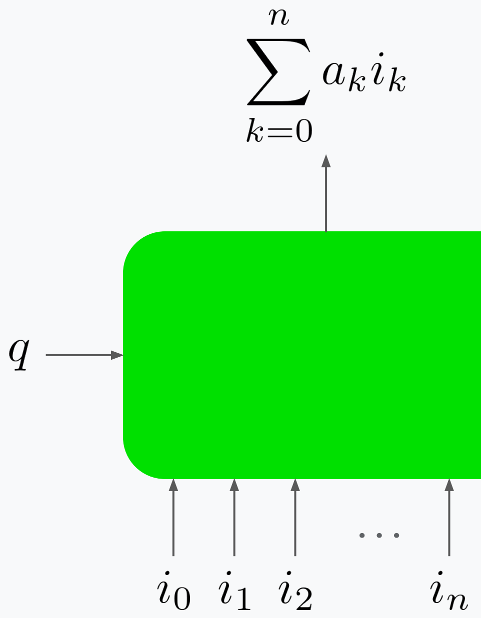
Attention head

$$a_k = \frac{e^{\phi(q, i_k)}}{\sum_{j=0}^{n} e^{\phi(q, i_j)}}$$

$$\phi(q, i_k) = \begin{cases} q^\top i_k & \text{(dot)} \\ q^\top W_a i_k & \text{(general)} \\ v_a \tanh(W_a[q^\top; i_k]) & \text{(concat)} \end{cases}$$

Luong dot product attention

Thang Luong et al.
Effective approaches to attention-based neural machine translation. 2015

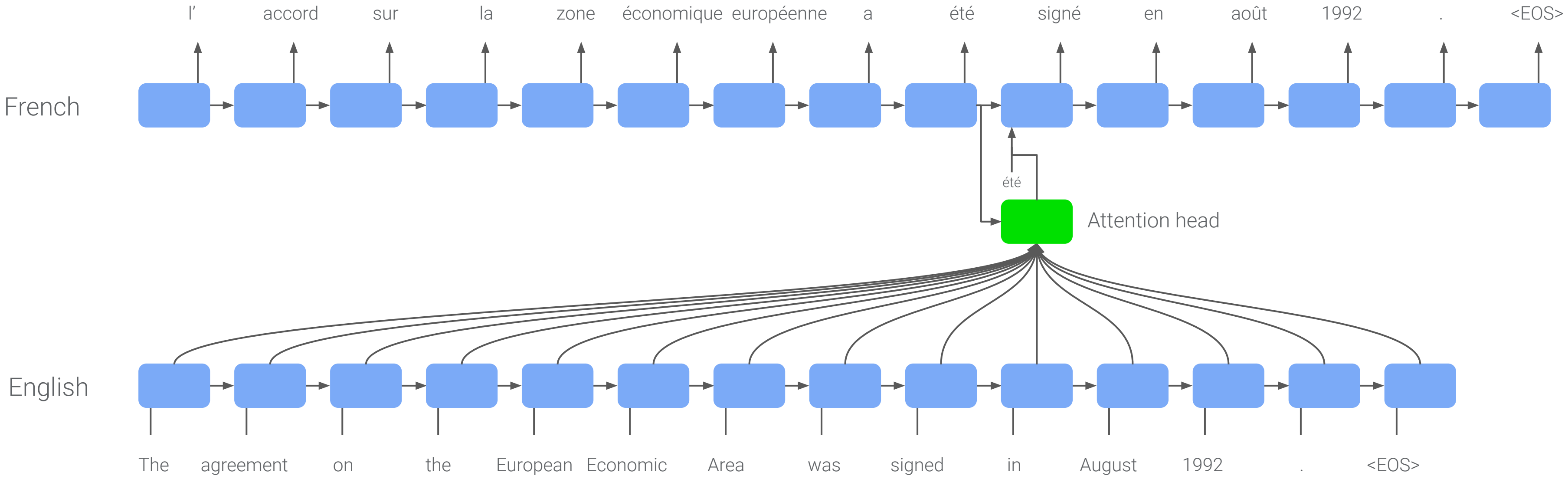


Attention head

$$a_k = \frac{e^{\phi(q, i_k)}}{\sum_{j=0}^{n} e^{\phi(q, i_j)}}$$

$$\phi(q, i_k) = q^\top i_k$$

Attention

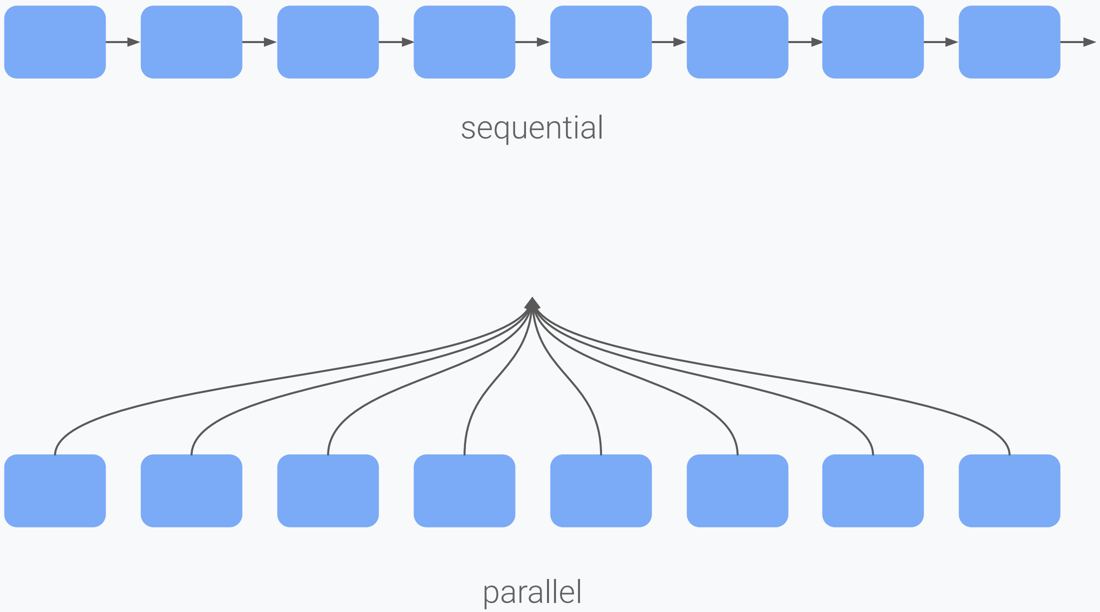


Example derived from Bahdanau, et al. 2014 (https://arxiv.org/pdf/1409.0473.pdf)

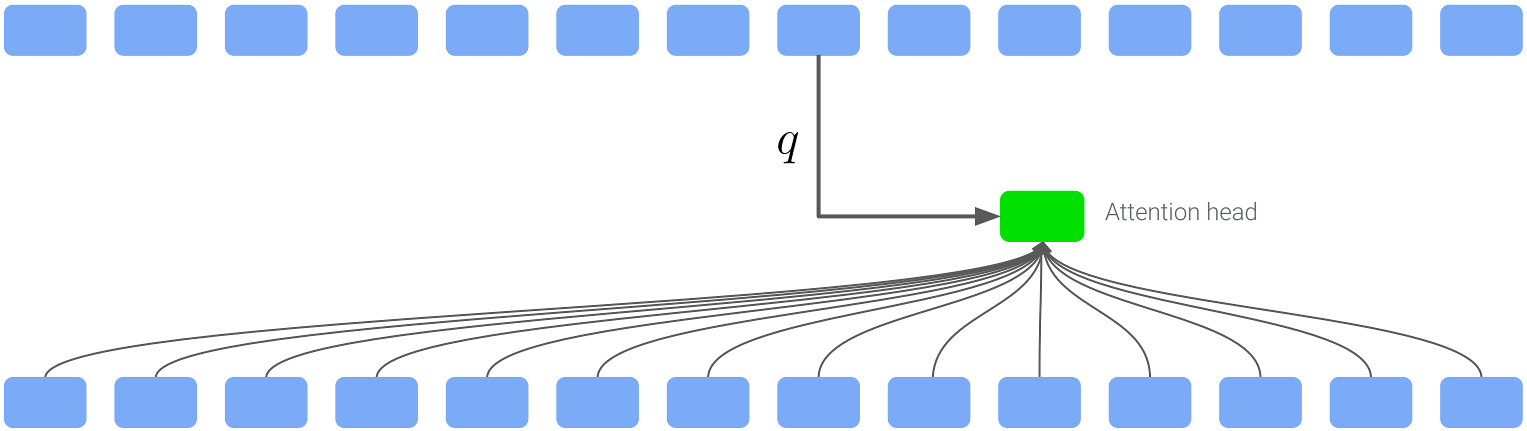
Transformers

Motivation

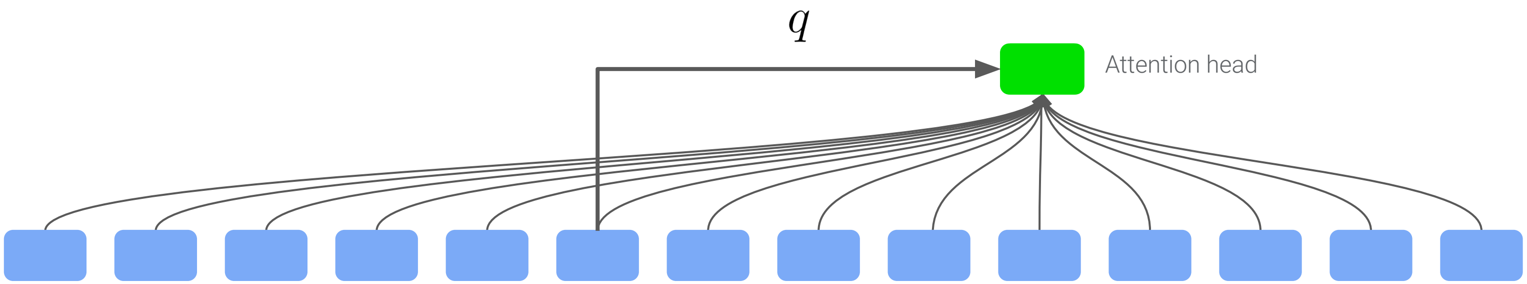
Sequential processing for RNNs can be a computational bottleneck



Attention

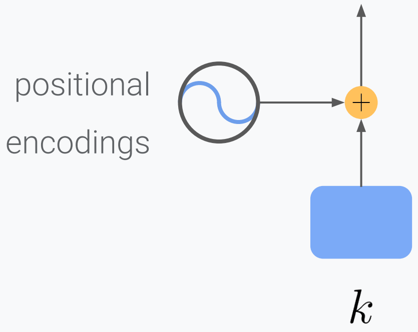


Self-Attention



Positional encodings

Note: positional encodings at two positions are a linear transformation away from each other



$$p(k, 2i) = \sin\left(\frac{k}{10000^{\frac{2i}{N}}}\right)$$

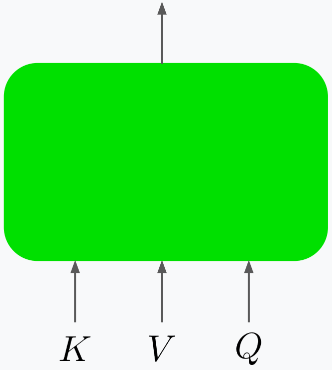
$$p(k, 2i+1) = \cos\left(\frac{k}{10000^{\frac{2i}{N}}}\right)$$

Scaled Dot-Product Attention

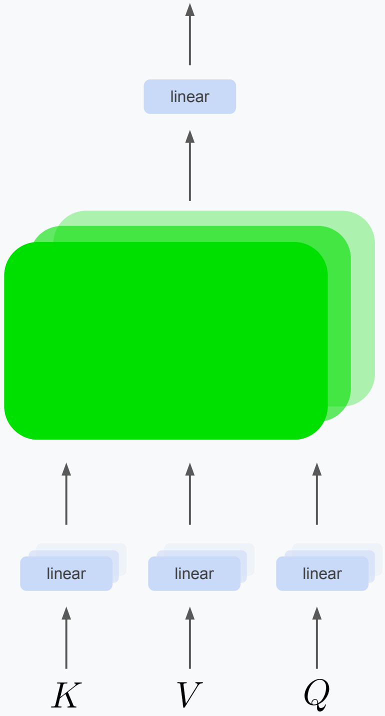
Queries keys and values

d_k is the dimensionality of the keys K

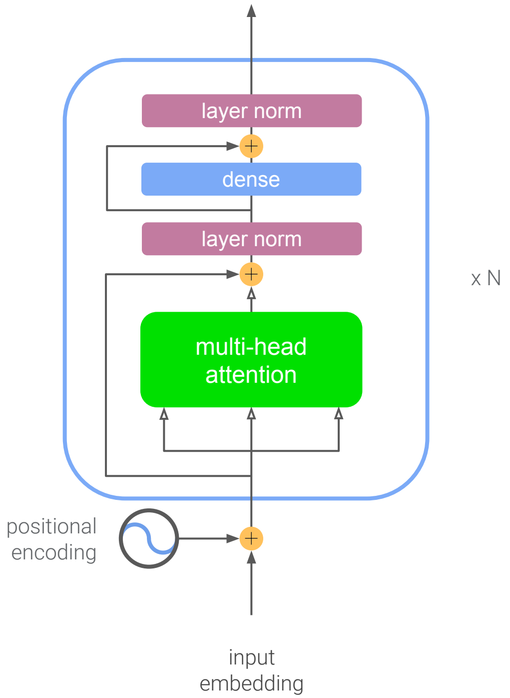
$$\mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

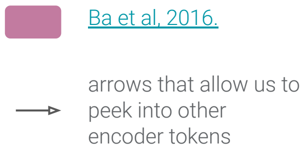


Multi-head attention

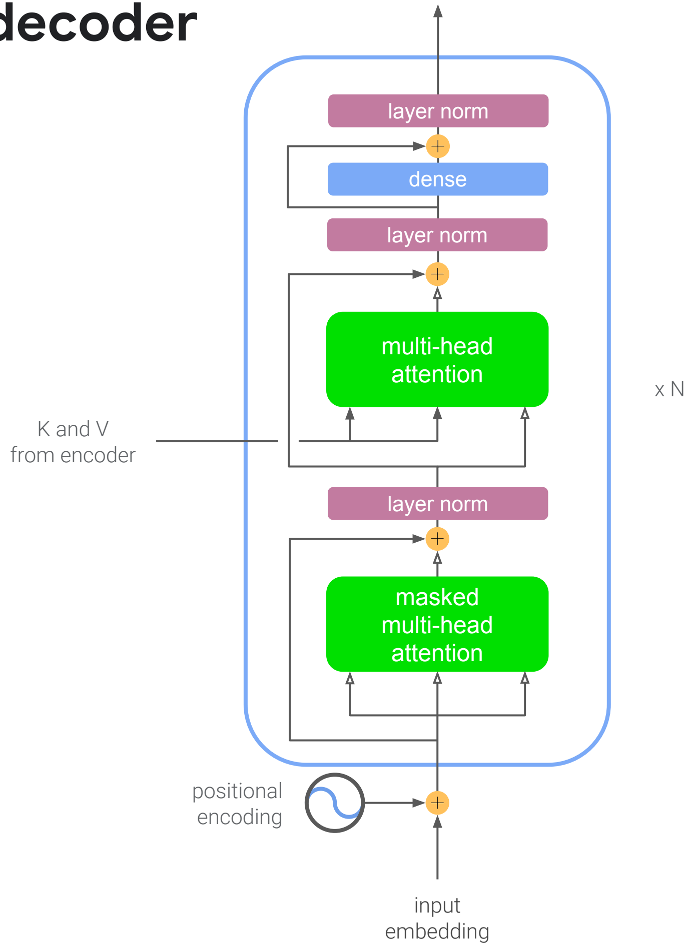


The transformer encoder





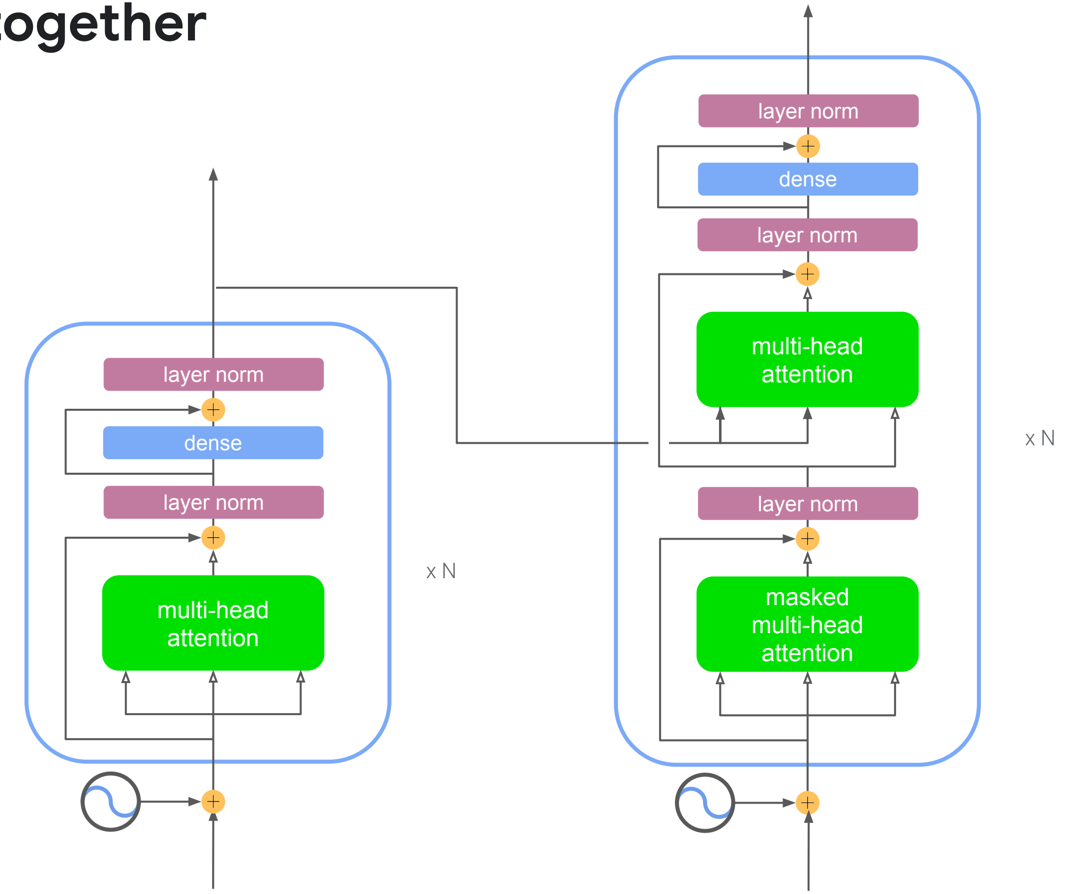
The transformer decoder




[Ba et al, 2016.](#)


arrows that allow us to peek into other decoder tokens

Putting it all together



LLaMA

LLaMA (Touvron et al., 2023)

LLaMA: Open and Efficient Foundation Language Models

Hugo Touvron[*], Thibaut Lavril[*], Gautier Izacard[*], Xavier Martinet
Marie-Anne Lachaux, Timothee Lacroix, Baptiste Rozière, Naman Goyal
Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin
Edouard Grave[*], Guillaume Lample[*]

Meta AI

LLaMA (Touvron et al., 2023)

```
Transformer(
    (tok_embeddings): ParallelEmbedding()
    (layers): ModuleList(
        (0-7): 8 x TransformerBlock(
            (attention): Attention(
                (wq): ColumnParallelLinear()
                (wk): ColumnParallelLinear()
                (wv): ColumnParallelLinear()
                (wo): RowParallelLinear()
            )
            (feed_forward): FeedForward(
                (w1): ColumnParallelLinear()
                (w2): RowParallelLinear()
                (w3): ColumnParallelLinear()
            )
            (attention_norm): RMSNorm()
            (ffn_norm): RMSNorm()
        )
    )
    (norm): RMSNorm()
    (output): ColumnParallelLinear()
)
```

LLaMA (Touvron et al., 2023)

Main changes from Vaswani et al., 2017:

- LayerNorm -> RMSNorm
- Rotary positional embeddings
- SwiGLU activations

LLaMA (Touvron et al., 2023) - Transformer Block

```python
class TransformerBlock(nn.Module):
    def __init__(self, layer_id: int, args: ModelArgs):
        super().__init__()
        self.n_heads = args.n_heads
        self.dim = args.dim
        self.head_dim = args.dim // args.n_heads
        self.attention = Attention(args)
        self.feed_forward = FeedForward(
            dim=args.dim, hidden_dim=4 * args.dim, multiple_of=args.multiple_of
        )
        self.layer_id = layer_id
        self.attention_norm = RMSNorm(args.dim, eps=args.norm_eps)
        self.ffn_norm = RMSNorm(args.dim, eps=args.norm_eps)

    def forward(self, x: torch.Tensor, start_pos: int, freqs_cis: torch.Tensor, mask: Optional[torch.Tensor]):
        h = x + self.attention.forward(self.attention_norm(x), start_pos, freqs_cis, mask)
        out = h + self.feed_forward.forward(self.ffn_norm(h))
        return out
```

LLaMA (Touvron et al., 2023) - Attention

```
class Attention(nn.Module):
    def forward(self, x: torch.Tensor, start_pos: int, freqs_cis: torch.Tensor, mask: Optional[torch.Tensor]):
        bsz, seqlen, _ = x.shape
        xq, xk, xv = self.wq(x), self.wk(x), self.wv(x)

        xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
        xk = xk.view(bsz, seqlen, self.n_local_heads, self.head_dim)
        xv = xv.view(bsz, seqlen, self.n_local_heads, self.head_dim)

        xq, xk = apply_rotary_emb(xq, xk, freqs_cis=freqs_cis)

        xq = xq.transpose(1, 2)
        keys = xk.transpose(1, 2)
        values = xv.transpose(1, 2)
        scores = torch.matmul(xq, keys.transpose(2, 3)) / math.sqrt(self.head_dim)
        if mask is not None:
            scores = scores + mask  # (bs, n_local_heads, slen, slen)
        scores = F.softmax(scores.float(), dim=-1).type_as(xq)
        output = torch.matmul(scores, values)  # (bs, n_local_heads, slen, head_dim)
        output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)

        return self.wo(output)
```

LLaMA (Touvron et al., 2023) - FeedForward

```
class FeedForward(nn.Module):
   def forward(self, x):
       return self.w2(F.silu(self.w1(x)) * self.w3(x))
```

LLaMA (Touvron et al., 2023) - The transformer

```
class Transformer(nn.Module):
    def __init__(self, params: ModelArgs):
        super().__init__()
        self.params = params
        self.vocab_size = params.vocab_size
        self.n_layers = params.n_layers

        self.tok_embeddings = ParallelEmbedding(
            params.vocab_size, params.dim, init_method=lambda x: x
        )

        self.layers = torch.nn.ModuleList()
        for layer_id in range(params.n_layers):
            self.layers.append(TransformerBlock(layer_id, params))

        self.norm = RMSNorm(params.dim, eps=params.norm_eps)
        self.output = ColumnParallelLinear(
            params.dim, params.vocab_size, bias=False, init_method=lambda x: x
        )

        self.freqs_cis = precompute_freqs_cis(
            self.params.dim // self.params.n_heads, self.params.max_seq_len * 2
        )
```

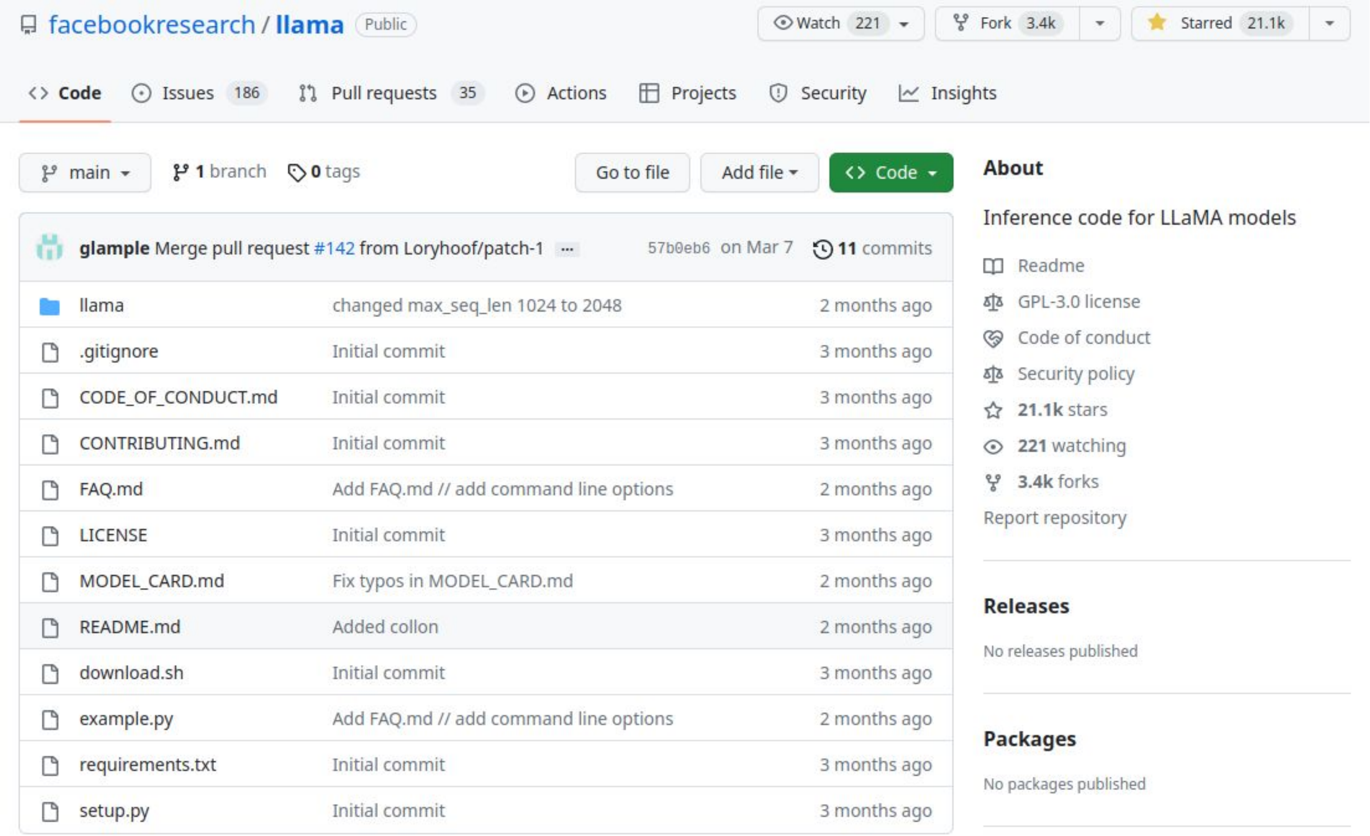
LLaMA (Touvron et al., 2023) - The transformer

```
@torch.inference_mode()
def forward(self, tokens: torch.Tensor, start_pos: int):
    _bsz, seqlen = tokens.shape
    h = self.tok_embeddings(tokens)
    self.freqs_cis = self.freqs_cis.to(h.device)
    freqs_cis = self.freqs_cis[start_pos : start_pos + seqlen]

    mask = None
    if seqlen > 1:
        mask = torch.full((1, 1, seqlen, seqlen), float("-inf"), device=tokens.device)
        mask = torch.triu(mask, diagonal=start_pos + 1).type_as(h)

    for layer in self.layers:
        h = layer(h, start_pos, freqs_cis, mask)
    h = self.norm(h)
    output = self.output(h[:, -1, :])  # only compute last logits
    return output.float()
```

LLaMA (Touvron et al., 2023) - Codebase



HW2: Make it train!