Attention and transformers

Sequence modelling

- Text-to-text translation
- Speech-to-speech translation
- Code-to-code translation
- Time series

- What it is: A set of symbols, i.e. embeddings (of words, sounds etc)

Sequence modelling - historical

- RNN
- LSTM

Problem

- Time series depend on past values - OK
- Grammar also depends on future intention
- Example: I saw a/an (?) → man/elephant (aha!)

Historical solution

- 1) Run RNN/LSTM from left to right
- 2) Run RNN/LSTM from right to left
- 3) Provide both outputs as input to next layer

- It works!

Attention

- Idea: Focus processing on the important parts no matter where they are.
- LSTM -> 1) Store important part 2) retrieve it where it is important
- Atttention → 1) Directly input parts that are most important for each new symbol

• Classic database

- Query: «Star Wars»
- Key: <Title of each movie>
- Value: video.mpg

- 1) Match query against each key. 2) Identify the best matching key 3) Return the corresponding video.mpg file/

Attention

- Components: Key, query, value

- Query: Which symbol am I looking for?
- Key: What am I looking at in each symbol?
- Score: How to weight the value
- Value: The actual value of the symbol

Attention example: self-attention

- Self-: Key, query, value are all based on X

- Query: $Q = W_q X$
- Key: $K = W_k X$
- Score: softmax(QK/n)
- Value: $W_V X$
- Output: $W_V X$ softmax(QK)

Attention example explained

- Components: Key, query, value

- Query: $Q = W_q X$ - looks for what is important to a symbol
- Key: $K = W_k X$ — looks for which other symbols match he query
- Score:$S = \mathrm{softmax}(QK/c)$ — Checks which query is matching
- (c is some constant, usually $c = \mathrm{sqrt}(d_K)$)
- Value: $V = W_V X$ — The value (+ linear transform)
- Output: $Y = V\ S$ — Returns the value corresponding to the best matching key

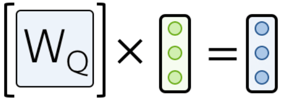
Self-Attention

$$Attention(q, k, v) = \overbrace{softmax\left(\frac{qk^T}{\sqrt{d_k}}\right)}^{\text{Attention weights}} v$$

from (q) to (k, v)

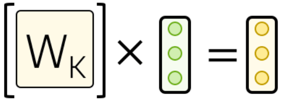
vector dimensionality of K, V (d_k)

Each vector receives three representations ("roles")



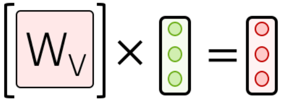
Query: vector **from** which the attention is looking

"Hey there, do you have this information?"



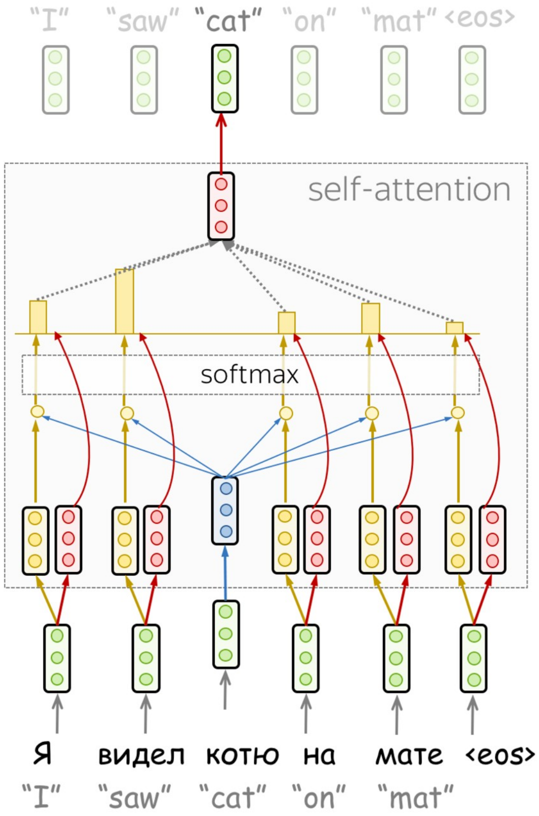
Key: vector **at** which the query looks to compute weights

"Hi, I have this information – give me a large weight!"



Value: their weighted sum is attention output

"Here's the information I have!"



Slide credit: Lena Voita

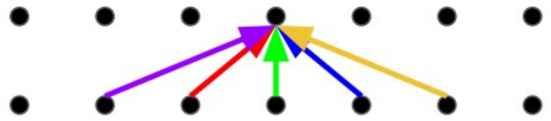
Attention example explained

What's missing from Self-Attention?

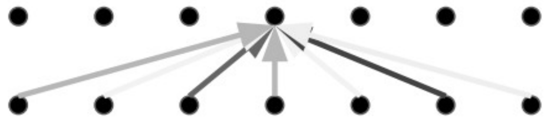
- Convolution: a different linear transformation for each relative position. Allows you to distinguish what information came from where.

- Self-Attention: a weighted average :(

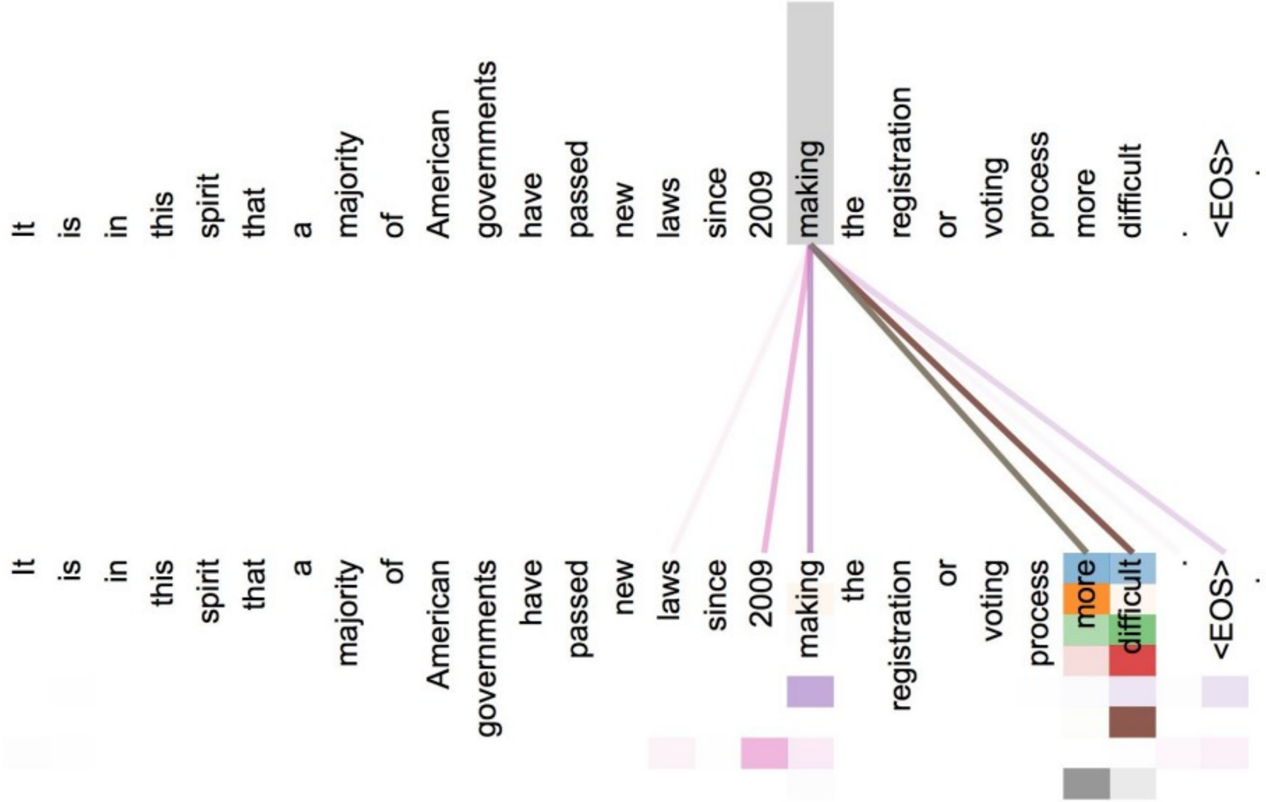
Convolution



Self-Attention



The Fix: Multi-Head Attention



Multihead attention

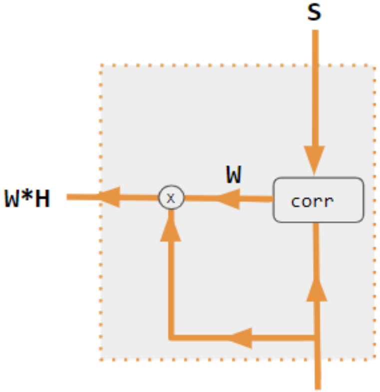
- We look for several values and concatenate
- MH(Q,K,V) = concat(head1(Q,K,V), head2(Q,K,V)) W_O

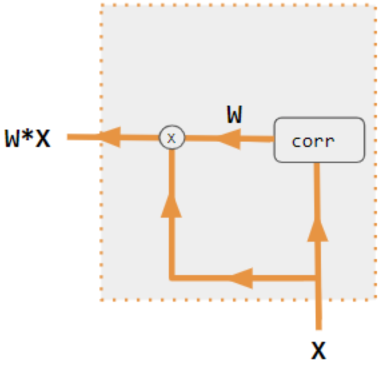
- Example:
- MH(X) = concat($head_1$(X), $head_2$(X))

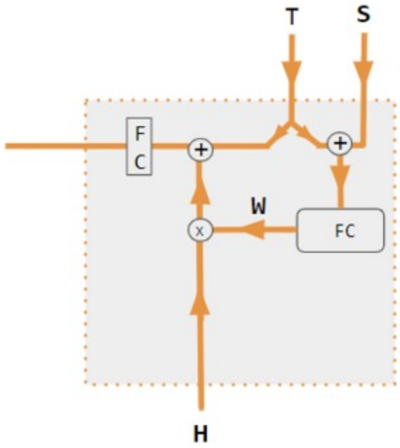
Summary so far

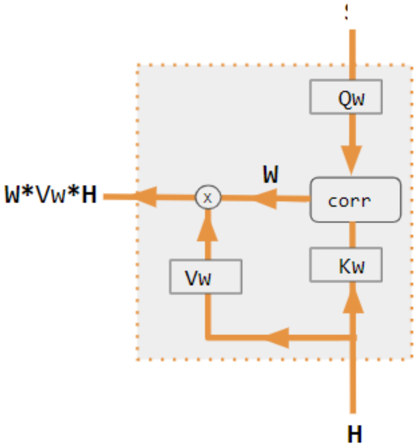
- Y = Attention(K,Q,V) sequence of length L
- Multihead attention = (head1, head2 ...) W
- LxO where O is output dim
- There are different types of attention (inputs, score functions, non-linearities etc)
- Self-attention is K=Q=V (=X)

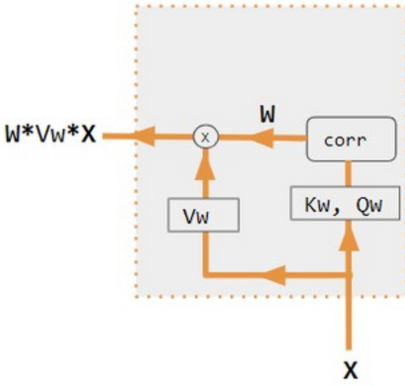
Other attention models







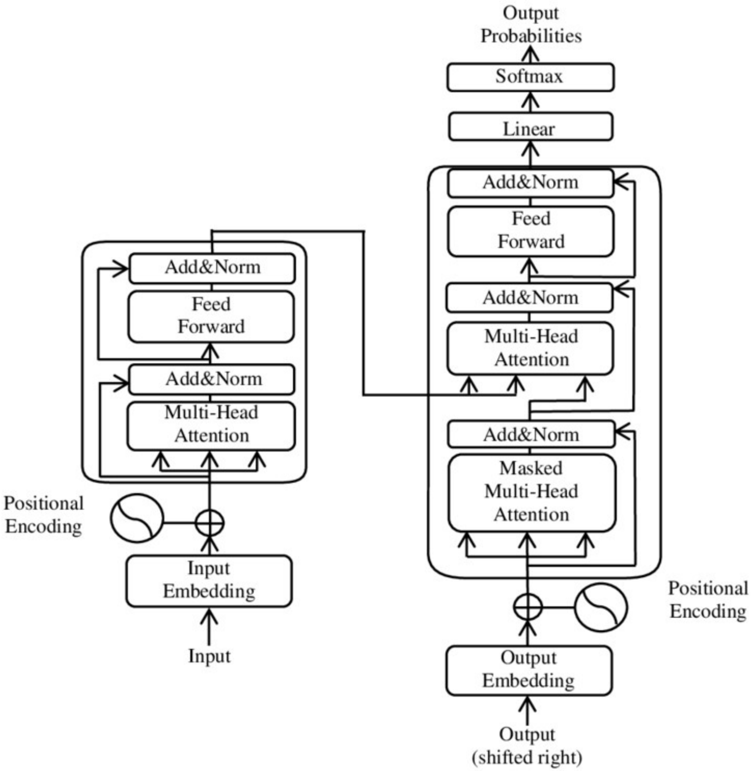




Transformer

- Attention is all you need (almost)
- Transformer: Layered self-attention
- An element in a sequence: a «symbol»
- Each layer:
  - N input symbols → N output symbols
  - Dimensions: $N*d_x \rightarrow N*d_y$

Transformer (original)



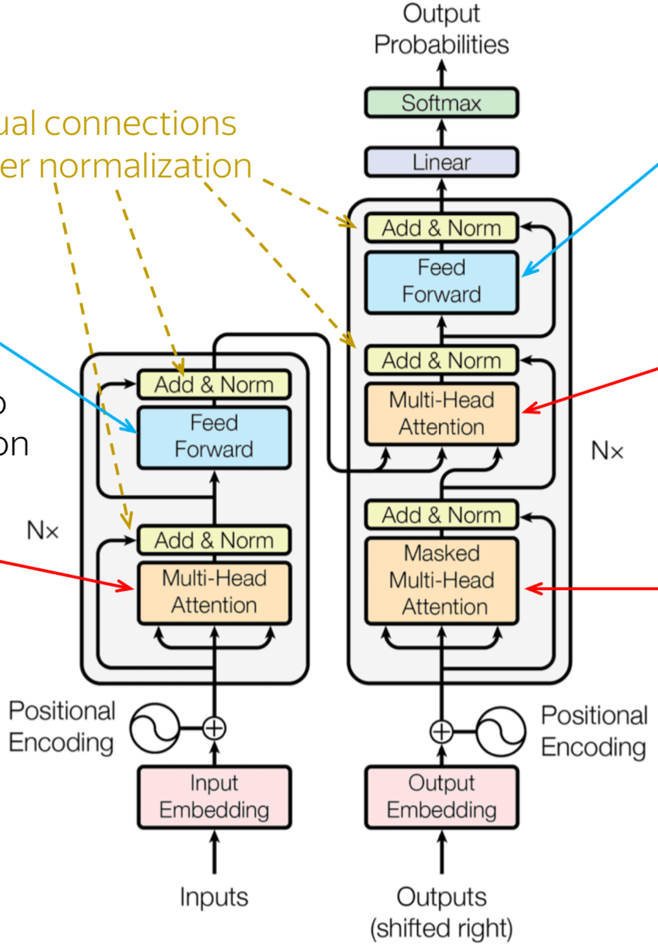
Transformer (annotated)

Residual connections
and layer normalization

Feed-forward network:
after taking information from
other tokens, take a moment to
think and process this information

Encoder self-attention:
tokens look at each other

queries, keys, values
are computed from
encoder states



Feed-forward network:
after taking information from
other tokens, take a moment to
think and process this information

Decoder-encoder attention:
target token looks at the source

queries – from decoder states; keys
and values from encoder states

Decoder self-attention (masked):
tokens look at the previous tokens

queries, keys, values are computed
from decoder states

Slide credit: Lena Voita

Decoder-encoder attention

- Idea: For each symbol in the decoder we perform one «search» for relevant information among the encoder symbols

- Query: Symbols from previous layer of decoder, D
- Key, value: Symbols from encoder, E
- Output: N_D symbols, i.e. same number as previous decoder layer

- Query: $Q = W_q D$
- Key: $K = W_k E$
- Score: softmax(QK/c)
- Value: $W_V E$
- Output: $W_V X$ softmax(QK) (i.e. value*score)

New blocks: Add&Norm

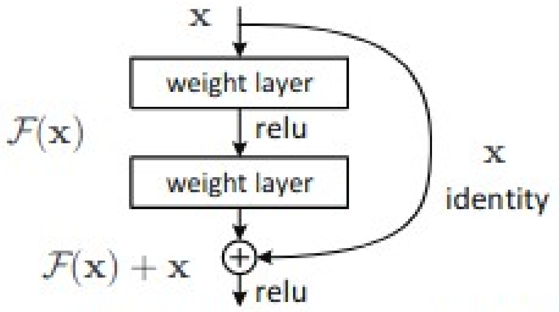
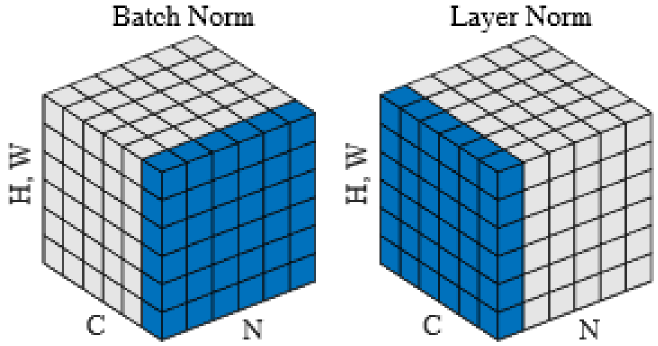


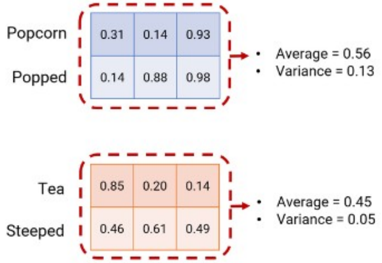
Figure 2. Residual learning: a building block.



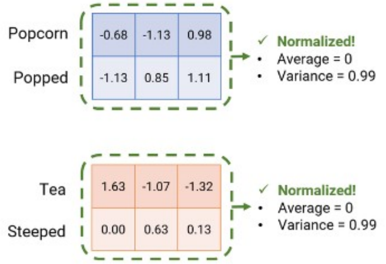
For each output: Sum of outputs for each sample sums up to 1 with unit variance.

Note: This calculation is applied *once for each symbol.*
N symbols create a N x d_o sized output for the layer, where d_o is the output size of the neural network.

Layer normalization



And again, after normalization, we'll have matrices with average of 0 and variance of 1:



New blocks: Embedding

Some pretrained language model that converts each word to a vector of fixed size, e.g. [0.4 0.5 0.7]

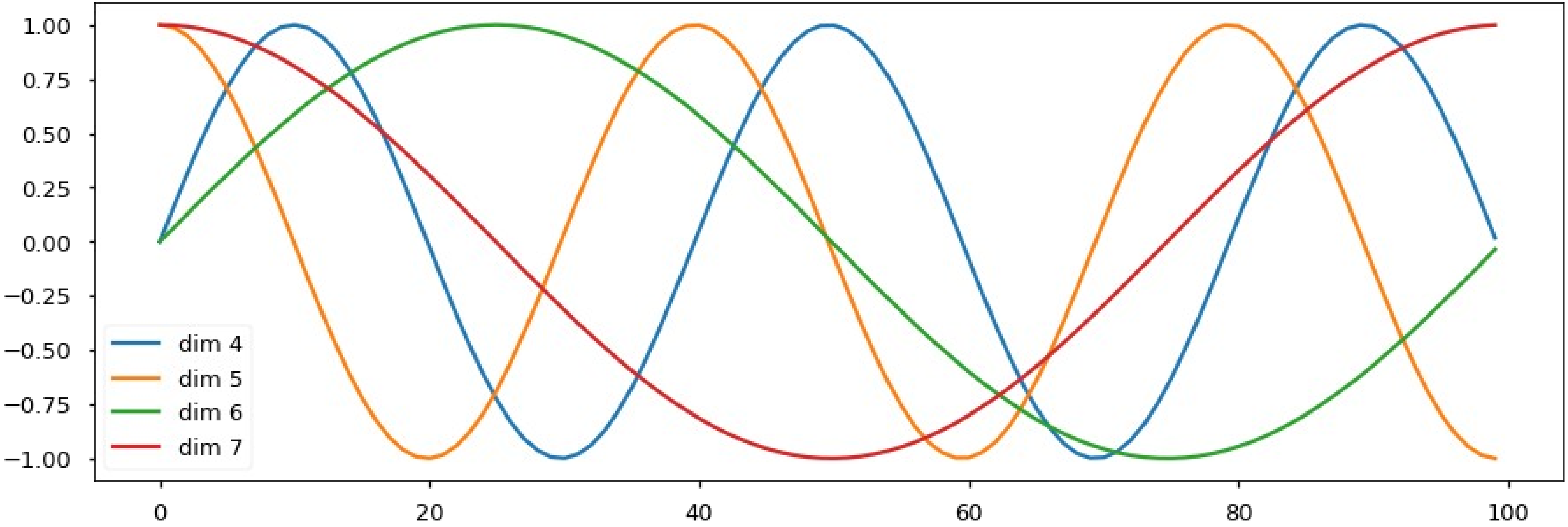
New blocks: Masking

Prevent attention to unwanted words. In particular, future words.

softmax(QK^T + M)

$$M = \begin{array}{c c} & \begin{array}{cccc} a^K & b^K & c^K & D^K \end{array} \\ \begin{array}{c} a^Q \\ b^Q \\ c^Q \\ D^Q \end{array} & \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

New blocks: Positional encoding



sin + cos transforms: sin(pos/10000$^{2i/d_model}$)
Adds information about the position to the word embedding vector
(10 k is just «large number») to avoid cycles

A continous form of:
0 0 0 1 0 0
0 0 1 1 0 1
0 1 0 1 1 0
0 1 1 1 1 1

New blocks: Positional encoding

$$\vec{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k + 1 \end{cases}$$

Why d-dimensional?

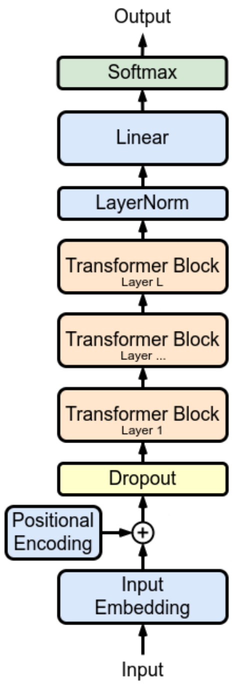
Oddity: Add position encoding to word embedding (instead of concatenating)

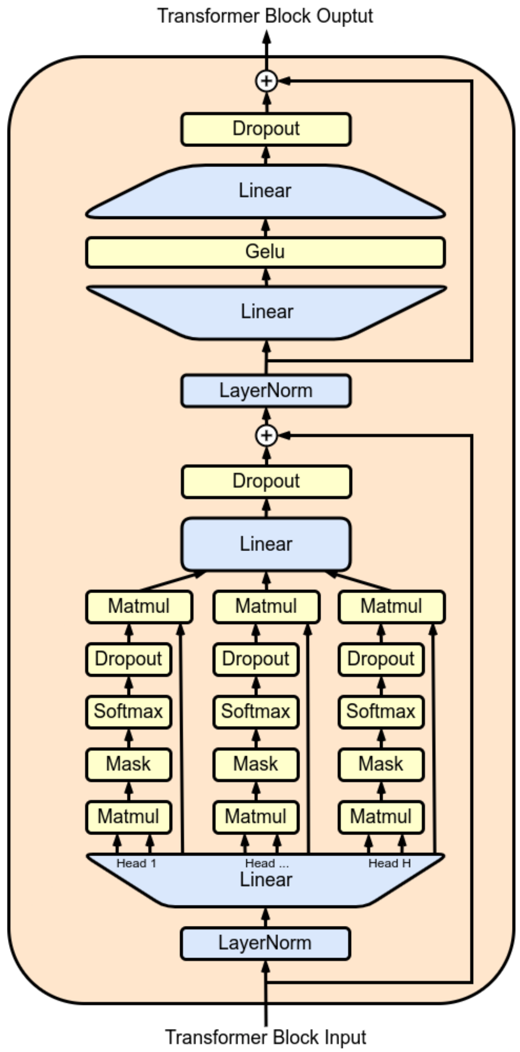
DeBERTa:
«the disentangled attention mechanism, where each word is represented using two vectors that encode its content and position…performs consistently better»

GPT2

and GPT3...

We use the same model and architecture as GPT-2 [RWC+19], including the modified initialization, pre-normalization, and reversible tokenization described therein, with the exception that we use alternating dense and locally banded sparse attention patterns in the layers of the transformer, similar to the Sparse Transformer [CGRS19]. To study the dependence of ML performance on model size, we train 8 different sizes of model, ranging over three orders of magnitude from 125 million parameters to 175 billion parameters, with the last being the model we call GPT-3. Previous work [KMH+20] suggests that with enough training data, scaling of validation loss should be approximately a smooth power law as a function of size; training models of many different sizes allows us to test this hypothesis both for validation loss and for downstream language tasks.





New blocks: Embedding

Some pretrained language model that converts each word to a vector of fixed size, e.g. [0.4 0.5 0.7]

RNN vs LSTM vs Self-attention

n: sequence length, d:encoding length

Multi-Head Attention with linear transformations. For each of the h heads, $d_q = d_k = d_v = d/h$	$n^2 \cdot d + n \cdot d^2$
Recurrent	$n \cdot d^2$
Convolutional	$n \cdot d^2$

Simplest: RNN
Most scalable with sequence length: LSTM/RNN
Fastest on typical hardware if n<<d: Self-attention
Fastest on sparse bandwidth: Self-attention

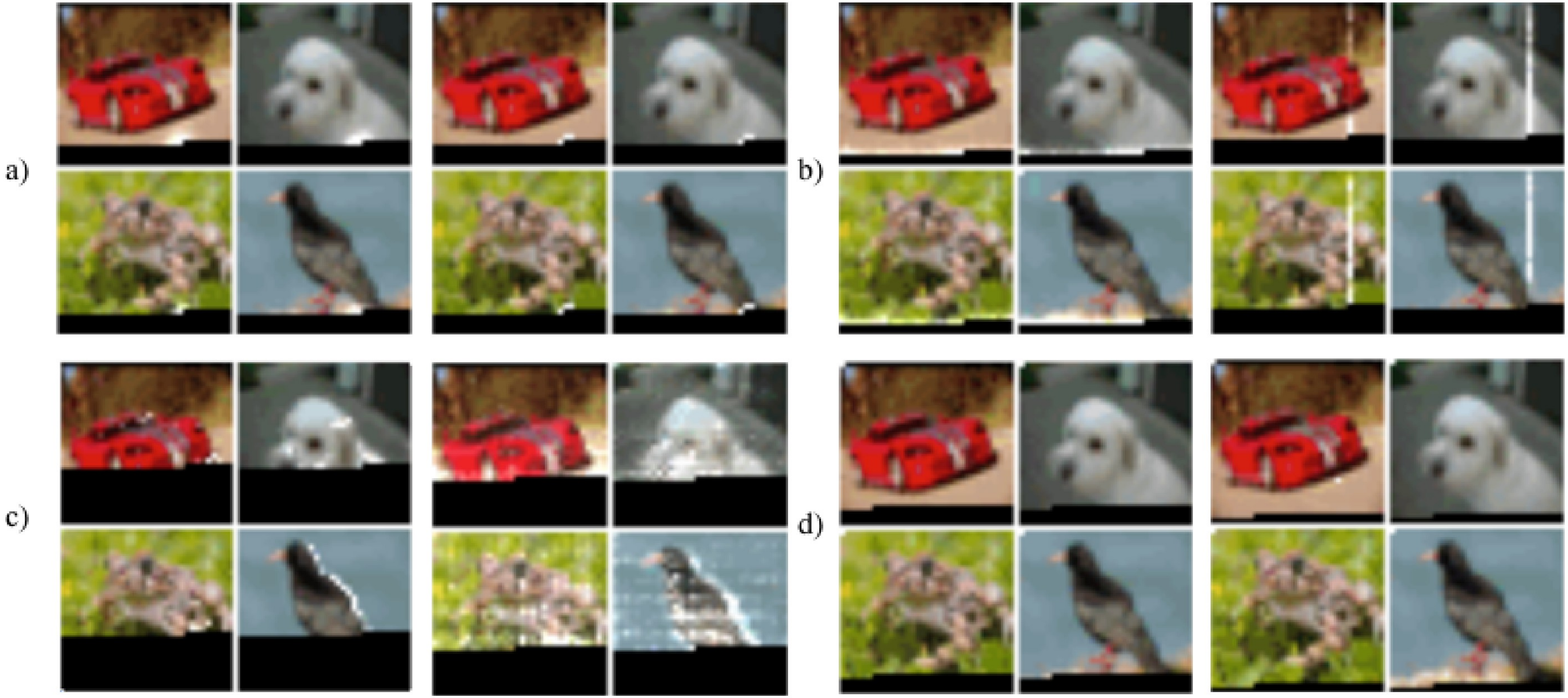
However, for training gradient quality and landscape might be more important

RNN vs LSTM vs Self-attention: Practical results

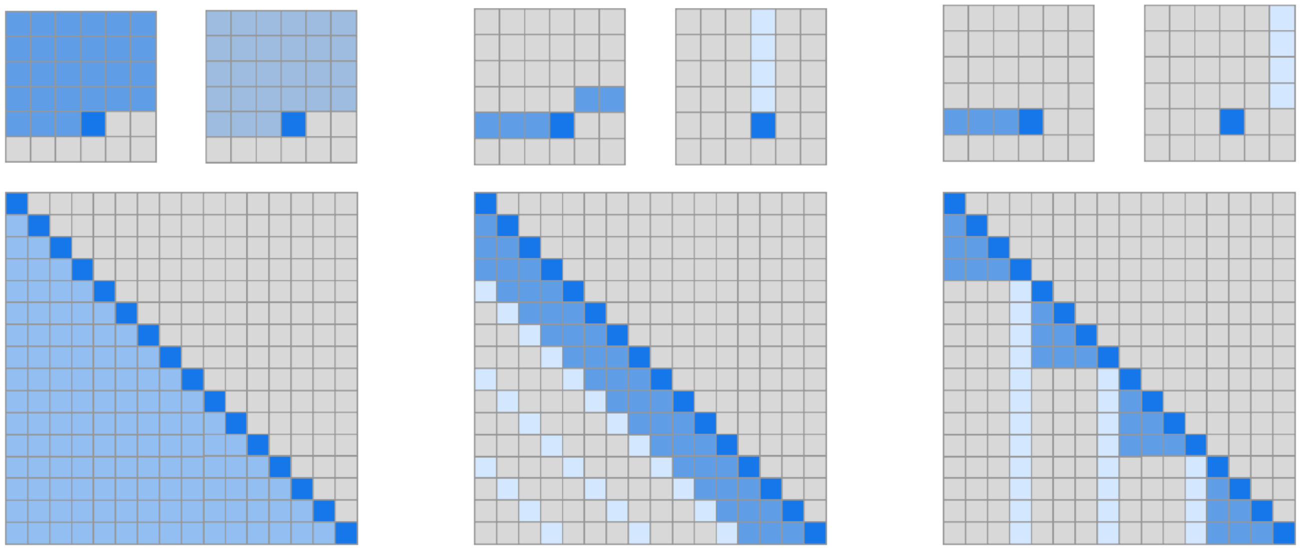
Machine Translation Results: WMT-14

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [17]	23.75			
Deep-Att + PosUnk [37]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [36]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [31]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [37]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [36]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

Sparse attention (white)

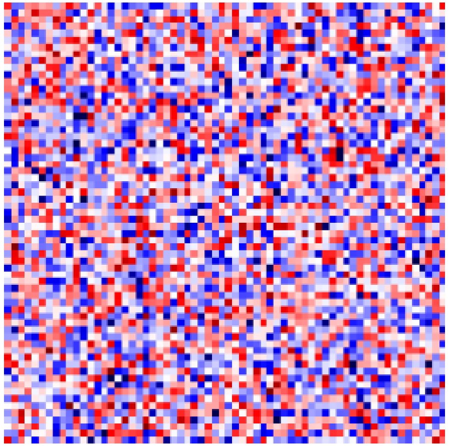


Sparse transformers

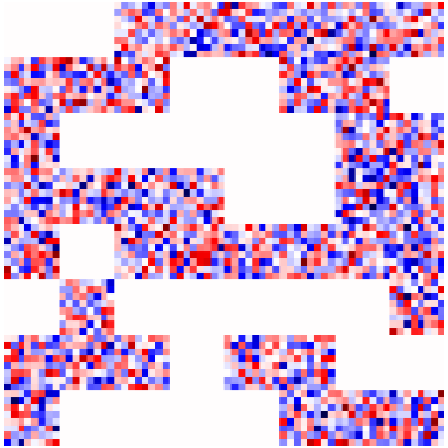


(a) Transformer (b) Sparse Transformer (strided) (c) Sparse Transformer (fixed)

Block-sparse (on GPU)



Dense weights



Block-sparse weights

0	0	1	1	1	1	1	1
1	1	1	0	0	1	1	0
1	0	0	0	0	0	1	1
1	1	1	1	0	0	1	1
1	0	1	1	1	1	1	1
0	1	0	0	0	0	0	0
1	1	1	0	1	1	0	0
1	0	0	0	0	1	1	1

Corresponding sparsity pattern

GPT2 code

```
def model(hparams, X, past=None, scope='model', reuse=False):
    with tf.variable_scope(scope, reuse=reuse):
        results = {}
        batch, sequence = shape_list(X)

        wpe = tf.get_variable('wpe', [hparams.n_ctx, hparams.n_embd],
                             initializer=tf.random_normal_initializer(stddev=0.01))
        wte = tf.get_variable('wte', [hparams.n_vocab, hparams.n_embd],
                             initializer=tf.random_normal_initializer(stddev=0.02))
        past_length = 0 if past is None else tf.shape(past)[-2]
        h = tf.gather(wte, X) + tf.gather(wpe, positions_for(X, past_length))

        # Transformer
        presents = []
        pasts = tf.unstack(past, axis=1) if past is not None else [None] * hparams.n_layer
        assert len(pasts) == hparams.n_layer
        for layer, past in enumerate(pasts):
            h, present = block(h, 'h%d' % layer, past=past, hparams=hparams)
            presents.append(present)
        results['present'] = tf.stack(presents, axis=1)
        h = norm(h, 'ln_f')

        # Language model loss.  Do tokens <n predict token n?
        h_flat = tf.reshape(h, [batch*sequence, hparams.n_embd])
        logits = tf.matmul(h_flat, wte, transpose_b=True)
        logits = tf.reshape(logits, [batch, sequence, hparams.n_vocab])
        results['logits'] = logits
        return results
```