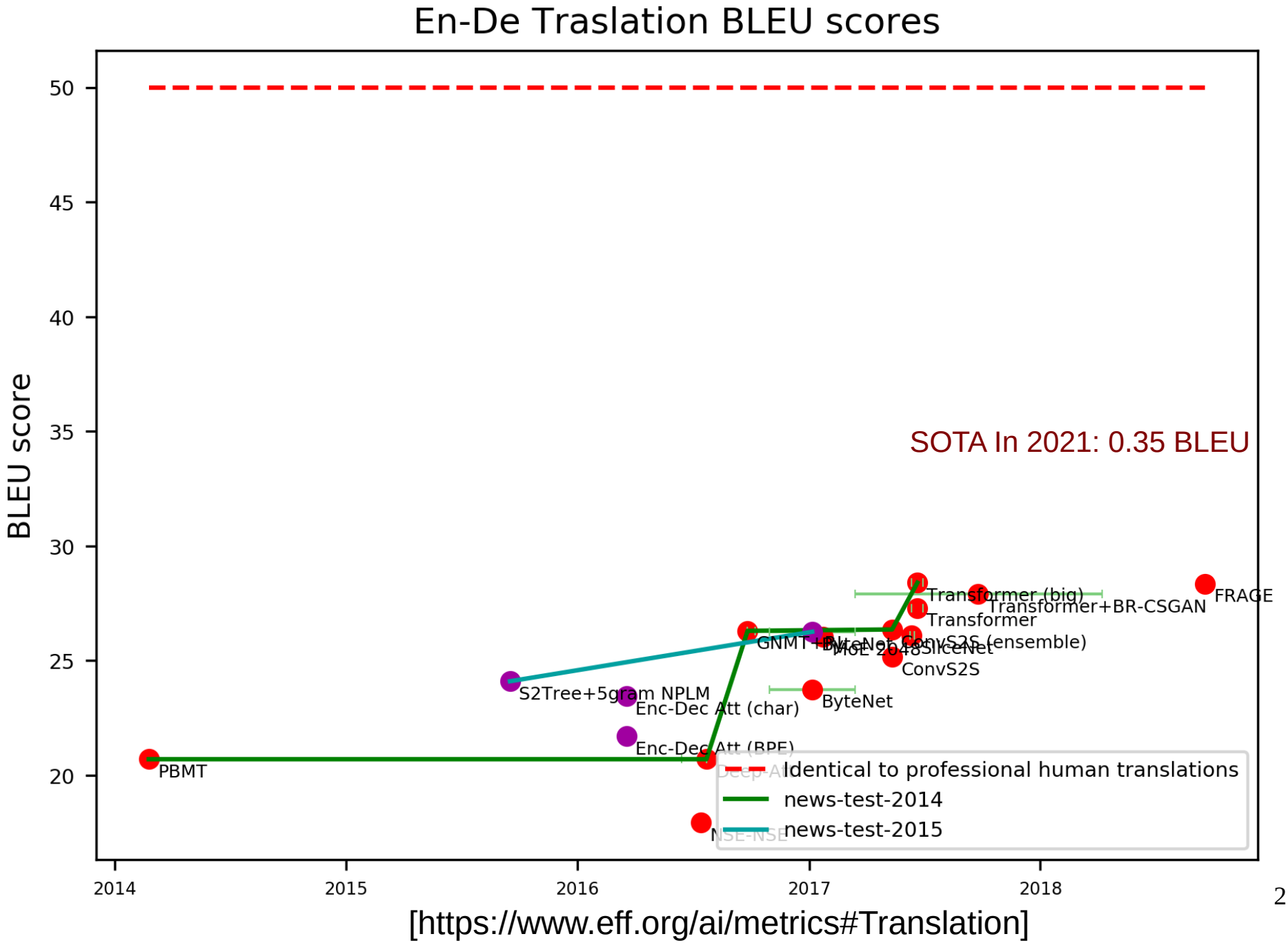
Transformer

Nikolay Arefyev
CMC MSU Department of Algorithmic Languages &
Samsung Moscow Research Center

MT vs. Human translation



[https://www.eff.org/ai/metrics#Translation]

Get rid of RNNs in MT?

- RNNs are slow, because not parallelizable over timesteps
- Attention is parallelizable + have shorter gradient paths
  - Sequence transduction w/o RNNs/CNNs (attention+FF)
  - SOTA on En→Ge WMT14, better than any single model on En→Fr WMT14 (but worse than ensembles)
  - Much faster than other best models (base/big: 12h/3.5d on 8GPUs)

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

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	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 [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	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	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

*Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention and started the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first Transformer models and has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention and the parameter-free position representation and became the other person involved in nearly every detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase and tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial codebase, and efficient inference and visualizations. Lukasz and Aidan spent countless long days designing various parts of and implementing tensor2tensor, replacing our earlier codebase, greatly improving results and massively accelerating our research.

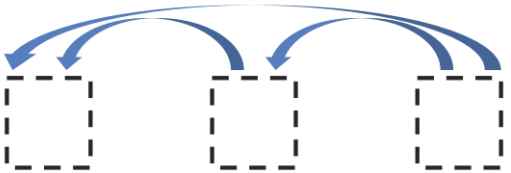
†Work performed while at Google Brain.

‡Work performed while at Google Research.

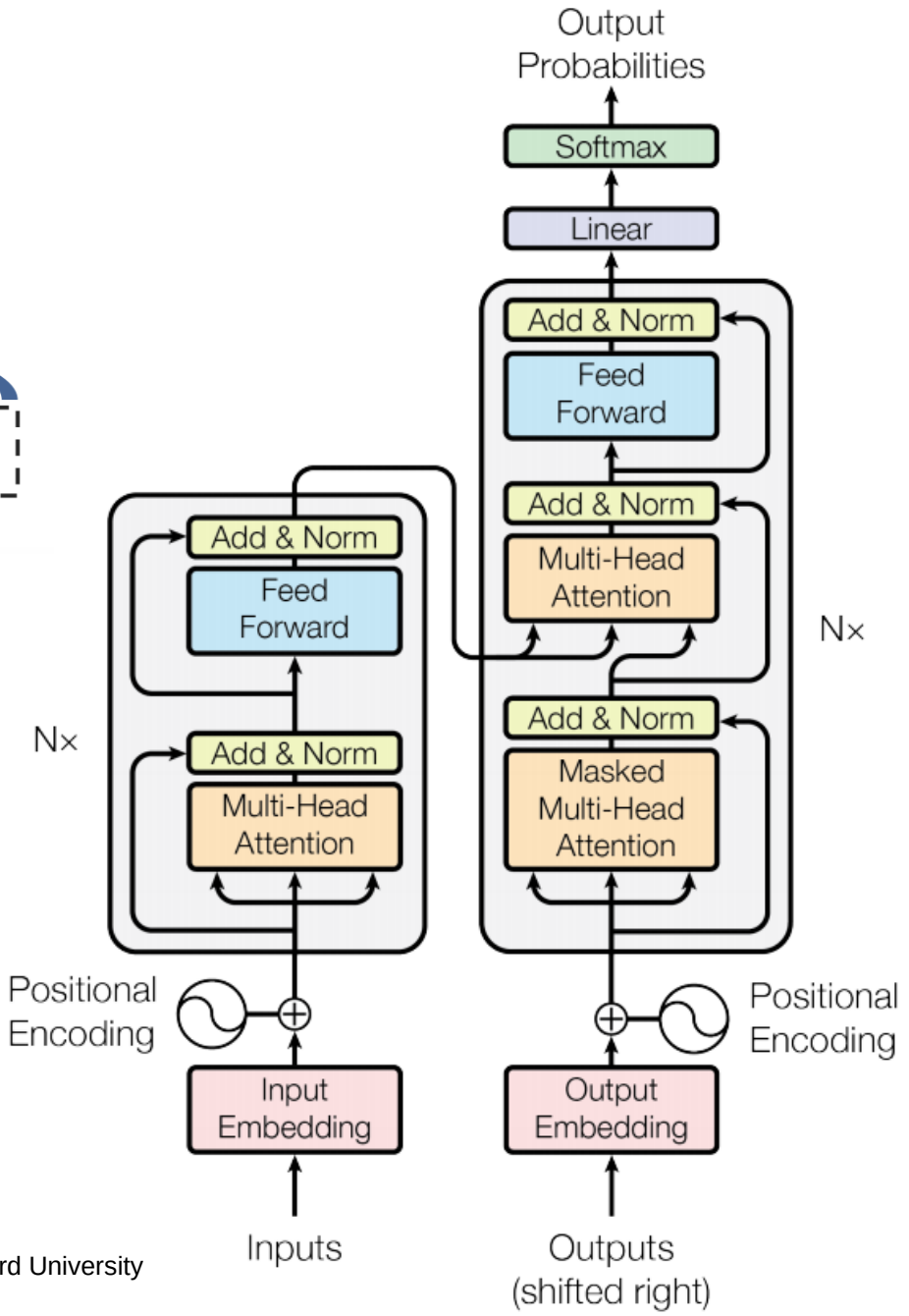
Vaswani et al, 2017. Attention is all you need.

Encoder-Decoder Attention

Encoder Self-Attention



MaskedDecoder Self-Attention



Lukasz Kaiser. 2017. Tensor2Tensor Transformers: New Deep Models for NLP. Lecture in Stanford University

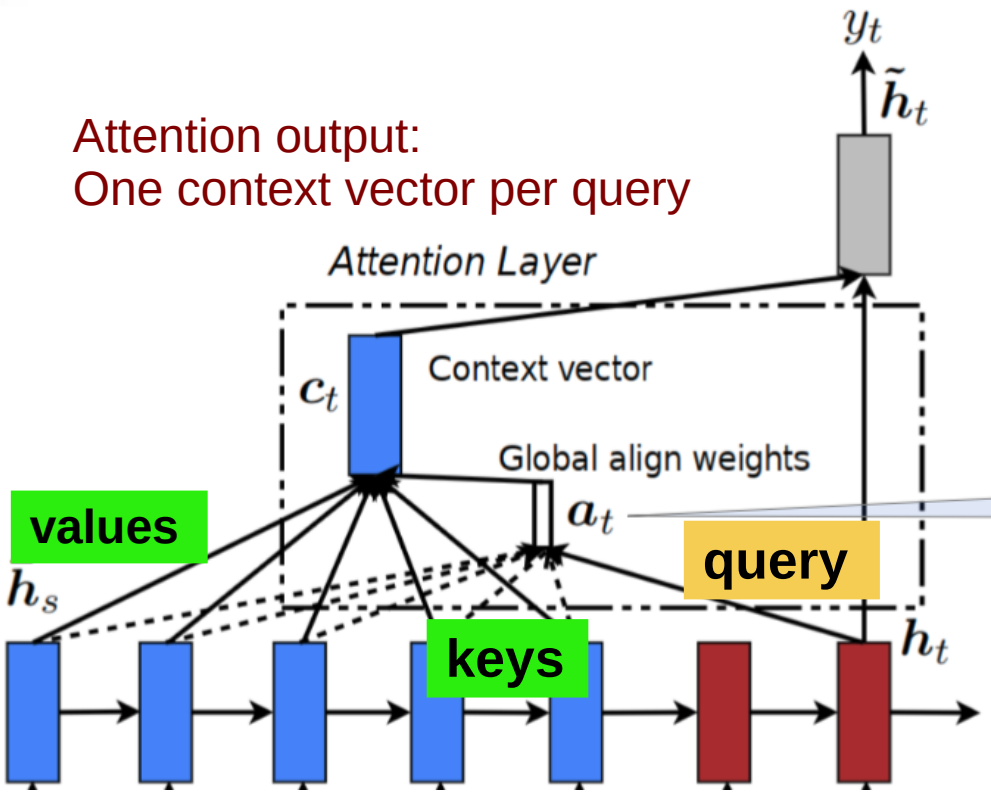
Vaswani et al, 2017. Attention is all you need.

Attention score functions

Bahdanau et al. used FFNN to calculate the attention scores (a.k.a “additive attention”):

$\text{score}(h_t, \bar{h}_s) = w_2^T \tanh(W_1[h_t; \bar{h}_s])$

Attention output:
One context vector per query



$$\text{score}(h_t, \bar{h}_s) = \begin{cases} h_t^\top \bar{h}_s & \textbf{Dot-prod.} \\ h_t^\top W_a \bar{h}_s & \textbf{Multiplicative} \\ W_a[h_t; \bar{h}_s] & \textbf{Additive} \end{cases}$$

$$a_t(s) = \frac{e^{score(h_t, \bar{h}_s)}}{\sum_{s'} e^{score(h_t, \bar{h}_{s'})}}$$

Luong et al. 2015. Effective Approaches to Attention-based Neural Machine Translation. In EMNLP

Scaled dot-product attention

- Comparison of attention functions showed:
  - For small query/key dim. Dot-product and Additive attention performed similarly
  - For large dim. Additive performed better
- Vaswani et al.: “We suspect that for large values of dk, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients”
  - Large d_k => large attention logits variance => large differences between them => peaky distribution and small gradients (DERIVE!)

Init FFNNs principle

If inputs have zero mean and unit variance, activations in each layer should have them also!

After random init *w~N(0,Var(w))*:

- DERIVE → $$Var\left(w^T x\right) = d\, Var\left(w\right) Var\left(x\right)$$
  - Use *w~N(0,1/d)* to save variance of the input
  - Principle used in Glorot/Xavier/He initializers
- Similarly, in the beginning of training vectors are random, queries and keys are independent:

$$Var\left(q^T k\right) = d\, Var\left(q\right) Var\left(k\right)$$

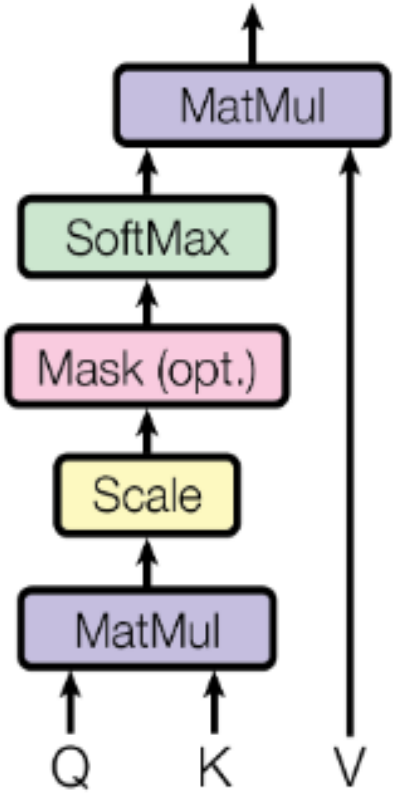
Softmax in attention weights

- As the query/key dimensionality grows, the variance of the inputs to the softmax grows,
  => grows the difference between the largest and the second largest logit,
  => grows the ratio of the largest and the second largest softmax output, i.e. softmax output approaches 1-hot distribution
  => one a_i is nearly 1, all others are nearly 0

$$\frac{\partial a_i}{\partial score_j} = \frac{\partial [softmax(score)]_i}{\partial score_j} = \begin{cases} -a_i a_j, i \neq j \\ a_i(1-a_i), i = j \end{cases}$$

All partial derivatives become nearly 0!

Scaled dot-product attention



Fast vectorized implementation: attention of all timesteps to all timesteps simultaneously:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

- $Q, K \in \mathbb{R}^{n \times d_k}$
- $V \in \mathbb{R}^{n \times d_v}$
- $out \in \mathbb{R}^{\text{n} \times d_v}$

For Queries of length m and keys/values of length n: length of the output?

Vaswani et al, 2017. Attention is all you need.

Masked self attention

During training, when processing each timestep, **decoder** shouldn't see future timesteps (they will not be available at test time)

- Set to attention scores (inputs to softmax), corresponding to illegal attention to future steps, to large negative values (-1e9)

  => attention weights are zero

```
>>> m = np.triu(np.ones((5,5)), k=1)
>>> print(m.astype(int))
[[0 1 1 1 1]
 [0 0 1 1 1]
 [0 0 0 1 1]
 [0 0 0 0 1]
 [0 0 0 0 0]]
```

```
def subsequent_mask(size):
    "Mask out subsequent positions."
    attn_shape = (1, size, size)
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
    return torch.from_numpy(subsequent_mask) == 0
```

np.triu: zero elements bolow k-th diagonal

Rush. The Annotated Transformer. https://nlp.seas.harvard.edu/2018/04/03/attention.html

(Masked) scaled dot-product impl.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

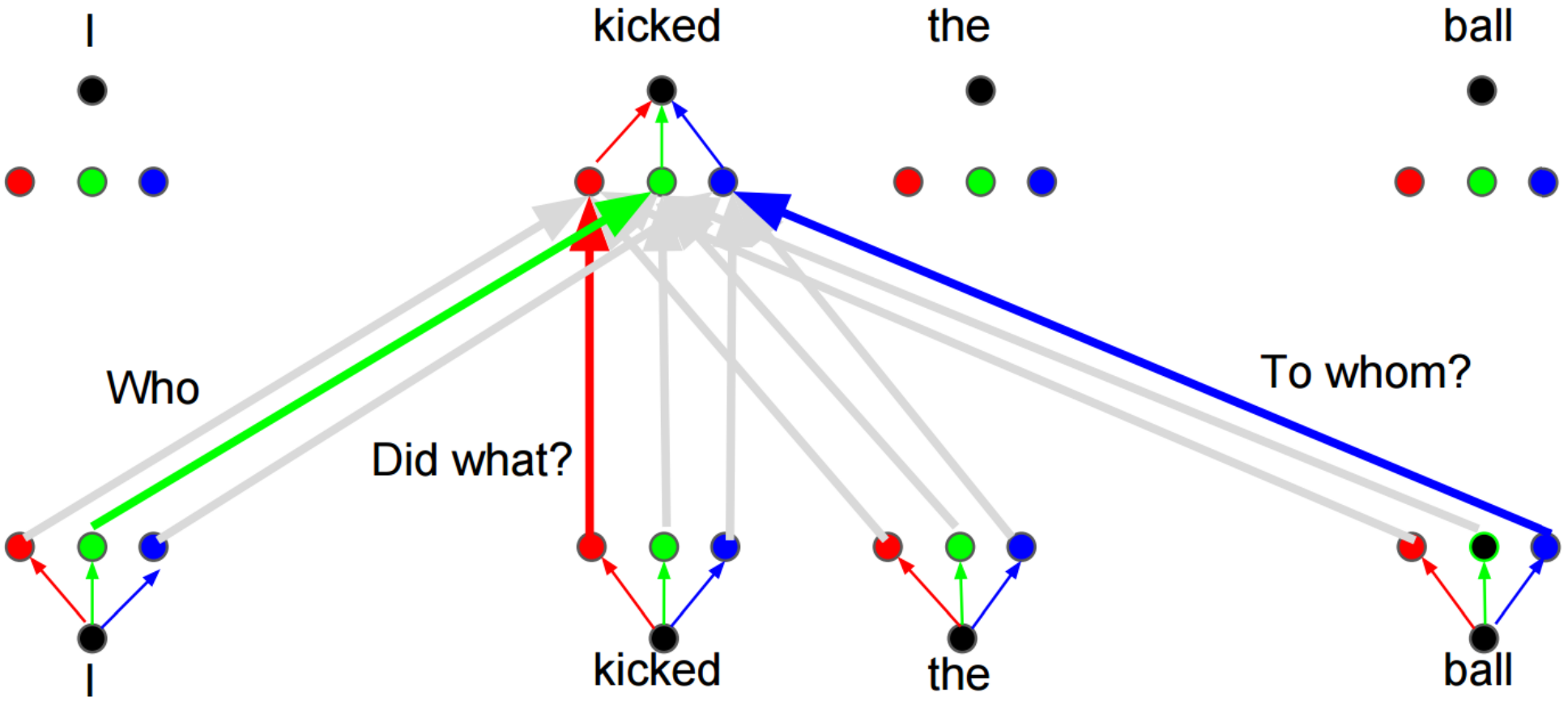
```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    key_t = key.transpose(-2, -1)
    scores = torch.matmul(query, key_t) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```


Rush. The Annotated Transformer. https://nlp.seas.harvard.edu/2018/04/03/attention.html

Multihead attention

- Single-head attention can attend to several words at once
  - But their representations are averaged (with weights)
  - What if we want to keep them separate?
    - Singular subject + plural object: can we restore number for each after averaging?
- Multi-head attention: make several parallel attention layers (attention heads).
  - How heads can the differ if there is no weights there?
    - Different Q,K,V
  - How Q,K,V can differ if they come from the same place?
    - Apply different linear transformations to them!
- Vaswani et al.: *“Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.”*

Multihead attention



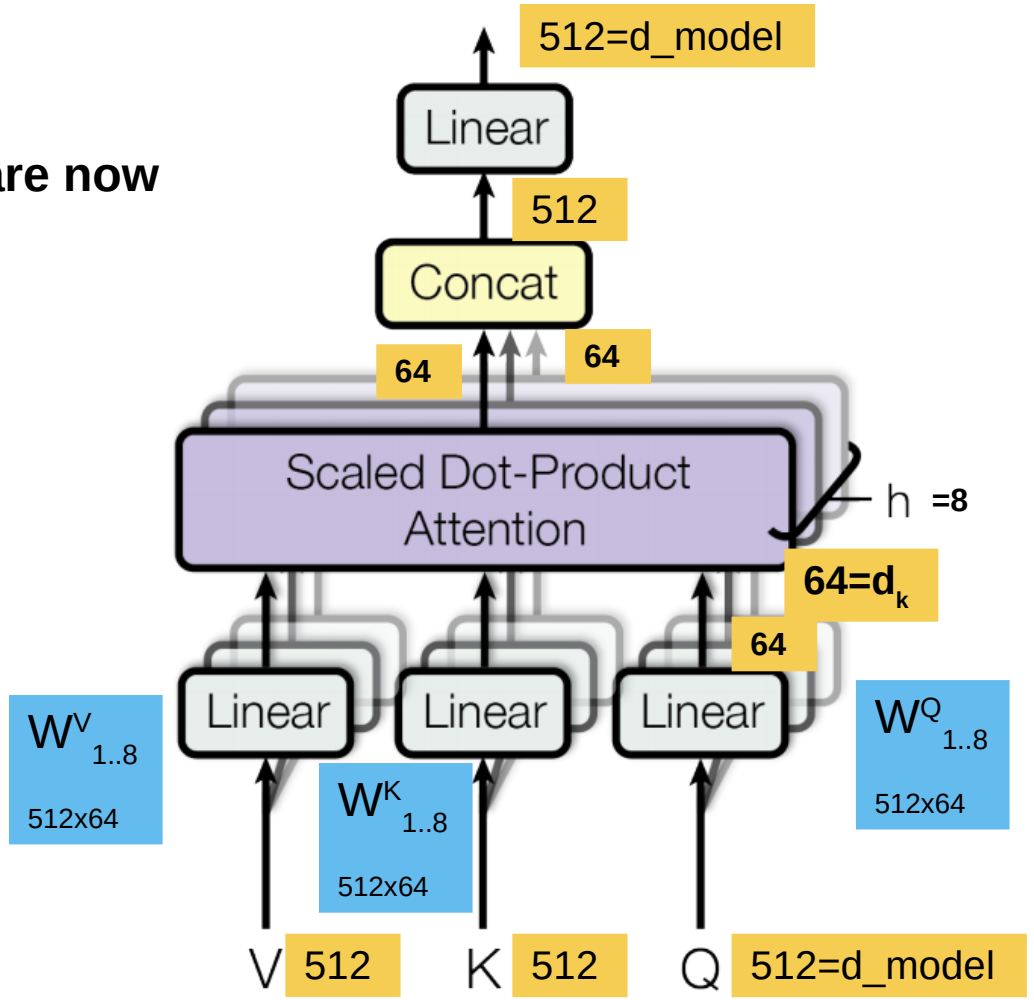
Ashish Vaswani and Anna Huang. Self-Attention For Generative Models

Multi-head attention

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Keys and Values are now different!



$d_k = d_v = d_{model} / h$

Vaswani et al, 2017. Attention is all you need.

Multihead attention impl

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        "Implements Figure 2"
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

        # 1) Do all the linear projections in batch from d_model => h x
        query, key, value = \
            [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (query, key, value))]

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous() \
             .view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

Suppose query is (bs, m, 512), key and value are (bs, n, 512)

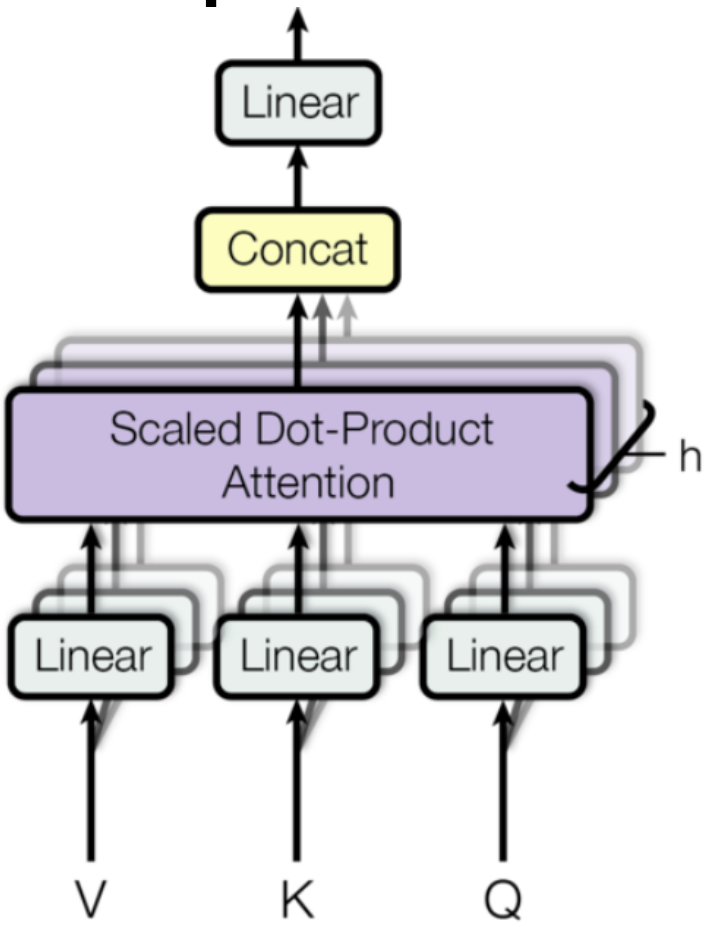
View as (bs, seqlen, 8, 64)→(bs, 8, seqlen, 64)

→(bs, 8, m, 64), (bs, 8, m, n)

→ (bs, m, 8, 64)

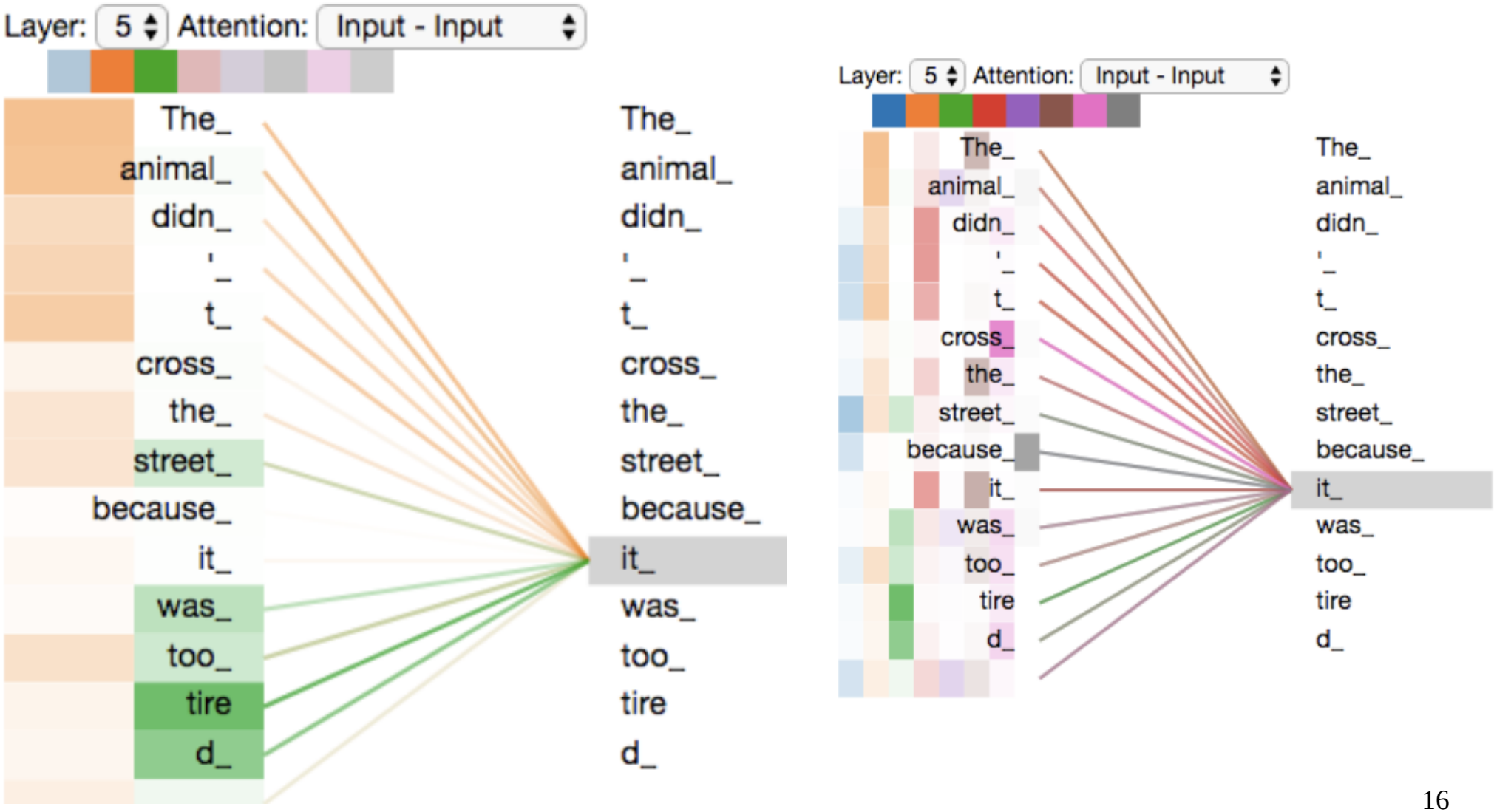
Concat: View as (bs, m, 512)

Result is (bs, m, 512)



Rush. The Annotated Transformer. https://nlp.seas.harvard.edu/2018/04/03/attention.html

Multihead self-attention in encoder



Jay Alammar. The Illustrated Transformer. http://jalammar.github.io/illustrated-transformer/

Complexity

- Self-attention layer is cheaper than convolutional or recurrent when d>>n (for sentence to sentence MT: n~70, d~1000)
- Multihead self-attention: **$O(n^2d+nd^2)$** ops, FFNNs add **$O(nd^2)$**
  - But parallel across positions (unlike RNNs), and isn't multiplied by kernel size (unlike CNNs)
- Relate each 2 positions by constant number of operations
  - good gradients to learn long-range dependencies

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

n: sequence length, k: kernel size, d: hidden size

Vaswani et al, 2017. Attention is all you need.

Memory Complexity

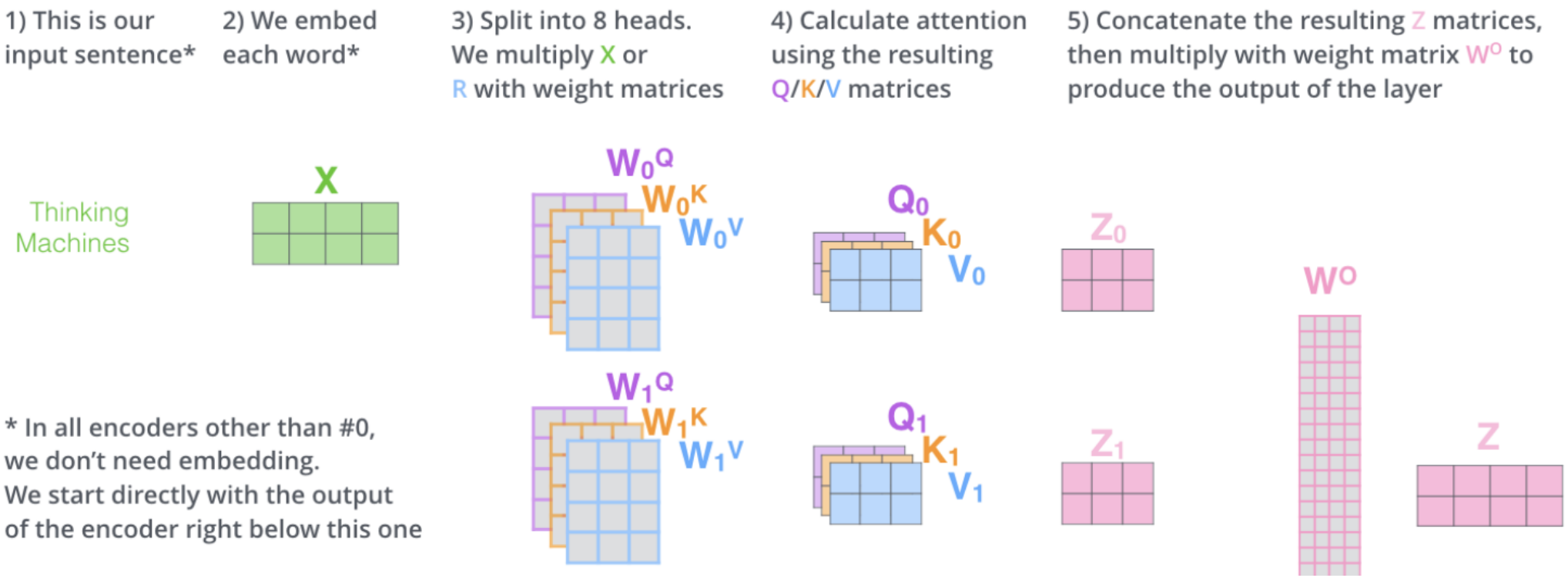
- The naive implementation of attention requires $O(n^2)$ memory to store the attention matrices.
  - Limited GPU memory (6GB-32GB) becomes a problem for larger texts (>300-500 words).
  - Can reduce the batch size and use gradient accumulation:
    loss = sum loss(ex) / batch_size

```
grads := 0 # init gradients w.r.t. all params. With 0
for ex in batch:
    grad += d loss(ex) / batch_size
```

- Efficient implementations exist that O(sqrt(n)) additional memory (+ O(n) for the outputs)
  - See Rabe and Staats. Self-attention does not require $O(n^2)$ memory, 2021
- Many modifications of Transformer were proposed for long sequences.
  - Big Bird, Longformer, etc.

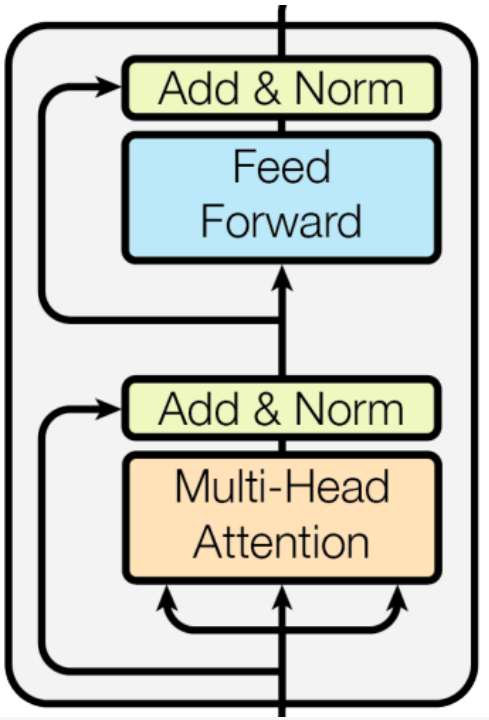
Multi-head attention

- Q,K,V “All the lonely people. Where do they all come from?”
  - Strikingly, they all are equal to the previous layer output: Q=K=V=X



Jay Alammar. The Illustrated Transformer. http://jalammar.github.io/illustrated-transformer/

Transformer layer (enc)




```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```


Rush. The Annotated Transformer. https://nlp.seas.harvard.edu/2018/04/03/attention.html

Positionwise FFNN

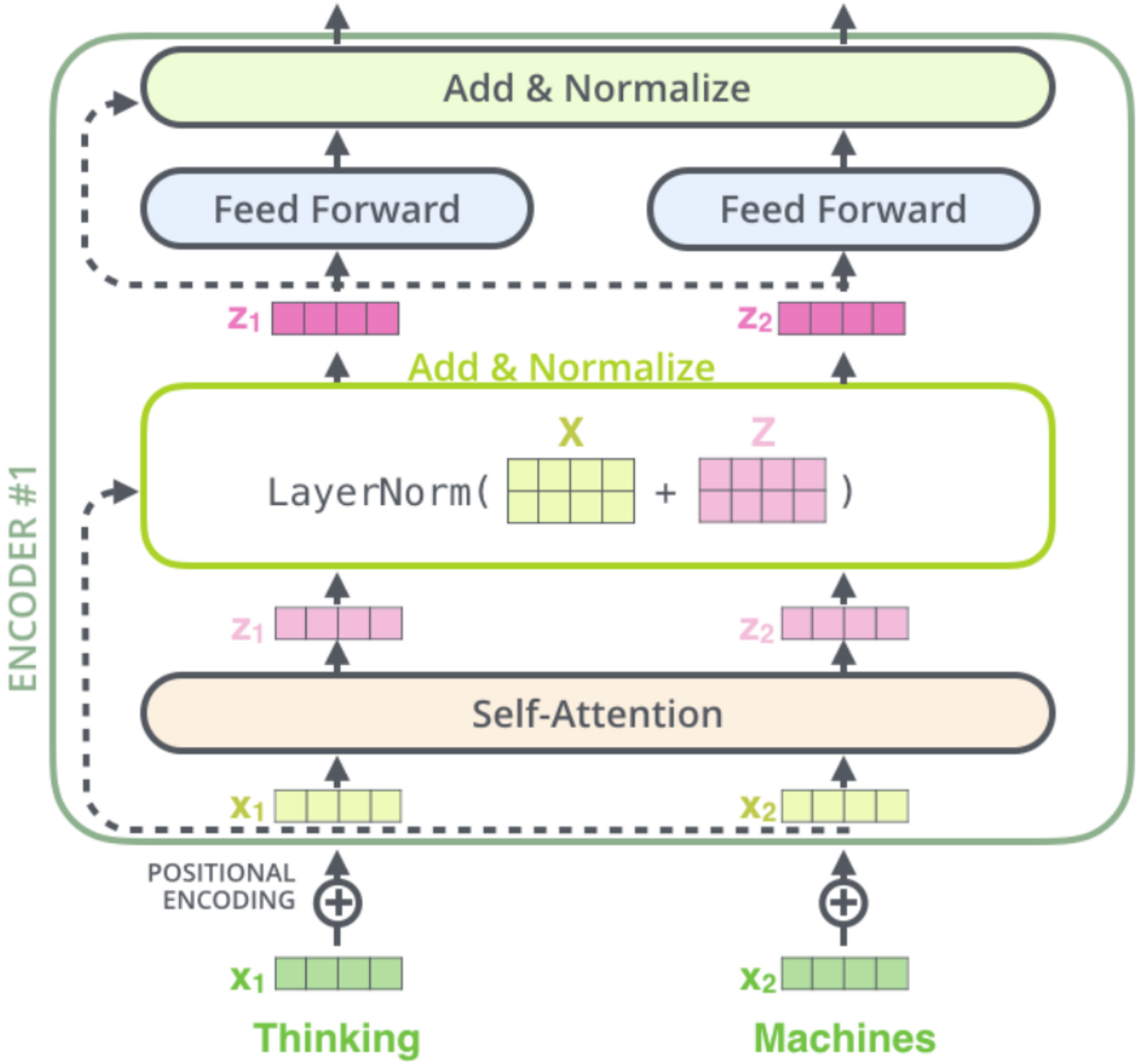
- Linear → ReLU → Linear $\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$
  - Base: 512 → 2048 → 512
  - Large: 1024 → 4096 → 1024
- Equal to 2 conv layers with kernel size 1

```
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```


Rush. The Annotated Transformer. https://nlp.seas.harvard.edu/2018/04/03/attention.html

Transformer layer (enc) unrolled



Jay Alammar. The Illustrated Transformer. http://jalammar.github.io/illustrated-transformer/

Layer normalization

$$\mathbf{h}^t = f\left[\frac{\mathbf{g}}{\sigma^t} \odot \left(\mathbf{a}^t - \mu^t\right) + \mathbf{b}\right] \qquad \mu^t = \frac{1}{H}\sum_{i=1}^{H} a_i^t \qquad \sigma^t = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^t - \mu^t\right)^2}$$

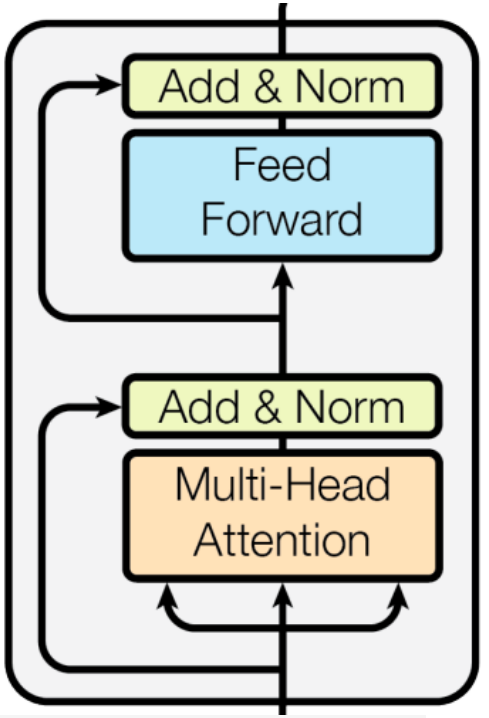
Ba, Kiros, Hinton. Layer Normalization, 2016

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```


Rush. The Annotated Transformer. https://nlp.seas.harvard.edu/2018/04/03/attention.html

Residuals



- The paper propose this order:

  LayerNorm(x + dropout(Sublayer(x)))

- And Rush use another order:

```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```


Rush. The Annotated Transformer. https://nlp.seas.harvard.edu/2018/04/03/attention.html

Residuals original impl. (v.1)

```
# Sequences of operations to perform on layer input and layer output.
# Used by common_layers.layer_preprocess, common_layers.layer_postprocess
# Each character represents an operation:
# none: no preprocessing
#    d: apply dropout
#    n: apply normalization (see norm_type and norm_epsilon)
#    a: add layer input (residual connection - only during postprocess)
# The special string "none" is used instead of the empty string
# to indicate no pre/postprocessing, since the empty string causes
# trouble for hyperparameter tuning.
# TODO(noam): The current settings ("", "dan") are the published version
# of the transformer.  ("n", "da") seems better for harder-to-learn
# models, so it should probably be the default.
layer_preprocess_sequence="none",
layer_postprocess_sequence="dan",
```


https://github.com/tensorflow/tensor2tensor/blob/v1.6.5/tensor2tensor/layers/common_hparams.py#L110-L112

Residuals original impl. (v.2)

```
def transformer_base_v2():
  """Set of hyperparameters."""
  hparams = transformer_base_v1()
  hparams.layer_preprocess_sequence = "n"
  hparams.layer_postprocess_sequence = "da"
  hparams.layer_prepostprocess_dropout = 0.1
  hparams.attention_dropout = 0.1
  hparams.relu_dropout = 0.1
  hparams.learning_rate_warmup_steps = 8000
  hparams.learning_rate = 0.2
  return hparams
```


https://github.com/tensorflow/tensor2tensor/blob/v1.6.5/tensor2tensor/layers/common_hparams.py#L110-L112

Positional encodings

- Transformer layer is permutation equivariant
  - Invariant vs equivariant
  - Encoding of each word depends on all other words, but doesn’t depend on their positions / order!

    enc(##berry | black ##berry and blue cat) =

    = enc(##berry | blue ##berry and black cat)

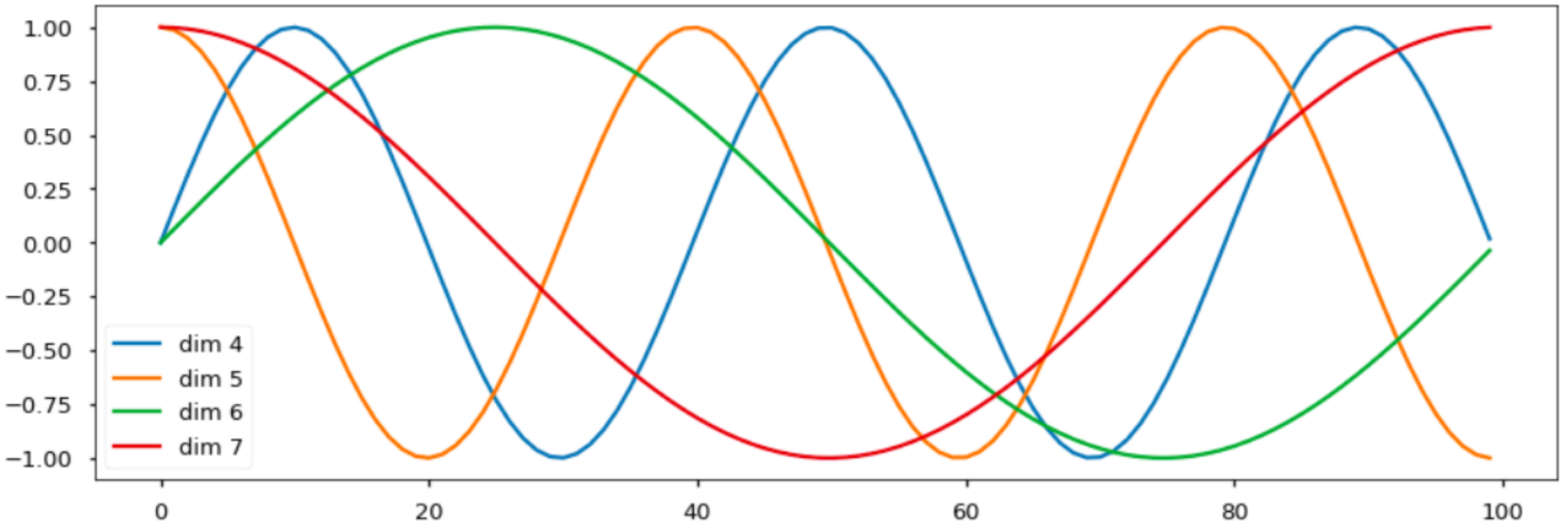
- Encode positions in inputs!

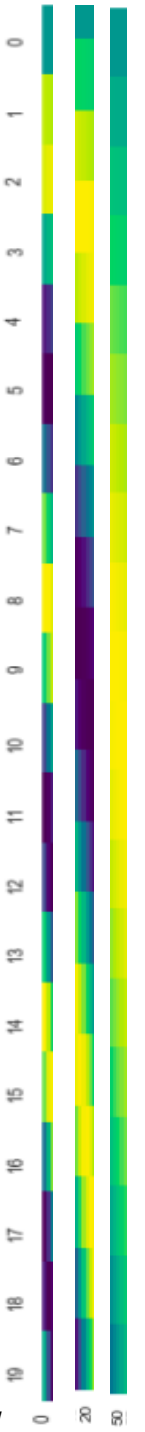
Positional encoding

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

- " we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}"





Jay Alammar. The Illustrated Transformer. http://jalammar.github.io/illustrated-transformer/
Rush. The Annotated Transformer. https://nlp.seas.harvard.edu/2018/04/03/attention.html

Positional Encoding

$$\begin{bmatrix} \Phi_1^{(k)} & 0 & \cdots & 0 \\ 0 & \Phi_2^{(k)} & \cdots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \cdots & \Phi_{\frac{d_{\text{model}}}{2}}^{(k)} \end{bmatrix} \begin{bmatrix} \sin(\lambda_1 t) \\ \cos(\lambda_1 t) \\ \sin(\lambda_2 t) \\ \cos(\lambda_2 t). \\ . \\ . \\ \sin(\lambda_{d_{model}/2} t) \\ \cos(\lambda_{d_{model}/2} t) \end{bmatrix} = PE_{(t+k)}$$

PE_t

$$\underbrace{\begin{bmatrix} \cos(\lambda_m k) & \sin(\lambda_m k) \\ -\sin(\lambda_m k) & \cos(\lambda_m k) \end{bmatrix}}_{\Phi_m^{(k)}} \begin{bmatrix} \sin(\lambda_m t) \\ \cos(\lambda_m t) \end{bmatrix} = \begin{bmatrix} \sin(\lambda_m (t+k)) \\ \cos(\lambda_m (t+k)) \end{bmatrix}$$

Images from https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

Alternative: position embeddings

Used in BERT:

- Fix maximum possible sequence length (512 tokens=subwords)
- Lookup table 512xd_{model} with a separate trainable embedding for each possible position
- Cannot generalize to sequences of length >512

Other alternatives:

- Relative position embeddings

Positional encoding

- Alternative – Positional embeddings:

  trainable embedding for each position

  - Same results, but limits input length for inference
  - “We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.”

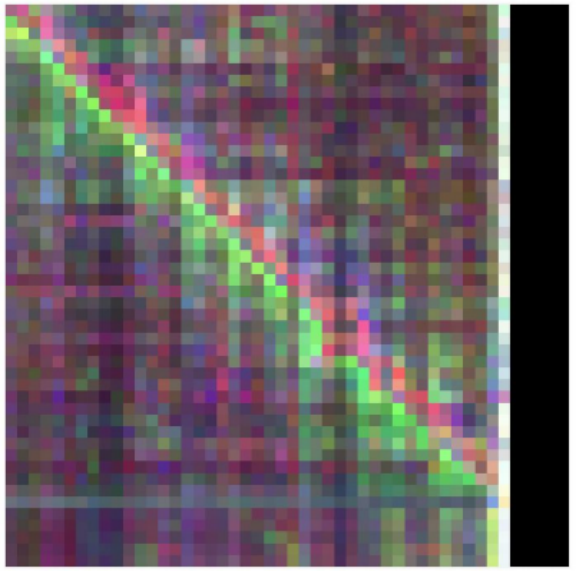
- BERT use Transformer with positional embeddings

  => input length <=512 subtokens

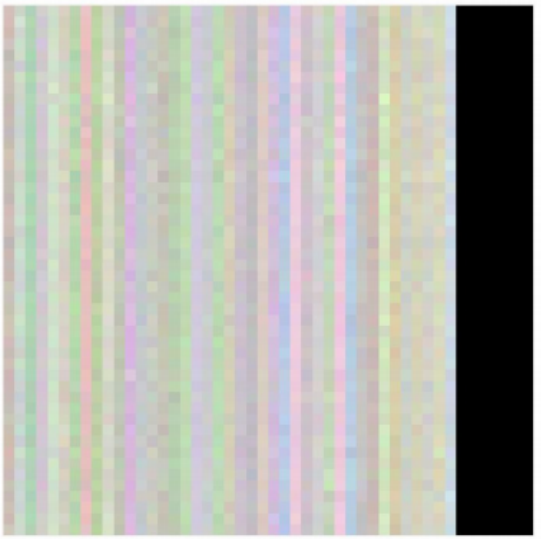
	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8
(E)	positional embedding instead of sinusoids									4.92	25.7

Vaswani et al, 2017. Attention is all you need.

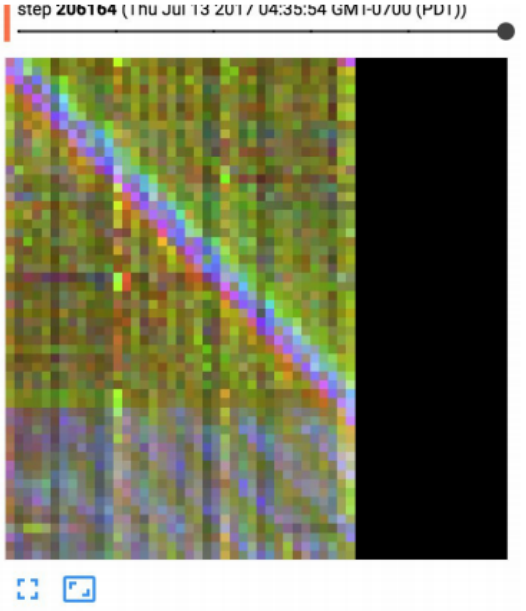
Residuals carry positional information to higher layers, among other information.



With residuals



Without residuals



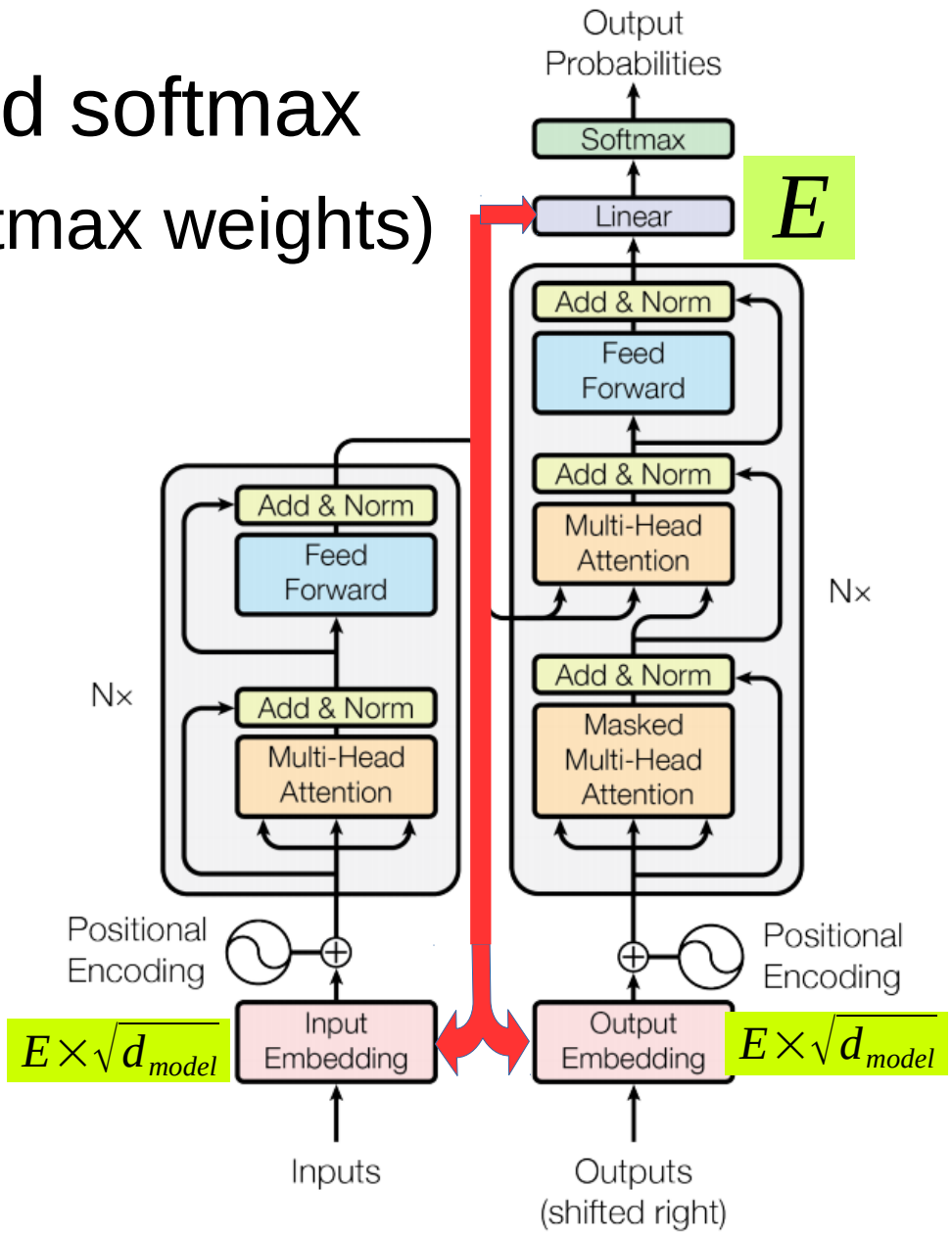
Without residuals, with timing signals

Embeddings

- Shared embeddings = tied softmax
  - Dec output embs (pre-softmax weights)
  - Dec input embs
  - Enc input embs

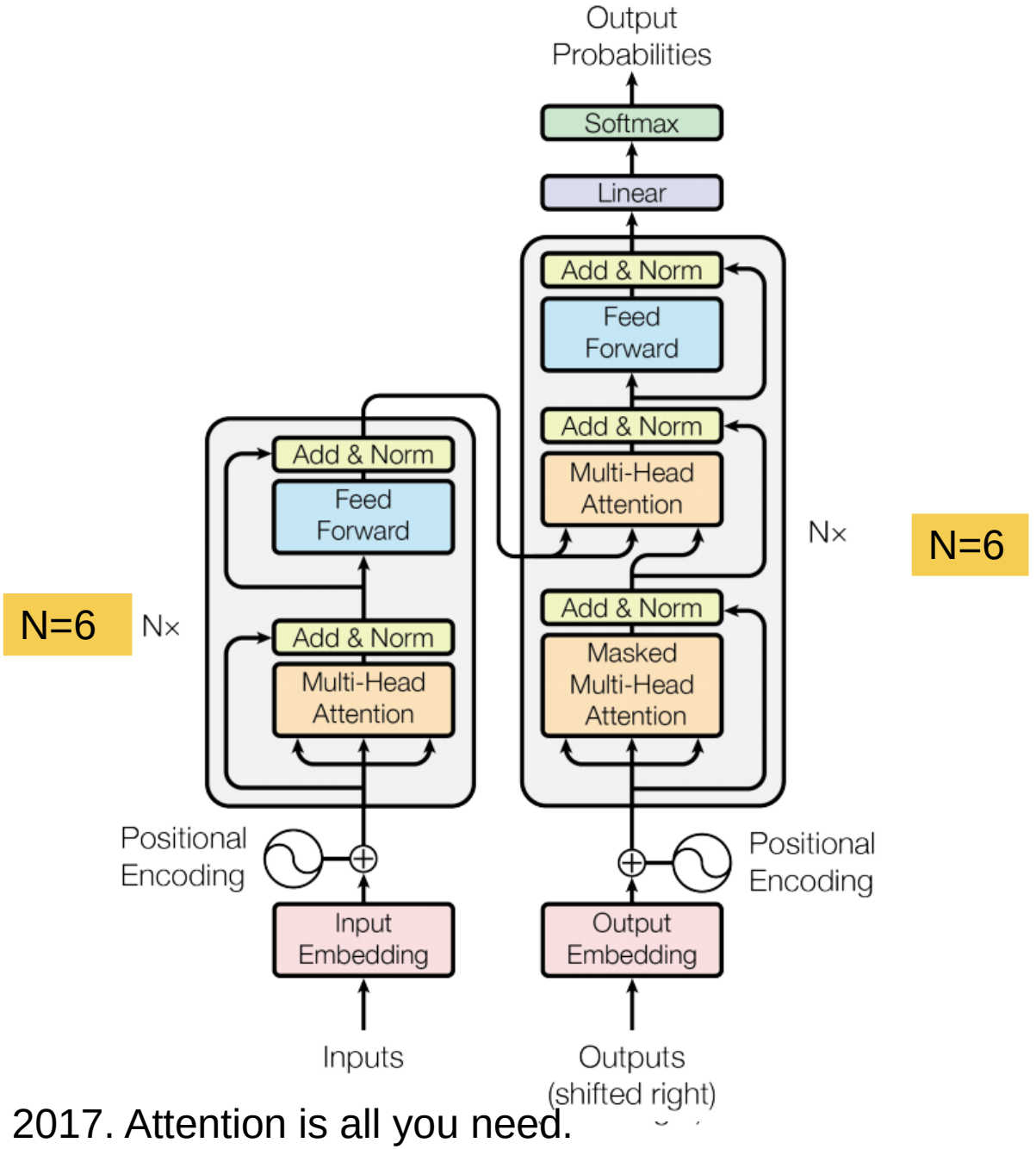
    => src-tgt vocab sharing!
- For larger dataset (en → fr)

  enc input embs are different



Vaswani et al, 2017. Attention is all you need.

The whole model



Vaswani et al, 2017. Attention is all you need.

Regularization

- Residual dropout
  - “… apply dropout to the output of each sub-layer, before it is added to the sub-layer input… ”
- Input dropout
  - “… apply dropout to the sums of the embeddings and the positional encodings… ”
- ReLU dropout
  - In FFNN, to the output of the hidden layer (after ReLU)

Regularization

- Residual dropout, ReLU dropout, Input dropout
- Attention dropout (only for some experiments)
  - Dropout on attention weights (after softmax)
- Label smoothing $CE(oh(y), \hat{y}) \rightarrow CE((1-\epsilon)oh(y) + \epsilon/K, \hat{y})$

$$H(q', p) = -\sum_{k=1}^{K} \log p(k) q'(k) = (1-\epsilon) H(q, p) + \epsilon H(u, p)$$

$\epsilon = 0.1$

Thus, LSR is equivalent to replacing a single cross-entropy loss $H(q, p)$ with a pair of such losses $H(q, p)$ and $H(u, p)$.

  - *H(q,p)* pulls predicted distribution towards *oh(y)*
  - *H(u,p)* – towards prior (uniform) distribution
  - "This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score."

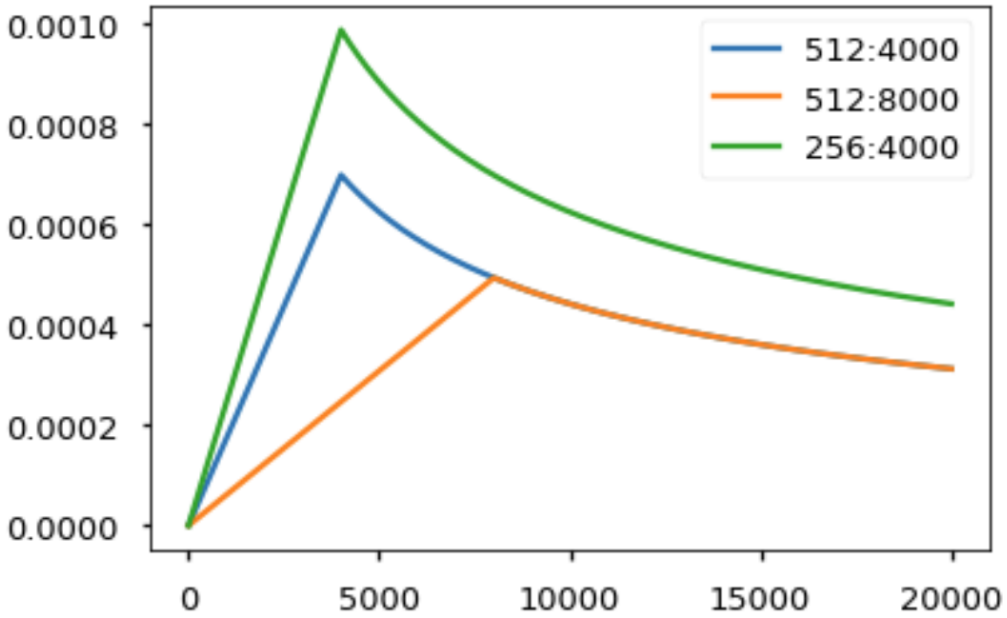
Label smoothing from:
Szegedy. Rethinking the Inception Architecture for Computer Vision, 2015

Training

- Adam, betas=0.9,0.98, eps=1e-9
- Learning rate: linear warmup: 4K-8K steps (3-10% is common) + square root decay

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

Noam Optimizer: Adam+this lr schedule



Rush. The Annotated Transformer. https://nlp.seas.harvard.edu/2018/04/03/attention.html

Base model: v1 vs v2

- Transformer base already has 3 versions of hyperparameters in codebase!
  - Main differences in dropouts and lr, lr schedule

```
def basic_params1():
    layer_prepostprocess_dropout=0.1
```

```
def transformer_base_v1():
  hparams.add_hparam("attention_dropout", 0.0)
  hparams.add_hparam("relu_dropout", 0.0)
  hparams.learning_rate = 0.1
  hparams.learning_rate_warmup_steps = 4000
```

```
def transformer_base_v2():
  """Set of hyperparameters."""
  hparams = transformer_base_v1()
  hparams.layer_preprocess_sequence = "n"
  hparams.layer_postprocess_sequence = "da"
  hparams.layer_prepostprocess_dropout = 0.1
  hparams.attention_dropout = 0.1
  hparams.relu_dropout = 0.1
  hparams.learning_rate_warmup_steps = 8000
  hparams.learning_rate = 0.2
  return hparams
```


https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/models/transformer.py

Hypers for parsing

- Seems like initially they used attention dropout only for parsing experiments, but later enabled them for MT
- Probably this brought them SOTA on En→Fr
  - 41.0(Jun'17)→41.8 (Dec'17)
  - vs. 41.29 (ConvS2S Ensemble)

```
def transformer_parsing_base():
  """HParams for parsing on WSJ only."""
  hparams = transformer_base()
  hparams.attention_dropout = 0.2
  hparams.layer_prepostprocess_dropout = 0.2
  hparams.max_length = 512 z
  hparams.learning_rate_warmup_steps = 16000
  hparams.hidden_size = 1024
  hparams.learning_rate = 0.05
  hparams.shared_embedding_and_softmax_weights = False
  return hparams
```


https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/models/transformer.py

Training

- WMT2014 En→De / Fr: 4.5M / 36M sent.pairs
  - word-pieces vocab: 37K shared / 32K x2 separate
  - Batches: sequences of approx. same length, dynamic batch size: 25K src & 25K tgt tokens
  - On 8 P100 GPU (16GB), base/big: 0.5/3.5 days, 100k/300k steps 0.4/1.0s per step
  - Average weights from last 5/20 checkpoints
  - Beam search with size 4, length penalty 0.6

Dev: newstest2013 en→de

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

Vaswani et al, 2017. Attention is all you need.

Results

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	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 [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	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	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Vaswani et al, 2017. Attention is all you need.