# Deep Learning & Engineering Applications

## 10. Attention Mechanism & Transformer

JIDONG J. YANG

COLLEGE OF ENGINEERING

UNIVERSITY OF GEORGIA

# Pay attention in class.

Human attention is a limited, valuable, and scarce resource.

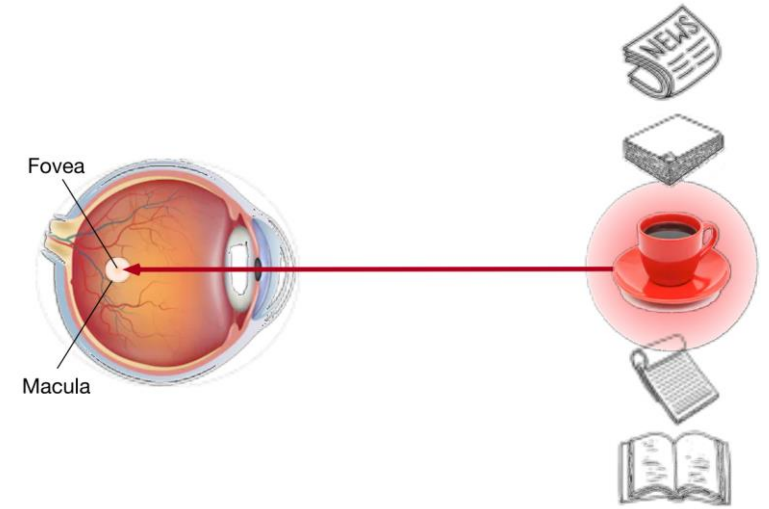
A two-component framework:

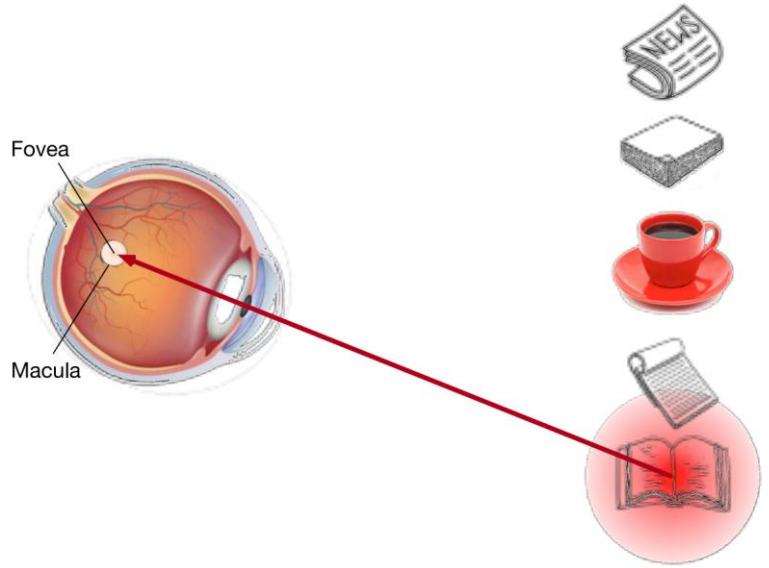
- subjects selectively direct the spotlight of attention using both the **nonvolitional cue** and **volitional cue.**
- The nonvolitional cue is based on the saliency and conspicuity of objects in the environment.
- The volitional cue is task-dependent and under cognitive and volitional control.

# Attention Cues

Using the nonvolitional cue based on saliency (red cup, non-paper), attention is involuntarily directed to the coffee.

Using the volitional cue (want to read a book) that is task-dependent, attention is directed to the book under volitional control.



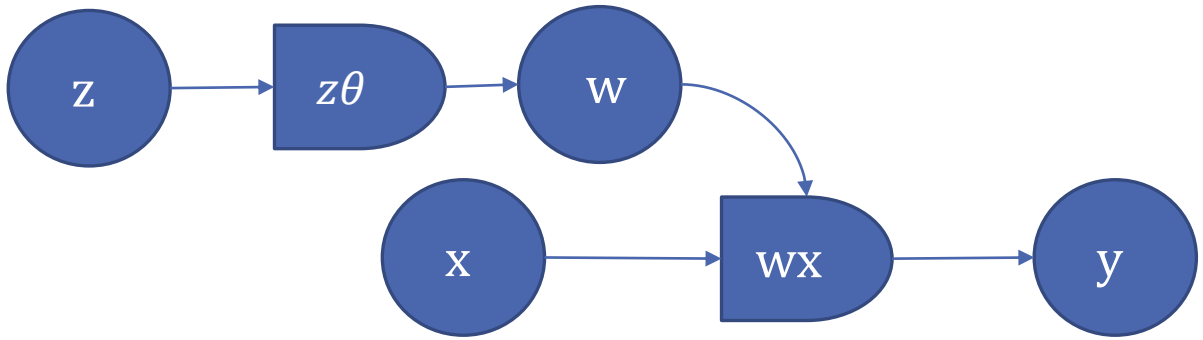
# Multiplicative Modules

**Quadratic layer, product units**

$$y_i = f_w(x) = \sum_j w_{ij} x_j$$

$$w_{ij} = g_\theta(z) = \sum_k \theta_{ijk} z_k$$

$$w_{ij} = f \circ g = \sum_{jk} \theta_{ijk} z_k x_j$$




**Attention module**

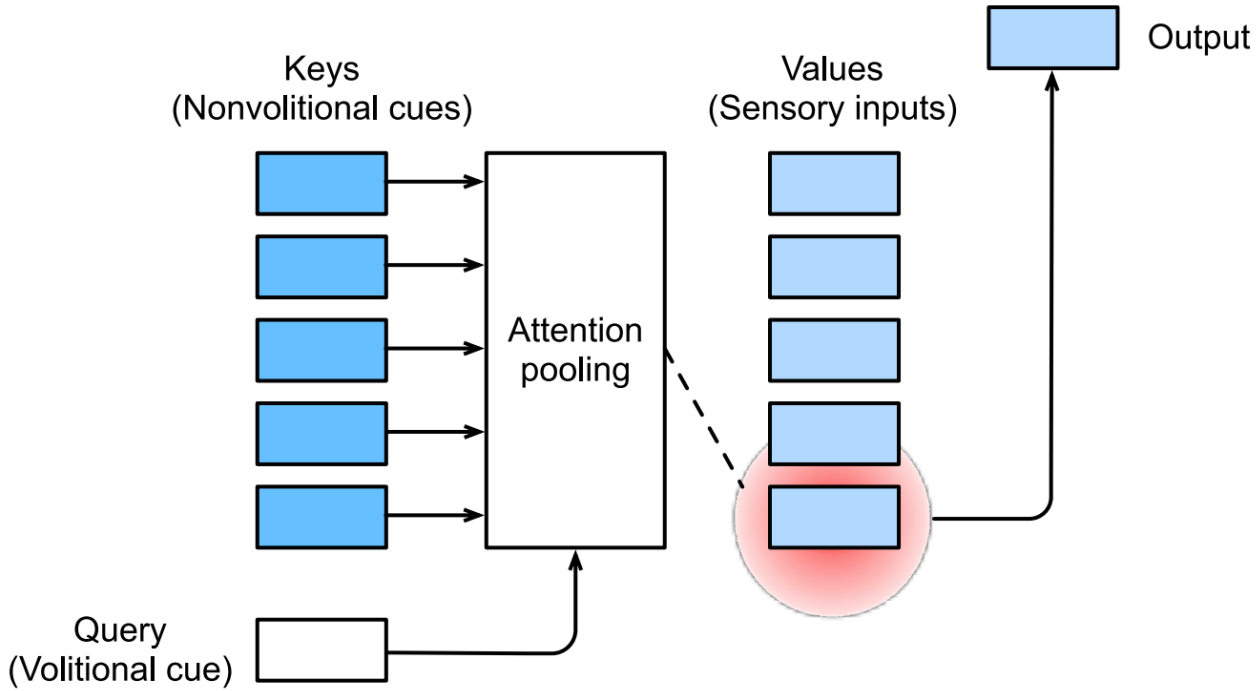
$$y_i = f_w(x) = \sum_j w_{ij} x_j \qquad w_{ij} = g(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

# Queries, Keys, and Values

Attention mechanisms bias selection over values (sensory inputs) via attention pooling, which incorporates queries (volitional cues) and keys (nonvolitional cues).

Design of attention mechanisms

- Nondifferentiable
- Differentiable



# Attention Pooling

$$y_i = 2\sin(x_i) + x_i^{0.8} + \epsilon$$

$$\{(x_1, y_1), \ldots, (x_n, y_n)\}$$

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} y_i$$

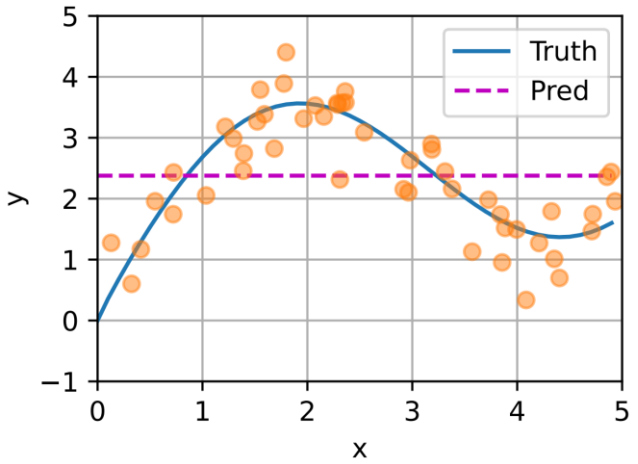
$$f(x) = \sum_{i=1}^{n} \frac{K(x - x_i)}{\sum_{j=1}^{n} K(x - x_j)} y_i = \sum_{i=1}^{n} \alpha(x, x_i) y_i$$

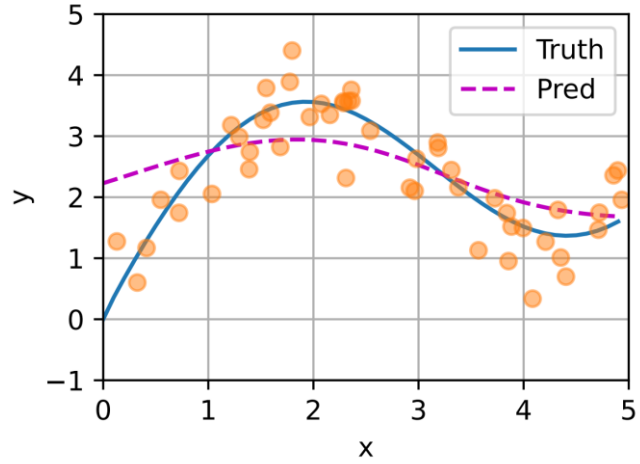
$$f(x) = \sum_{i=1}^{n} \alpha(x, x_i) y_i$$

$$= \sum_{i=1}^{n} \frac{\exp\left(-\frac{1}{2}(x - x_i)^2\right)}{\sum_{j=1}^{n} \exp\left(-\frac{1}{2}(x - x_j)^2\right)} y_i$$

$$= \sum_{i=1}^{n} \mathrm{softmax}\left(-\frac{1}{2}(x - x_i)^2\right) y_i$$

query keys values



# Attention Pooling

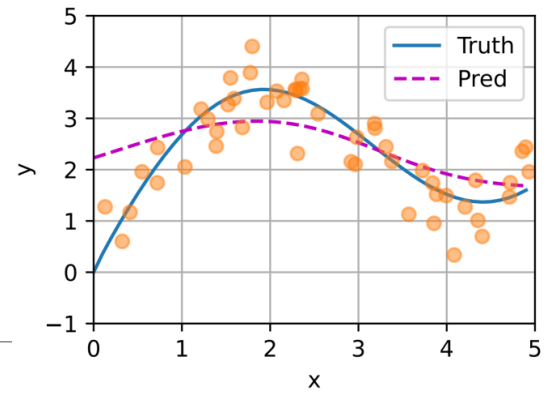
Nonparametric Attention Pooling

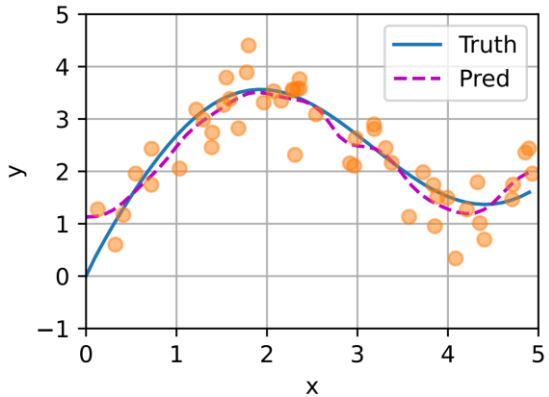
$$
\begin{aligned}
f(x) &= \sum_{i=1}^{n} \alpha(x, x_i) y_i \\
&= \sum_{i=1}^{n} \frac{\exp\left(-\frac{1}{2}(x - x_i)^2\right)}{\sum_{j=1}^{n} \exp\left(-\frac{1}{2}(x - x_j)^2\right)} y_i \\
&= \sum_{i=1}^{n} \mathrm{softmax}\left(-\frac{1}{2}(x - x_i)^2\right) y_i
\end{aligned}
$$

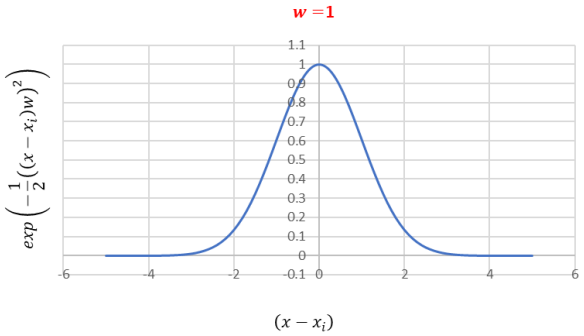
Parametric Attention Pooling (**parameter: w**)

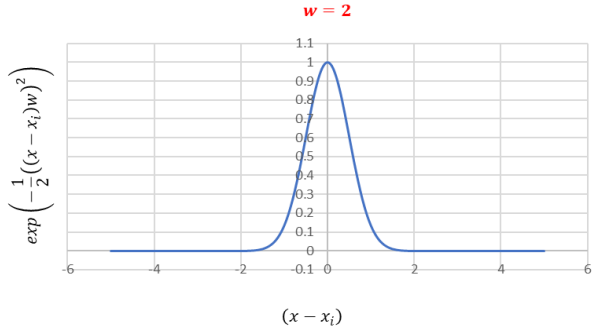
$$
\begin{aligned}
f(x) &= \sum_{i=1}^{n} \alpha(x, x_i) y_i \\
&= \sum_{i=1}^{n} \frac{\exp\left(-\frac{1}{2}((x - x_i)w)^2\right)}{\sum_{j=1}^{n} \exp\left(-\frac{1}{2}((x - x_j)w)^2\right)} y_i \\
&= \sum_{i=1}^{n} \mathrm{softmax}\left(-\frac{1}{2}((x - x_i)w)^2\right) y_i
\end{aligned}
$$

**"w" can be learned with gradient descent.**



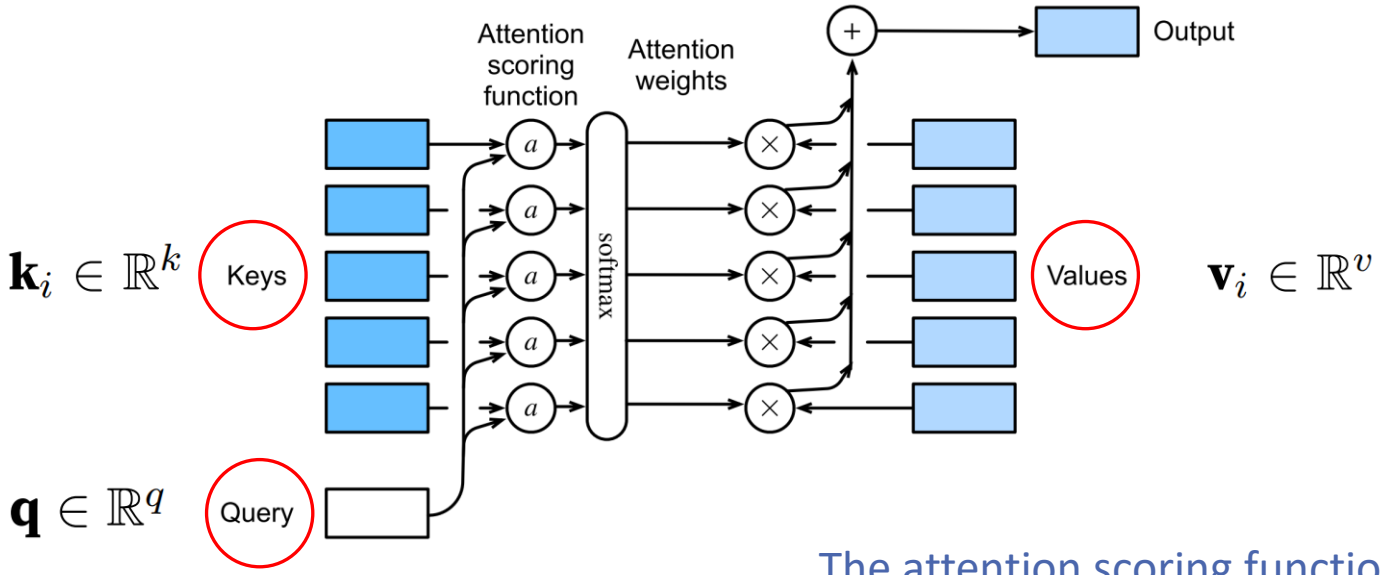




```
class NWKernelRegression(nn.Module):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.w = nn.Parameter(torch.rand((1,), requires_grad=True))

    def forward(self, queries, keys, values):
        # Shape of the output `queries` and `attention_weights`:
        # (no. of queries, no. of key-value pairs)
        queries = queries.repeat_interleave(keys.shape[1]).reshape(
            (-1, keys.shape[1]))
        self.attention_weights = nn.functional.softmax(
            -((queries - keys) * self.w)**2 / 2, dim=1)
        # Shape of `values`: (no. of queries, no. of key-value pairs)
        return torch.bmm(self.attention_weights.unsqueeze(1),
                         values.unsqueeze(-1)).reshape(-1)
```

# Attention Scoring Functions




Compute the output of attention pooling as a weighted average of values:

$$f(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)) = \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i \in \mathbb{R}^v$$

The attention scoring function $a(\boldsymbol{q}, \boldsymbol{k})$ that maps two vectors to a scalar:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \mathrm{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^{m} \exp(a(\mathbf{q}, \mathbf{k}_j))} \in \mathbb{R}$$

Different choices of $a(\boldsymbol{q}, \boldsymbol{k})$ lead to different behaviors of attention pooling.

# Two popular choices of attention scoring functions

## Additive Attention

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q\mathbf{q} + \mathbf{W}_k\mathbf{k}) \in \mathbb{R}$$

$$\mathbf{W}_q \in \mathbb{R}^{h\times q}, \mathbf{W}_k \in \mathbb{R}^{h\times k}, \quad \mathbf{w}_v \in \mathbb{R}^h$$

## Scaled Dot-Product Attention

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k}/\sqrt{d}$$

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n\times v}$$

$$\mathbf{Q} \in \mathbb{R}^{n\times d}, \mathbf{K} \in \mathbb{R}^{m\times d}, \mathbf{V} \in \mathbb{R}^{m\times v}$$

```python
class AdditiveAttention(nn.Module):
    """Additive attention."""
    def __init__(self, key_size, query_size, num_hiddens, dropout, **kwargs):
        super(AdditiveAttention, self).__init__(**kwargs)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=False)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=False)
        self.w_v = nn.Linear(num_hiddens, 1, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, queries, keys, values, valid_lens):
        queries, keys = self.W_q(queries), self.W_k(keys)
        # After dimension expansion, shape of `queries`: (`batch_size`, no. of
        # queries, 1, `num_hiddens`) and shape of `keys`: (`batch_size`, 1,
        # no. of key-value pairs, `num_hiddens`). Sum them up with
        # broadcasting
        features = queries.unsqueeze(2) + keys.unsqueeze(1)
        features = torch.tanh(features)
        # There is only one output of `self.w_v`, so we remove the last

        # one-dimensional entry from the shape. Shape of `scores`:
        # (`batch_size`, no. of queries, no. of key-value pairs)
        scores = self.w_v(features).squeeze(-1)
        self.attention_weights = masked_softmax(scores, valid_lens)
        # Shape of `values`: (`batch_size`, no. of key-value pairs, value
        # dimension)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

```python
class DotProductAttention(nn.Module):
    """Scaled dot product attention."""
    def __init__(self, dropout, **kwargs):
        super(DotProductAttention, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)

    # Shape of `queries`: (`batch_size`, no. of queries, `d`)
    # Shape of `keys`: (`batch_size`, no. of key-value pairs, `d`)
    # Shape of `values`: (`batch_size`, no. of key-value pairs, value
    # dimension)
    # Shape of `valid_lens`: (`batch_size`,) or (`batch_size`, no. of queries)
    def forward(self, queries, keys, values, valid_lens=None):
        d = queries.shape[-1]
        # Set `transpose_b=True` to swap the last two dimensions of `keys`
        scores = torch.bmm(queries, keys.transpose(1, 2)) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, valid_lens)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

# Masked Softmax

We can specify a valid sequence length (in number of tokens) to filter out those beyond this specified range when computing softmax.

```
def masked_softmax(X, valid_lens):
    """Perform softmax operation by masking elements on the last axis."""
    # `X`: 3D tensor, `valid_lens`: 1D or 2D tensor
    if valid_lens is None:
        return nn.functional.softmax(X, dim=-1)
    else:
        shape = X.shape
        if valid_lens.dim() == 1:
            valid_lens = torch.repeat_interleave(valid_lens, shape[1])
        else:
            valid_lens = valid_lens.reshape(-1)
        # On the last axis, replace masked elements with a very large negative
        # value, whose exponentiation outputs 0
        X = d2l.sequence_mask(X.reshape(-1, shape[-1]), valid_lens,
                              value=-1e6)
        return nn.functional.softmax(X.reshape(shape), dim=-1)
```

# Recall RNN encoder-decoder

The encoder transforms (encodes) an input sequence of variable length into a fixed-shape context variable **c**.

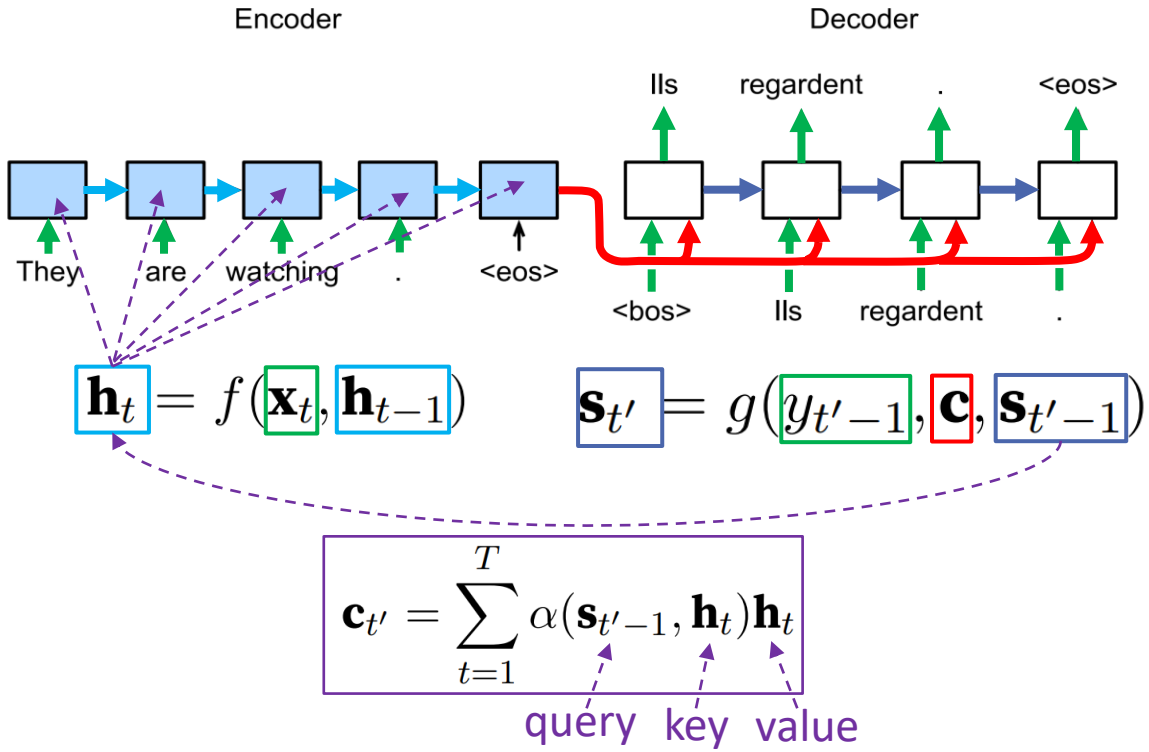
Concatenate the context variable with the decoder input at all the time steps.

Bahdanau attention

- Treats the decoder hidden state at the previous time step ($s_{t'-1}$) as the query, and the encoder hidden states at all the time steps as both the keys and values.

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^{m} \exp(a(\mathbf{q}, \mathbf{k}_j))}$$

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k})$$




$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}_{t'}, \mathbf{s}_{t'-1})$$

# Self-Attention

Given a sequence of input tokens $\mathbf{x}_1, \ldots, \mathbf{x}_n$ where any $\mathbf{x}_i \in \mathbb{R}^d$ $(1 \leq i \leq n)$

self-attention outputs a sequence of the same length $\mathbf{y}_1, \ldots, \mathbf{y}_n$, where

$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d$$

$$f(x) = \sum_{i=1}^{n} \alpha(x, x_i) x_i$$

**In self-attention, the queries, keys, and values all come from the same place.**

Recall attention pooling:

$$f(x) = \sum_{i=1}^{n} \alpha(x, x_i) y_i$$

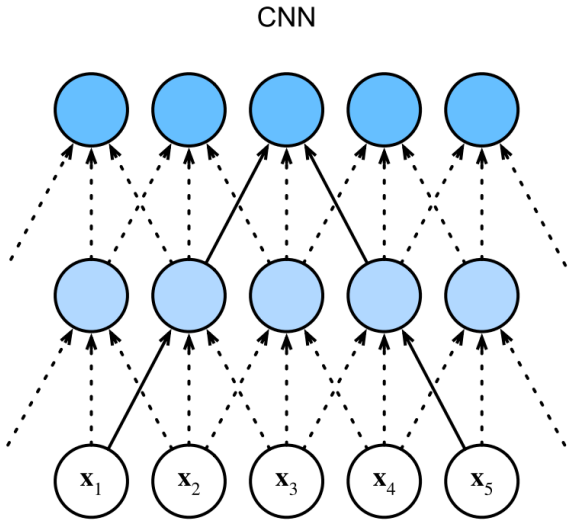
Recall attention:

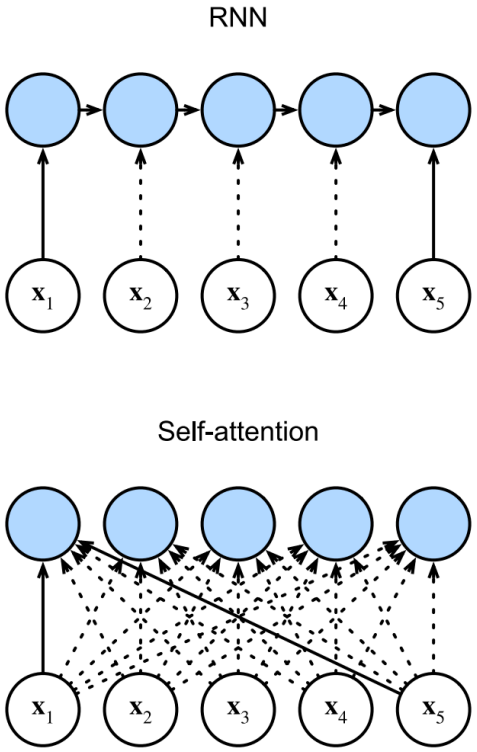
$$f(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)) = \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i \in \mathbb{R}^v$$

# Comparison of Self-attention, CNN and RNN

| | Self-attention | CNN | RNN |
|---|---|---|---|
| Computational complexity | $O(n^2 d)$ | $O(knd^2)$ | $O(nd^2)$ |
| Sequential Operations | $O(1)$ | $O(1)$ | $O(n)$ |
| Max path length | $O(1)$ | $O(log_k n)$ | $O(n)$ |

$$f(x) = \sum_{i=1}^{n} \alpha(x, x_i) x_i$$



# Issues with RNN

We need the output of the previous step to complete the next step. This prevents us from parallelizing the computation using modern GPUs.

Gradient vanishing/explosion over a long sequence of steps.

Difficulty with modeling long-term dependency.

# Position Encoding

RNNs recurrently process tokens of a sequence one by one.

Self-attention ditches sequential operations in favor of parallel computation. Therefore, the sequence order information is lost.

We can inject positional information by adding positional encoding to the input representations.

# Position Encoding

input representation $\mathbf{X} \in \mathbb{R}^{n \times d}$

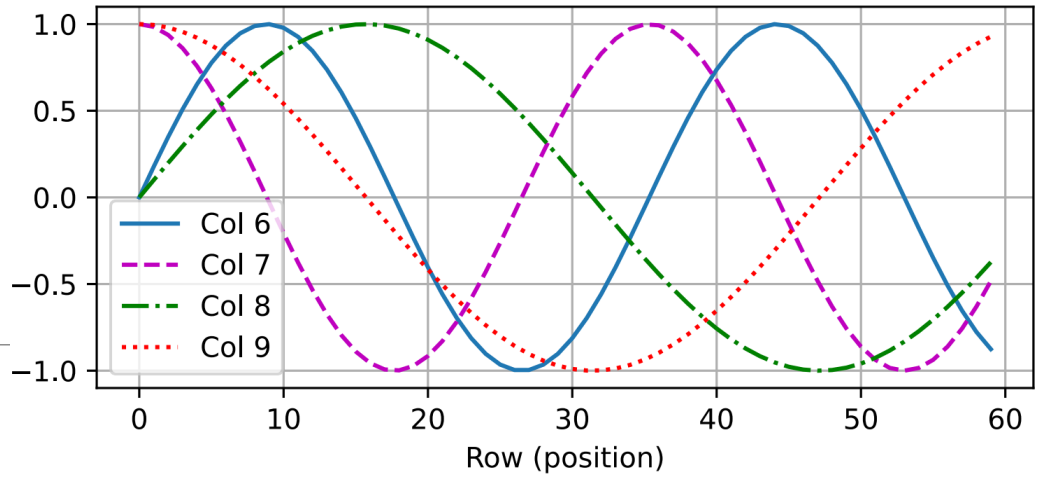
$d$ dimensional embeddings

$n$ tokens of a sequence

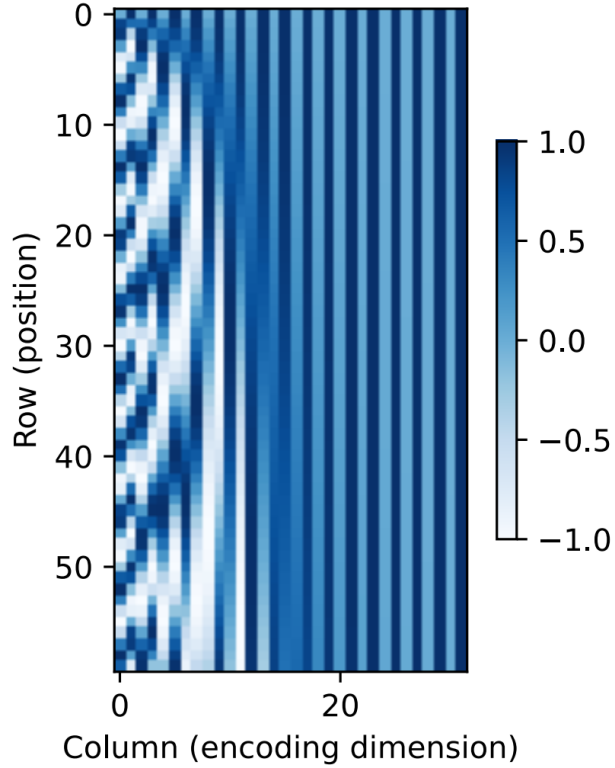
positional embedding matrix $\mathbf{P} \in \mathbb{R}^{n \times d}$

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right)$$

$$p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right)$$




```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

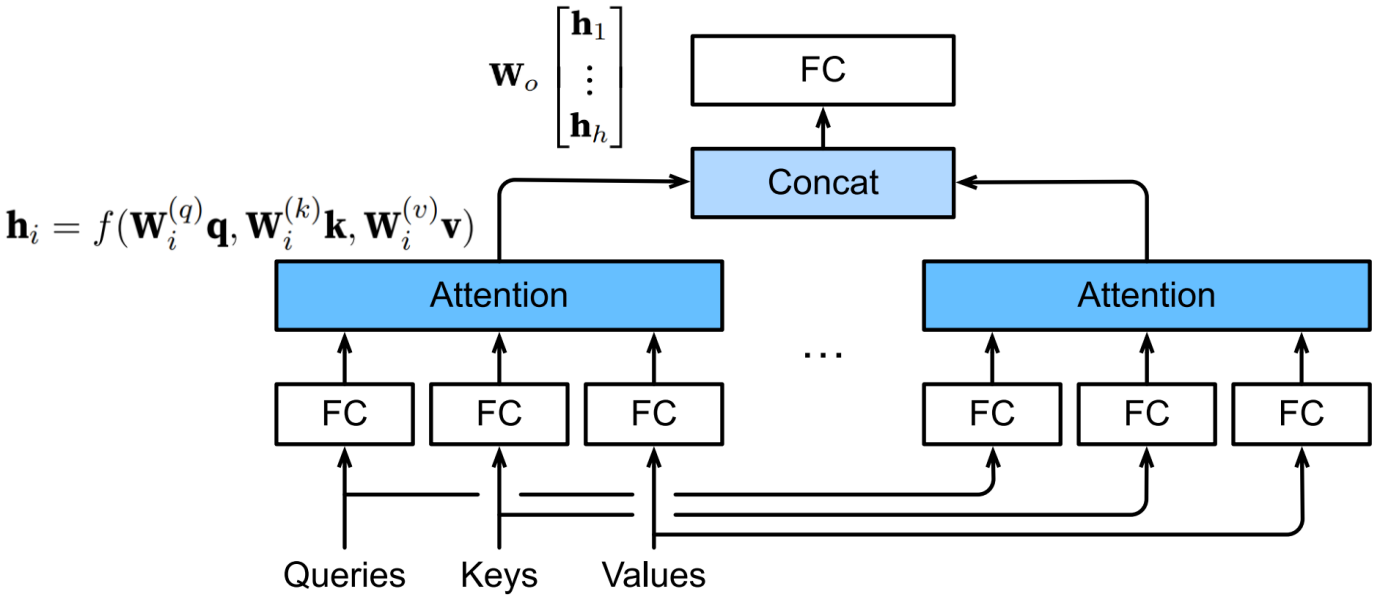


# Multihead Attention

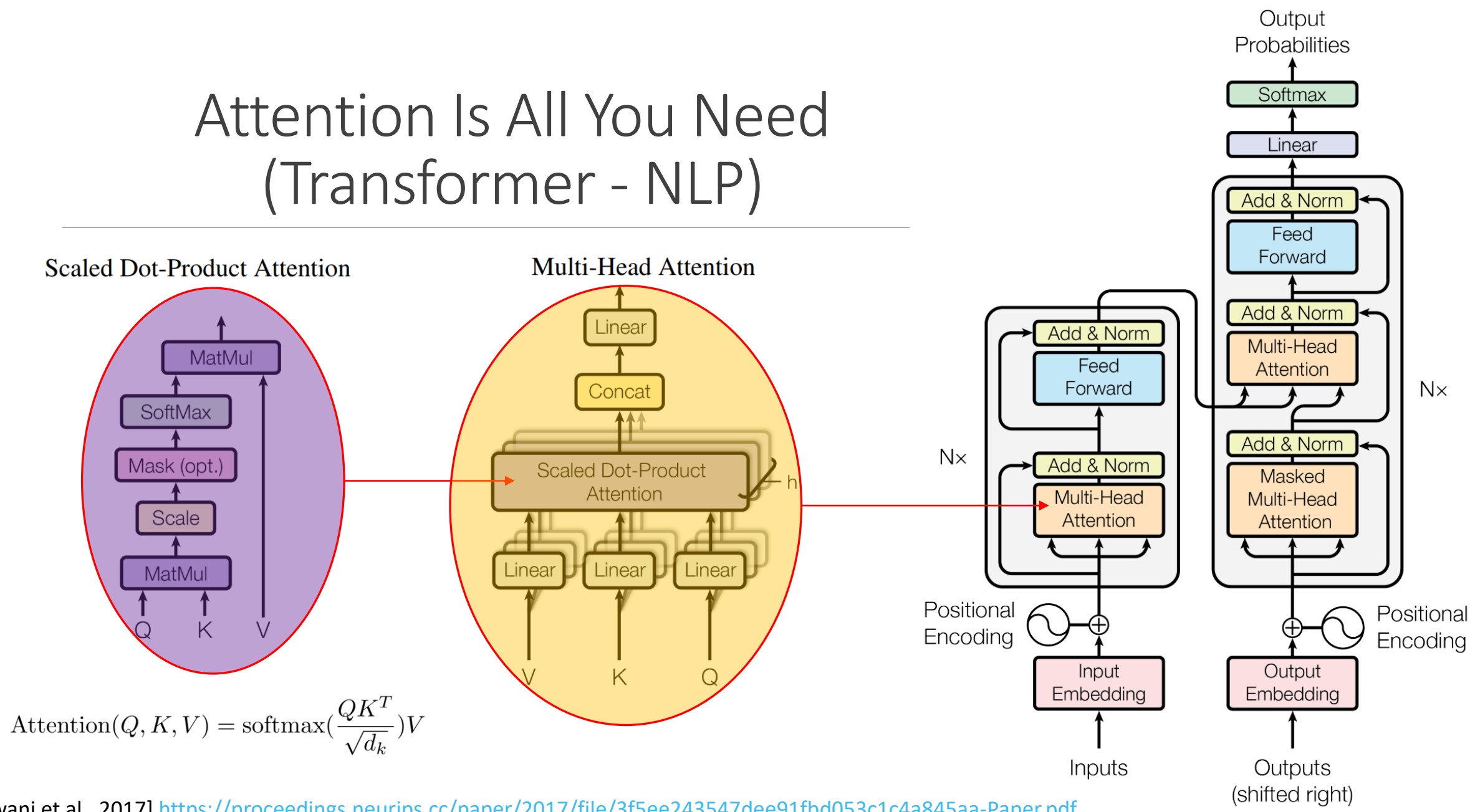
Instead of performing a single attention pooling, queries, keys, and values can be transformed with multiple (h) independently learned linear projections.

Then these h projected queries, keys, and values are fed into attention pooling in parallel.

Finally, the outputs are concatenated and transformed with another learned linear projection to produce the final output.



# Attention Is All You Need (Transformer - NLP)



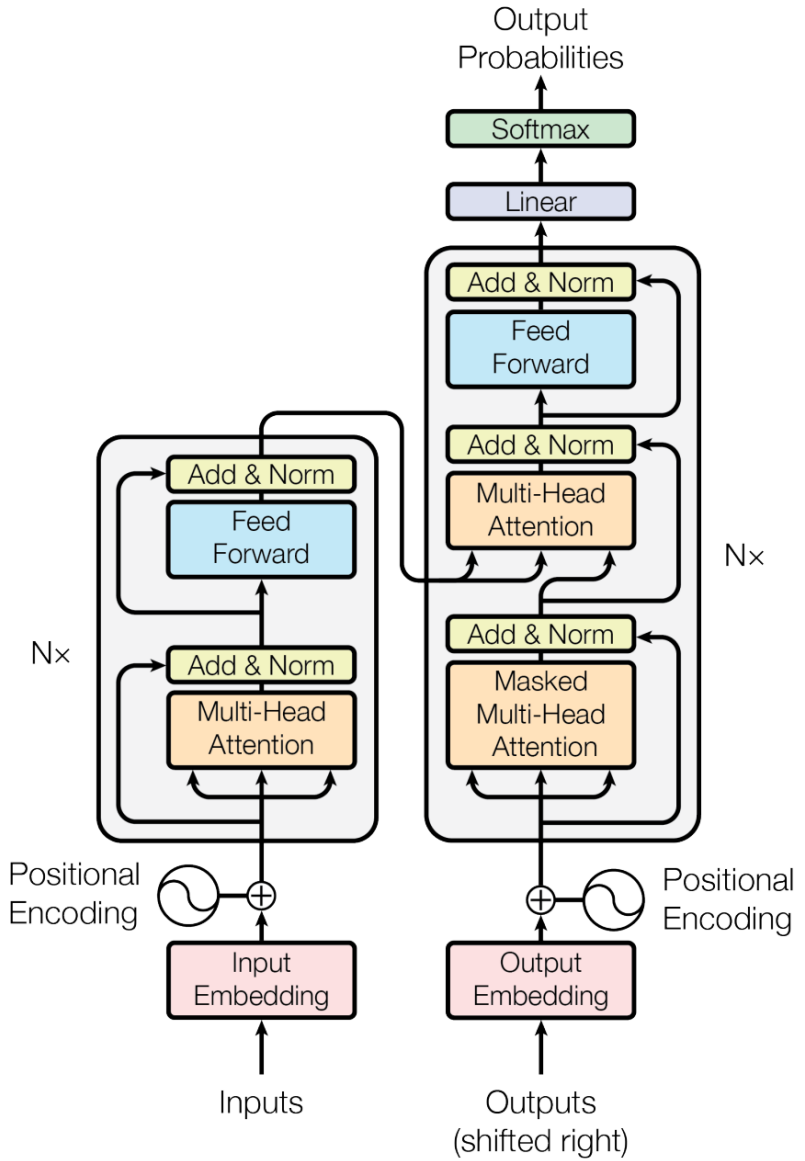

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

[Vaswani et al., 2017] https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

# Attention Is **Not** All You Need

In March 2021, Google researchers published a paper titled “Attention Is Not All You Need: Pure Attention Loses Rank Doubly Exponentially with Depth”.

- self-attention alone converges to a rank 1 matrix.
- skip connections and MLP help to address this issue.
- Layer normalization plays no role.




[Dong et al., 2021] https://arxiv.org/pdf/2103.03404.pdf

# Vision Transformer

In order to calculate the attention of each word with respect to all the others, we have to perform $N^2$ calculations that, which are expensive even parallelizable.

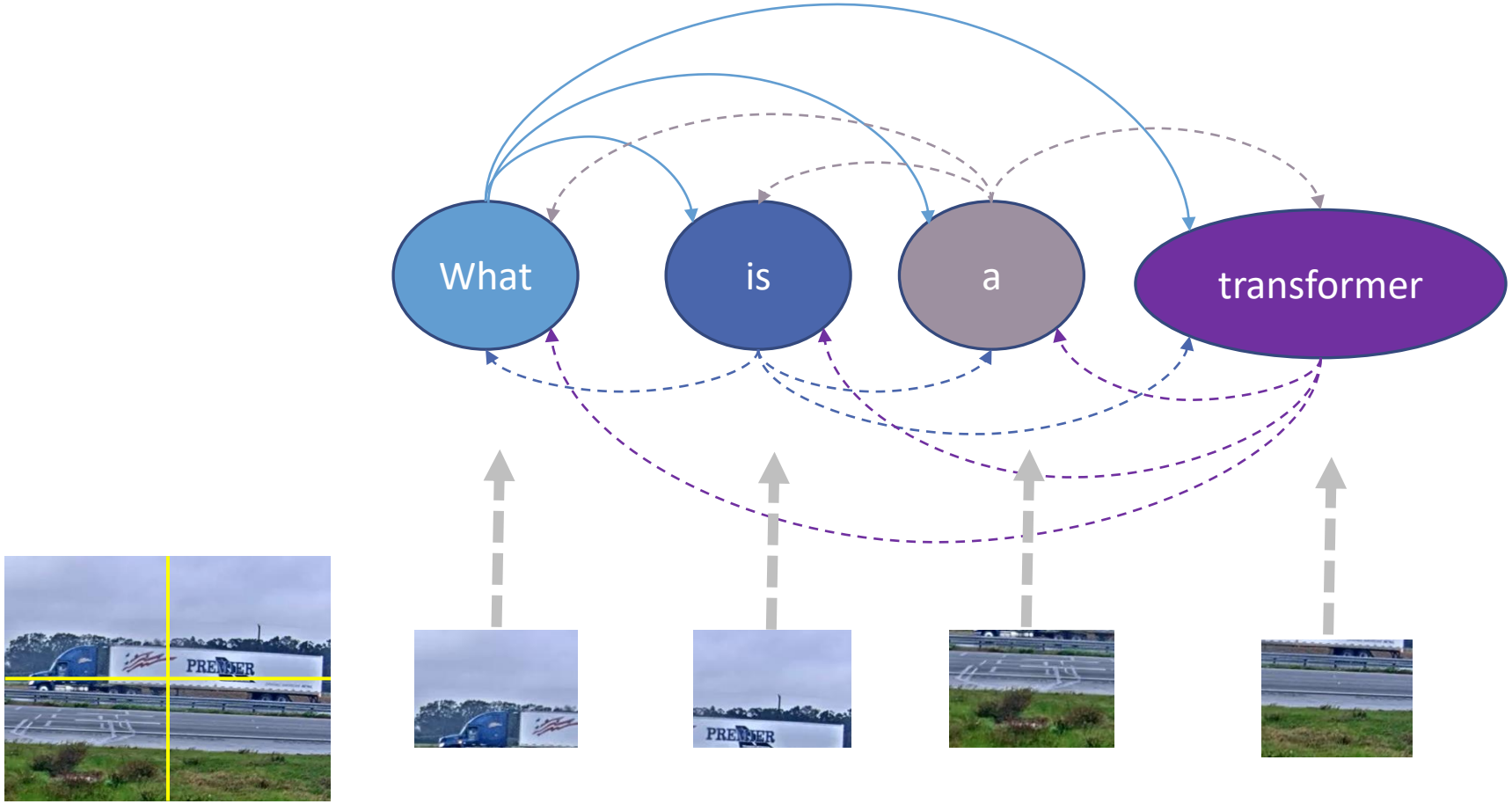
400 px

400 px

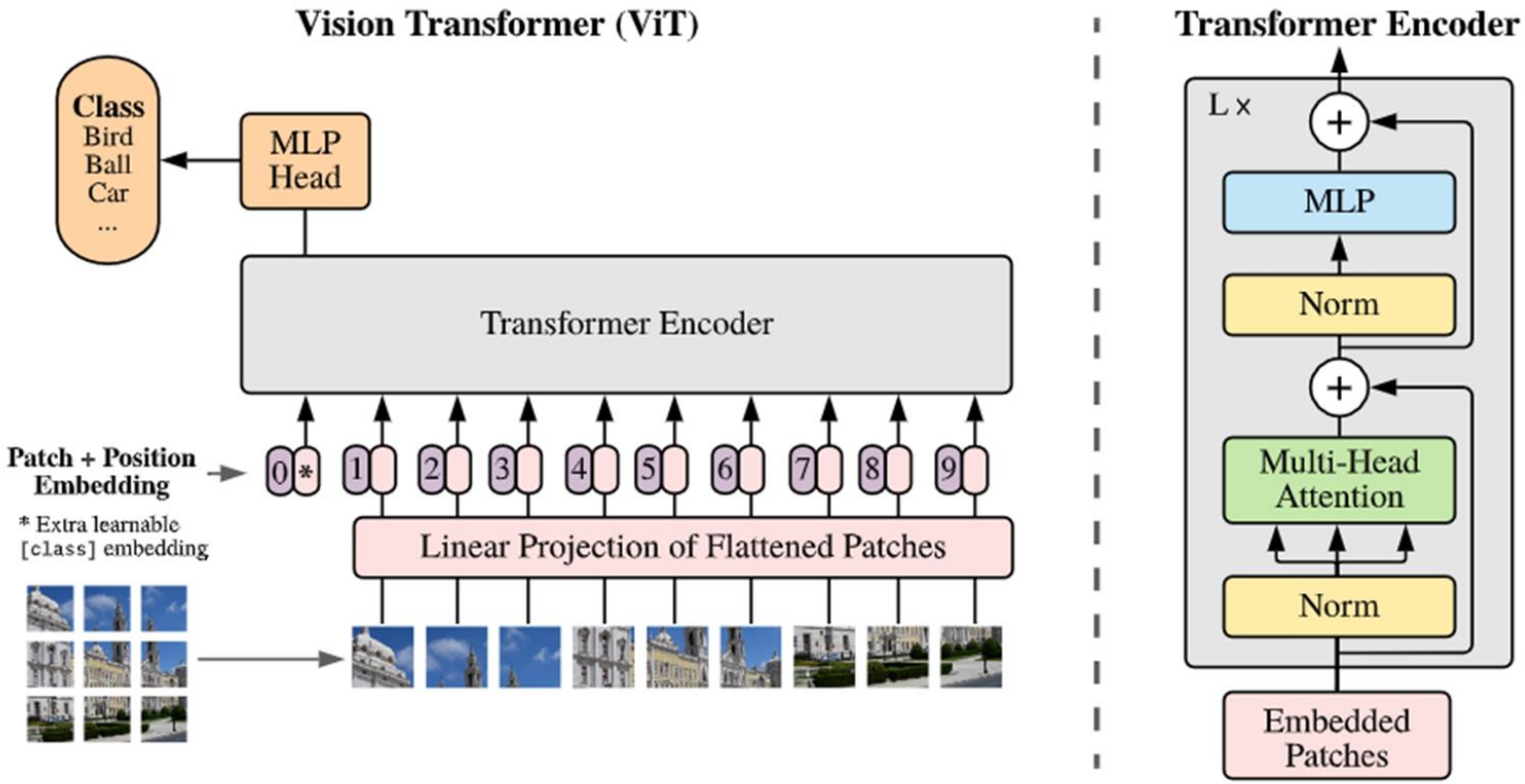
**160,000** px

**25,600,000,000** attention calc

# Analogy



# Vision Transformer (ViT)




*Dosovitskiy et al. (2020) "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale."* https://arxiv.org/pdf/2010.11929
*Phil Wang (2020). Vision Transformer – Pytorch.* https://github.com/lucidrains/vit-pytorch