Lecture 6

Attention & Transformers

Part 1: Pooling: Fixed vs. Adaptive

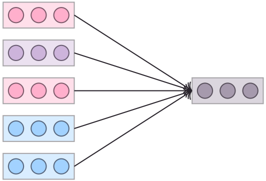
Machine Learning for Structured Data

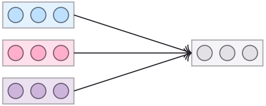
Vlad Niculae · LTL, UvA · https://vene.ro/mlsd

1. Pooling: Fixed vs. Adaptive
2. Hierarchical Attention
3. Self-Attention
4. Transformers

Let's talk about pooling.

$$\mathbf{z} = \text{AveragePool}(\mathbf{z}_1, \ldots, \mathbf{z}_n) := \frac{1}{n}\sum_{j=1}^{n} \mathbf{z}_i$$





Used to get one representation of a variable-size set or sequence.

Combine n input vectors into one single output vector,
with equal contribution.

Let's talk about pooling.

$$\boldsymbol{z} = \text{AveragePool}(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) := \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{z}_i$$

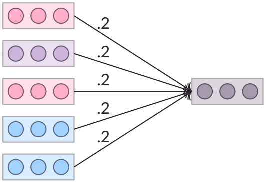
Used to get one representation of a variable-size set or sequence.

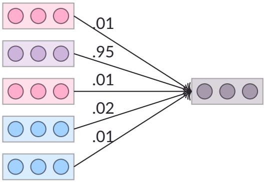
Combine n input vectors into one single output vector,
with equal contribution.

But what if some of the inputs should contribute more than others?

Weighted Average Pooling

$$\boldsymbol{z} = \sum_i \alpha_i \boldsymbol{z}_i, \quad \text{where} \quad \alpha_i \geq 0, \sum_i \alpha_i = 1$$





The weights $\boldsymbol{\alpha}$ control the relative importance of the inputs.

Weighted Average Pooling

$$\boldsymbol{z} = \sum_i \alpha_i \boldsymbol{z}_i, \quad \text{where} \quad \alpha_i \geq 0, \sum_i \alpha_i = 1$$

The weights $\boldsymbol{\alpha}$ control the relative importance of the inputs.

But how to come up with these weights?
How to decide what's important in a given context?

Attention

Key idea: have a representation of the "context" as a vector $\boldsymbol{q} \in \mathbb{R}^d$.

Then, say the importance of $\boldsymbol{z}_i$ is proportional to its alignment ($\sim$ angle) to $\boldsymbol{q}$:

$$\alpha_i = \underbrace{\frac{\exp(\boldsymbol{q} \cdot \boldsymbol{z}_i)}{\sum_j \exp(\boldsymbol{q} \cdot \boldsymbol{z}_j)}}_{[\mathrm{softmax}([\boldsymbol{q}\cdot\boldsymbol{z}_1,\ldots,\boldsymbol{q}\cdot\boldsymbol{z}_n])]_i} \quad ; \qquad \mathrm{Attn}(\boldsymbol{q}; \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) := \sum_i \alpha_i \boldsymbol{z}_i.$$

Attention

Key idea: have a representation of the "context" as a vector $\boldsymbol{q} \in \mathbb{R}^d$.

Then, say the importance of $\boldsymbol{z}_i$ is proportional to its alignment ($\sim$ angle) to $\boldsymbol{q}$:

$$\alpha_i = \underbrace{\frac{\exp(\boldsymbol{q} \cdot \boldsymbol{z}_i)}{\sum_j \exp(\boldsymbol{q} \cdot \boldsymbol{z}_j)}}_{[\text{softmax}([\boldsymbol{q}\cdot\boldsymbol{z}_1,\ldots,\boldsymbol{q}\cdot\boldsymbol{z}_n])]_i} \quad ; \quad \text{Attn}(\boldsymbol{q}; \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) := \sum_i \alpha_i \boldsymbol{z}_i.$$

This is the basic **attention mechanism**:

Pool a bunch of vectors, with varying weights,
depending on how aligned they are with a context.

Attention

Key idea: have a representation of the "context" as a vector $\boldsymbol{q} \in \mathbb{R}^d$.

Then, say the importance of $\boldsymbol{z}_i$ is proportional to its alignment ($\sim$ angle) to $\boldsymbol{q}$:

$$\alpha_i = \underbrace{\frac{\exp(\boldsymbol{q} \cdot \boldsymbol{z}_i)}{\sum_j \exp(\boldsymbol{q} \cdot \boldsymbol{z}_j)}}_{[\text{softmax}([\boldsymbol{q}\cdot\boldsymbol{z}_1,\ldots,\boldsymbol{q}\cdot\boldsymbol{z}_n])]_i} \quad ; \qquad \text{Attn}(\boldsymbol{q}; \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) := \sum_i \alpha_i \boldsymbol{z}_i.$$

This is the basic **attention mechanism**:

Pool a bunch of vectors, with varying weights,
depending on how aligned they are with a context.

What could be the context?

- Could be just a static learned parameter.
- If training on multiple tasks or domains, $\boldsymbol{q}$ can be an embedding of the domain.
- In machine translation (say EN→NL), $\boldsymbol{z}_i$ are the EN words,
  $\boldsymbol{q}$ can be an embedding of the last NL word predicted (one by one).

Attention In Math And Code

$$\alpha_i = \underbrace{\frac{\exp(\boldsymbol{q} \cdot \boldsymbol{z}_i)}{\sum_j \exp(\boldsymbol{q} \cdot \boldsymbol{z}_j)}}_{[\mathrm{softmax}([\boldsymbol{q}\cdot\boldsymbol{z}_1,\ldots,\boldsymbol{q}\cdot\boldsymbol{z}_n])]_i}$$

$$\boldsymbol{z} = \sum_i \alpha_i \boldsymbol{z}_i$$

```
words = [21, 79, 14]  # indices
emb = Embedding(vocab_sz, dim)
# optionally add RNN, CNN, whatever

Z = emb(words)  # (3 × dim)

q = randn(dim)  # (random context)

s = Z @ q
# [-.3, -1.0, 1.8]

alpha = softmax(s, dim=0)
# [.10, .05, .85]

z = alpha @ Z  # (dim)
```

Attention and Expressivity

Attention by itself doesn't make a model more expressive: intuitively, all the same "information" is there in a uniform average too.

But it provides a sort of "shortcut": it makes it easier to represent useful functions.

Lecture 6

Attention & Transformers

Part 2: Hierarchical Attention

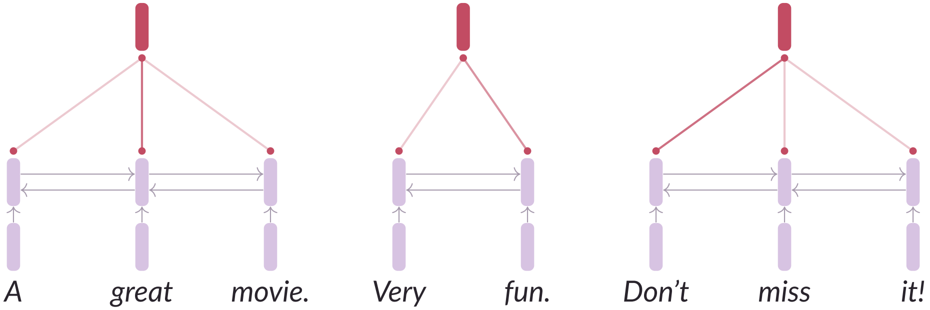
Machine Learning for Structured Data

Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

1. Pooling: Fixed vs. Adaptive
2. Hierarchical Attention
3. Self-Attention
4. Transformers

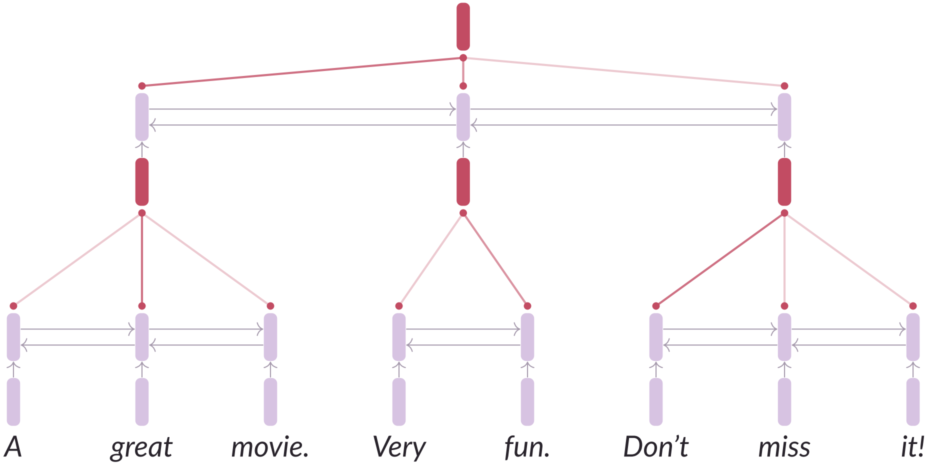
Hierarchical Attention

Encode and pool each sentence separately, then repeat over the sentence vectors.



Hierarchical Attention

Encode and pool each sentence separately, then repeat over the sentence vectors.



Machine Learning for Structured Data
Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

1. Pooling: Fixed vs. Adaptive
2. Hierarchical Attention
3. **Self-Attention**
4. Transformers

Key-Value Attention

Attention resembles set lookup:

Soft lookup (attention):

$$s_j = \boldsymbol{q} \cdot \boldsymbol{z}_j$$

$$\boldsymbol{\alpha} = \text{softmax}(\boldsymbol{s})$$

$$\text{return } \textstyle\sum_i \boldsymbol{\alpha}_i \boldsymbol{z}_i$$

Hard set lookup (by similarity):

$$s_j = \boldsymbol{q} \cdot \boldsymbol{z}_j$$

$$i = \arg\max(\boldsymbol{s})$$

$$\text{return } \boldsymbol{z}_i$$

Key-Value Attention

Attention resembles set lookup:

Soft lookup (attention):

$$s_j = \boldsymbol{q} \cdot \boldsymbol{z}_j$$

$$\boldsymbol{\alpha} = \mathrm{softmax}(\boldsymbol{s})$$

$$\text{return } \sum_i \alpha_i \boldsymbol{z}_i$$

Hard set lookup (by similarity):

$$s_j = \boldsymbol{q} \cdot \boldsymbol{z}_j$$

$$i = \arg\max(\boldsymbol{s})$$

$$\text{return } \boldsymbol{z}_i$$

How about a dictionary lookup? If we have key-value pairs $(\boldsymbol{k}_i, \boldsymbol{v}_i)$,

Key-Value Attention

Attention resembles set lookup:

Soft lookup (attention):

$$s_j = \boldsymbol{q} \cdot \boldsymbol{z}_j$$

$$\boldsymbol{\alpha} = \mathrm{softmax}(\boldsymbol{s})$$

$$\text{return } \textstyle\sum_i \alpha_i \boldsymbol{z}_i$$

Hard set lookup (by similarity):

$$s_j = \boldsymbol{q} \cdot \boldsymbol{z}_j$$

$$i = \arg\max(\boldsymbol{s})$$

$$\text{return } \boldsymbol{z}_i$$

How about a dictionary lookup? If we have key-value pairs $(\boldsymbol{k}_i, \boldsymbol{v}_i)$,

$$s_j = \boldsymbol{q} \cdot \boldsymbol{k}_j$$

$$\boldsymbol{\alpha} = \mathrm{softmax}(\boldsymbol{s})$$

$$\text{return } \textstyle\sum_i \alpha_i \boldsymbol{v}_i$$

Key-Value Attention

Attention resembles set lookup:

Soft lookup (attention):

$$s_j = \boldsymbol{q} \cdot \boldsymbol{z}_j$$

$$\boldsymbol{\alpha} = \text{softmax}(\boldsymbol{s})$$

$$\text{return } \sum_i \alpha_i \boldsymbol{z}_i$$

Hard set lookup (by similarity):

$$s_j = \boldsymbol{q} \cdot \boldsymbol{z}_j$$

$$i = \arg\max(\boldsymbol{s})$$

$$\text{return } \boldsymbol{z}_i$$

How about a dictionary lookup? If we have key-value pairs $(\boldsymbol{k}_i, \boldsymbol{v}_i)$,

$$s_j = \boldsymbol{q} \cdot \boldsymbol{k}_j$$

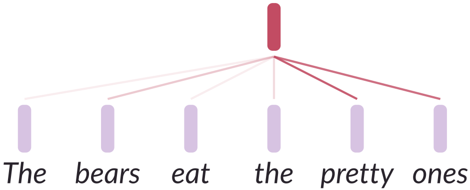
$$\boldsymbol{\alpha} = \text{softmax}(\boldsymbol{s})$$

$$\text{return } \sum_i \alpha_i \boldsymbol{v}_i$$

If we only have item embeddings $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$, we can learn a key/value "views":

$$\boldsymbol{k}_i = \boldsymbol{W}_K^\top \boldsymbol{z}_i$$
$$\boldsymbol{v}_i = \boldsymbol{W}_V^\top \boldsymbol{z}_i$$

Self-Attention

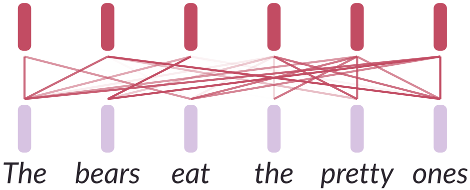


Can we represent an item (word) as a combination of items relevant to it? (i.e., the item is itself the context)

If we had $K = V = Q = Z$ we would always retrieve the item itself.

```
z_out = []

K = Z @ Wk  # (n x dim)
V = Z @ Wv  # (n x dim)
Q = Z @ Wq  # (n x dim)

for i in range(n):
  zi = softmax(K @ Q[i]) @ V
  z_out.append(zi)
```

Self-Attention

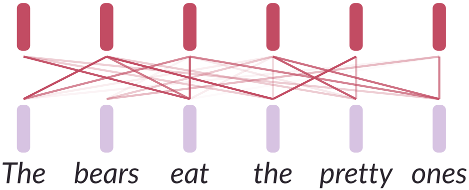


Can we represent an item (word) as a combination of items relevant to it? (i.e., the item is itself the context)

If we had $K = V = Q = Z$ we would always retrieve the item itself.

```
z_out = []

K = Z @ Wk  # (n x dim)
V = Z @ Wv  # (n x dim)
Q = Z @ Wq  # (n x dim)

for i in range(n):
  zi = softmax(K @ Q[i]) @ V
  z_out.append(zi)
```

Self-Attention



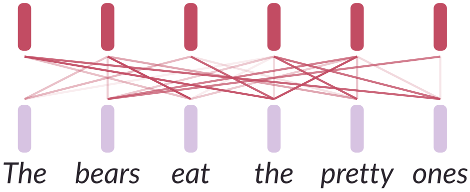

```
z_out = []

K = Z @ Wk  # (n x dim)
V = Z @ Wv  # (n x dim)
Q = Z @ Wq  # (n x dim)

for i in range(n):
  zi = softmax(K @ Q[i]) @ V
  z_out.append(zi)
```

Can we represent an item (word) as a combination of items relevant to it? (i.e., the item is itself the context)

If we had $K = V = Q = Z$ we would always retrieve the item itself.

Attention at each position is independent and parallel. What happens if we permute the inputs?

Self-Attention




```
z_out = []

K = Z @ Wk  # (n x dim)
V = Z @ Wv  # (n x dim)
Q = Z @ Wq  # (n x dim)

for i in range(n):
  zi = softmax(K @ Q[i]) @ V
  z_out.append(zi)
```

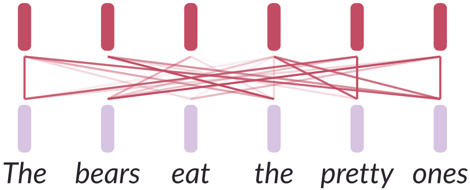
Can we represent an item (word) as a combination of items relevant to it? (i.e., the item is itself the context)

If we had $K = V = Q = Z$ we would always retrieve the item itself.

Attention at each position is independent and parallel. What happens if we permute the inputs?

The outputs just get permuted the same way! (equivariance).

Self-Attention




```
z_out = []

K = Z @ Wk  # (n x dim)
V = Z @ Wv  # (n x dim)
Q = Z @ Wq  # (n x dim)

for i in range(n):
  zi = softmax(K @ Q[i]) @ V
  z_out.append(zi)
```

Can we represent an item (word) as a combination of items relevant to it? (i.e., the item is itself the context)

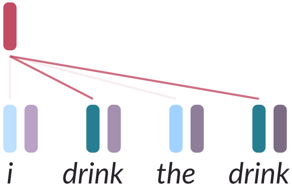
If we had $K = V = Q = Z$ we would always retrieve the item itself.

Attention at each position is independent and parallel. What happens if we permute the inputs?

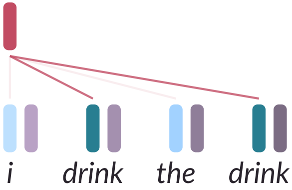
The outputs just get permuted the same way! (equivariance).

So, by default, self-attention **is unaware of sequential order**. (Unlike CNN or RNN encoders!)

Self-Attention for Sequences: Positional Embeddings



Self-Attention for Sequences: Positional Embeddings

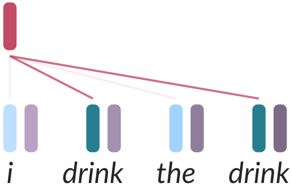


Add to each input vector an "offset" vector that encodes (only) the position in the sequence.

$\tilde{\boldsymbol{z}}_i = \boldsymbol{z}_i + \boldsymbol{p}_i$

Output now depends on the order: if permuting by σ, $\tilde{\boldsymbol{z}}_i = \boldsymbol{z}_{\sigma(i)} + \boldsymbol{p}_i$.

Self-Attention for Sequences: Positional Embeddings



Add to each input vector an "offset" vector that encodes (only) the position in the sequence.

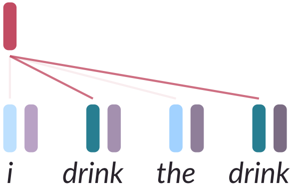
$\tilde{\boldsymbol{z}}_i = \boldsymbol{z}_i + \boldsymbol{p}_i$

Output now depends on the order: if permuting by σ, $\tilde{\boldsymbol{z}}_i = \boldsymbol{z}_{\sigma(i)} + \boldsymbol{p}_i$.

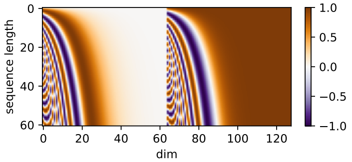
Positional embeddings can be

1. fixed: e.g., based on trig functions of i.
2. learned: i.e., a separate `torch.nn.Embedding(num_embeddings=max_len)` through which we embed the sequence of position indices $(0, 1, 2, \ldots, n-1)$.

Self-Attention for Sequences: Positional Embeddings



fixed sinusoidal embeddings:



Add to each input vector an "offset" vector that encodes (only) the position in the sequence.

$\tilde{\boldsymbol{z}}_i = \boldsymbol{z}_i + \boldsymbol{p}_i$

Output now depends on the order: if permuting by σ, $\tilde{\boldsymbol{z}}_i = \boldsymbol{z}_{\sigma(i)} + \boldsymbol{p}_i$.

Positional embeddings can be

1. fixed: e.g., based on trig functions of i.
2. learned: i.e., a separate `torch.nn.Embedding(num_embeddings=max_len)` through which we embed the sequence of position indices $(0, 1, 2, \ldots, n-1)$.

(2) is easier to code, but (1) can generalize to sequences seen in training, due to its fixed pattern.

Self-Attention for Graphs

But hey: maybe permutation equivariance is sometimes a good thing!

For instance, for **graph neural networks!**

Remember in GNN we computed the message from neighbors as a sum:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_j$$

Self-Attention for Graphs

But hey: maybe permutation equivariance is sometimes a good thing!

For instance, for **graph neural networks!**

Remember in GNN we computed the message from neighbors as a sum:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_i$$

Instead, self-attention over neighbors:

$$\alpha_{ij} = \frac{\exp(\boldsymbol{q}_i \cdot \boldsymbol{k}_j)}{\sum_{j' \in N(i)} \exp(\boldsymbol{q}_i \cdot \boldsymbol{k}_{j'})}$$
$$\boldsymbol{m}_i = \sum_{j \in N(i)} \alpha_{ij} \boldsymbol{v}_j$$

Self-Attention for Graphs

But hey: maybe permutation equivariance is sometimes a good thing!

For instance, for **graph neural networks!**

Remember in GNN we computed the message from neighbors as a sum:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_i$$

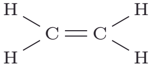
Instead, self-attention over neighbors:

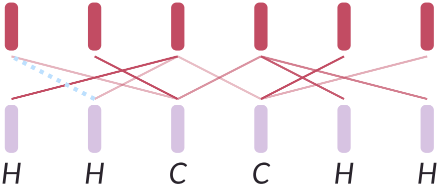
$$\alpha_{ij} = \frac{\exp(\boldsymbol{q}_i \cdot \boldsymbol{k}_j)}{\sum_{j' \in N(i)} \exp(\boldsymbol{q}_i \cdot \boldsymbol{k}_{j'})}$$

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \alpha_{ij} \boldsymbol{v}_j$$

In other words: self-attention constrained by the adjacency structure

(no attention allowed where there is no edge)

Ethylene (C_2H_4): 



Machine Learning for Structured Data
Vlad Niculae · LTL, UvA · `https://vene.ro/mlsd`

1. Pooling: Fixed vs. Adaptive
2. Hierarchical Attention
3. Self-Attention
4. **Transformers**

Multi-Head Attention

Apply attention M times, independently, with different linear transformations to give different keys/vals/queries.

$$
\begin{aligned}
\boldsymbol{k}_i^{(m)} &= \boldsymbol{W}_K^{(m)\top} \boldsymbol{z}_i \\
\boldsymbol{v}_i^{(m)} &= \boldsymbol{W}_V^{(m)\top} \boldsymbol{z}_i \\
\boldsymbol{q}^{(m)} &= \boldsymbol{W}_Q^{(m)\top} \boldsymbol{q} \\
\boldsymbol{z}^{(m)} &= \mathrm{KeyValAttn}(\boldsymbol{q}, \boldsymbol{k}_{1,\ldots,n}^{(m)}, \boldsymbol{v}_{1,\ldots,n}^{(m)})
\end{aligned}
$$

and concatenate the outputs:

$$
\boldsymbol{z} = [\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(M)}]
$$

Intuition: learn M different ways to look at the same set.

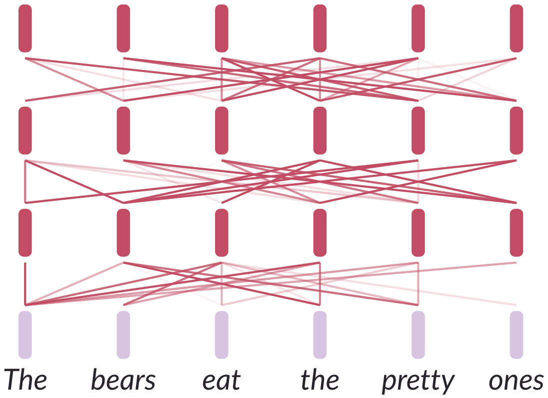
Multi-Head Attention In Code

$$\boldsymbol{k}_i^{(m)} = \boldsymbol{W}_K^{(m)\top} \boldsymbol{z}_i$$
$$\boldsymbol{v}_i^{(m)} = \boldsymbol{W}_V^{(m)\top} \boldsymbol{v}_i$$
$$\boldsymbol{q}^{(m)} = \boldsymbol{W}_Q^{(m)\top} \boldsymbol{q}$$
$$\boldsymbol{z}^{(m)} = \mathrm{KeyValAttn}(\boldsymbol{q}, \boldsymbol{k}_{1,\ldots,n}^{(m)}, \boldsymbol{v}_{1,\ldots,n}^{(m)})$$
$$\boldsymbol{z} = [\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(M)}]$$

```
zms = []

for m in range(M):
  Km = Z @ Wk[m]
  Vm = Z @ Wv[m]
  qm = q @ Wq[m]

  zm = softmax(Km @ qm) @ Vm
  zms.append(zm)

z = cat(zms)
```

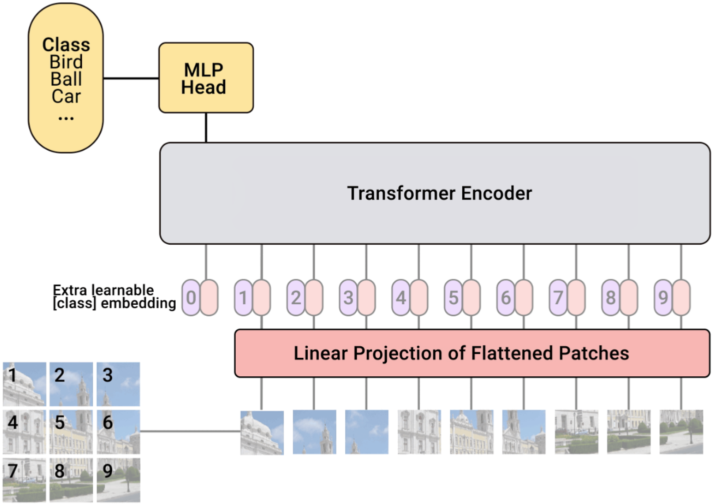
Transformer

Stacked multi-head attention (+ some annoying details like LayerNorm)



- Combines some of the strengths of CNN and RNN:
- Global even without much depth: every output depends on every input.
- Parallelizable: each position and each head can be computed separately. (still one layer at a time)
- Sequence-aware thanks to positional embeddings.

Vision Transformer (ViT)



Source: Google blog https://ai.googleblog.com/2020/12/transformers-for-image-recognition-at.html ©Dosovitskiy, Houlsby, Weissenborn, et al.

Wrapping Up

- Transformers are very popular right now. Important to understand why.
- All things being equal, my (and my friends') experience is that they are **harder to train** than RNNs and CNNs.
- But their parallelizable nature lets them make best use of today's best supercomputing hardware!
- It's not that two Transformer layers > two GRU layers.
  But, we can train deeper Transformers faster and longer (on more data), and currently this looks like the better trade-off.
  This will probably change!