Attention for Sequence Modeling
Attention is (Mostly) All you Need: Transformers
Administrative:

- HW2 due 10/05 11:59pm + 48hr grace period.
- No class this Thu (10/05)
Recurrent Neural Networks: Process Sequences
We can process a sequence of vectors $\mathbf{x}$ by applying a recurrence formula at every time step:

$$ h_t = f_W(h_{t-1}, x_t) $$

new state (vector) \quad old state (vector) \quad input vector at some time step

some function with parameters $W$

Can set initial state $h_0$ to all 0’s
RNN: Computational Graph: Many to Many
Truncated Backpropagation through time

Loss
Vanilla RNN Gradient Flow

Gradients over multiple time steps:

\[
\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}
\]

Always < 1
Vanishing gradients

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Vanilla RNN Gradient Flow

Gradients over multiple time steps:

What if we assumed no non-linearity?

\[
\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}
\]

\[
\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} W^{T-1} \frac{\partial h_1}{\partial W}
\]

Largest eigen value > 1: Exploding gradients

Largest eigen value < 1: Vanishing gradients

We need a new RNN architecture!

Bengio et al., "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al., "On the difficulty of training recurrent neural networks", ICML 2013
Long Short Term Memory (LSTM)

Vanilla RNN

\[ h_t = \tanh \left( W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \]

LSTM

\[
\begin{pmatrix}
    i \\
    f \\
    o \\
    g
\end{pmatrix} = \begin{pmatrix}
    \sigma \\
    \sigma \\
    \sigma \\
    \tanh
\end{pmatrix} W \begin{pmatrix}
    h_{t-1} \\
    x_t
\end{pmatrix}
\]

\[ c_t = f \odot c_{t-1} + i \odot g \]

\[ h_t = o \odot \tanh(c_t) \]

Learn to control information flow from previous state to the next state

Hochreiter and Schmidhuber, “Long Short Term Memory”, Neural Computation 1997
Long Short Term Memory (LSTM)

Vanilla RNN

\[ h_t = \tanh \left( W \left( h_{t-1}, x_t \right) \right) \]

LSTM

\[
\begin{pmatrix}
  i \\
  f \\
  o \\
  g
\end{pmatrix}
= \begin{pmatrix}
  \sigma \\
  \sigma \\
  \sigma \\
  \tanh
\end{pmatrix}
W \begin{pmatrix}
  h_{t-1} \\
  x_t
\end{pmatrix}
\]

\[
c_t = f \odot c_{t-1} + i \odot g
\]

\[ h_t = o \odot \tanh(c_t) \]

Long-term memory \( c \) determines how much information should go into the hidden state \( h \) (short-term memory)

Two “memory vectors”

Hochreiter and Schmidhuber, “Long Short Term Memory”, Neural Computation 1997
Long Short Term Memory (LSTM)
[Hochreiter et al., 1997]
Long Short Term Memory (LSTM)
[Hochreiter et al., 1997]

\[
\begin{align*}
\text{memory from before (h)} & \quad \text{input from below (x)} \\
\text{sigmoid} & \quad \text{sigmoid} \\
\text{sigmoid} & \quad \text{sigmoid} \\
\text{tanh} & \quad \text{tanh} \\
\end{align*}
\]

\[
\begin{align*}
(i) &= \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W (h_{t-1} \begin{pmatrix} x_t \\ c_{t-1} \end{pmatrix}) \\
&= (i, f, o, g) \\
&= \begin{pmatrix} \sigma \odot c_{t-1} + i \odot g \end{pmatrix} \\
&= (i, f, o, g) \\
&= \begin{pmatrix} h_t \end{pmatrix}
\end{align*}
\]
Long Short Term Memory (LSTM)
[Hochreiter et al., 1997]

\[
\begin{align*}
  c_t &= f \odot c_{t-1} + i \odot g \\
  h_t &= o \odot \tanh(c_t)
\end{align*}
\]
Long Short Term Memory (LSTM)  
[Hochreiter et al., 1997]

\[
\begin{align*}
\text{x} & \quad \text{Input gate (i)}, \text{whether to write to cell} \\
\text{ht-1} & \quad \text{Forget gate (f)}, \text{whether to erase cell} \\
\text{h} & \quad \text{Output gate (o)}, \text{how much to reveal cell} \\
\text{g} & \quad \text{Gate gate (?), what to write to cell} \\
\end{align*}
\]
Long Short Term Memory (LSTM)  
[Hochreiter et al., 1997]

- **Input gate** ($i$): Whether to write to cell
- **Forget gate** ($f$): Whether to erase cell
- **Output gate** ($o$): How much to reveal cell
- **Gate gate** ($g$): What to write to cell

The LSTM model is represented by a diagram showing the flow of information:

- Input from below ($x$) and memory from before ($h_{t-1}$) combine through weight matrix $W$.
- Sigmoid and tanh functions are applied to create gates $i$, $f$, $o$, and $g$.
- The state $c_t$ is updated as $c_t = f \odot c_{t-1} + i \odot g$.
- The output $h_t$ is calculated as $h_t = o \odot \tanh(c_t)$.

The diagram includes a mathematical representation of the LSTM equations.
Long Short Term Memory (LSTM)
[Hochreiter et al., 1997]

- **i**: Input gate, whether to write to cell
- **f**: Forget gate, whether to erase cell
- **g**: Gate gate (?), what to write to cell

\[
\begin{pmatrix}
i \\ f \\ o \\ g
\end{pmatrix} =
\begin{pmatrix}
\sigma \\ \sigma \\ \sigma \\ \tanh
\end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}
\]

\[
c_t = f \odot c_{t-1} + i \odot g
\]

\[
h_t = o \odot \tanh(c_t)
\]
Long Short Term Memory (LSTM)  
[Hochreiter et al., 1997]

\[
\begin{align*}
&\mathbf{h}_t = \mathbf{tanh} \left( \mathbf{W} x + \mathbf{LSTM} \right) \\
&\mathbf{LSTM} = \begin{pmatrix}
\mathbf{i} \\
\mathbf{f} \\
\mathbf{o} \\
\mathbf{g}
\end{pmatrix} = \begin{pmatrix}
\sigma \\
\sigma \\
\sigma \\
\tanh
\end{pmatrix} \mathbf{x}_t \mathbf{w} + \begin{pmatrix}
\mathbf{h}_{t-1}
\end{pmatrix} \\
&c_t = \mathbf{f} \odot c_{t-1} + \mathbf{i} \odot \mathbf{g} \\
h_t = \mathbf{o} \odot \tanh(c_t)
\end{align*}
\]

- **i**: Input gate, whether to write to cell
- **f**: Forget gate, whether to erase cell
- **o**: Output gate, how much to reveal cell
- **g**: Gate gate (?), what to write to cell

4h x 2h 4h

input from below (x) memory from before (h) "gates"

\[
\begin{pmatrix}
\mathbf{i} \\
\mathbf{f} \\
\mathbf{o} \\
\mathbf{g}
\end{pmatrix} = \begin{pmatrix}
\sigma \\
\sigma \\
\sigma \\
\tanh
\end{pmatrix} \mathbf{x}_t \mathbf{w} + \begin{pmatrix}
\mathbf{h}_{t-1}
\end{pmatrix}
\]
Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]

Backpropagation from $c_t$ to $c_{t-1}$ only elementwise multiplication by $f$ (forget gate), no matrix multiply with a fixed $W$

$$
\begin{pmatrix}
i \\
f \\
o \\
g \\
\end{pmatrix} =
\begin{pmatrix}
\sigma \\
\sigma \\
\sigma \\
tanh \\
\end{pmatrix} W
\begin{pmatrix}
h_{t-1} \\
x_t \\
\end{pmatrix}
$$

$$
c_t = f \odot c_{t-1} + i \odot g
$$

$$
h_t = o \odot \tanh(c_t)
$$
Long Short Term Memory (LSTM): Gradient Flow
[Hochreiter et al., 1997]

Uninterrupted gradient flow!

Notice that the gradient contains the $f$ gate’s vector of activations
- allows better control of gradients values, using suitable parameter updates of the forget gate.
The hidden state is emitted from $c$ with an output gate ($o$), instead of recurrent multiplication with a weight vector.
Long Short Term Memory (LSTM): Gradient Flow
[Hochreiter et al., 1997]

Q: What if $f = 1$ and $i = 0$?

\[
\begin{pmatrix}
i \\
f \\
o \\
g
\end{pmatrix}
= \begin{pmatrix}
\sigma \\
\sigma \\
\tanh
\end{pmatrix} W \begin{pmatrix}
h_{t-1} \\
x_t
\end{pmatrix}
\]

\[
c_t = f \odot c_{t-1} + i \odot g
\]

\[
h_t = o \odot \tanh(c_t)
\]
Long Short Term Memory (LSTM): Gradient Flow
[Hochreiter et al., 1997]

Q: What if $f = 1$ and $i = 0$?
A: LSTM doesn’t forget / take in new information!

\[
\begin{pmatrix}
  i \\
  f \\
  o \\
  g
\end{pmatrix}
= \begin{pmatrix}
  \sigma \\
  \sigma \\
  \sigma \\
  \tanh
\end{pmatrix}
W \begin{pmatrix}
  h_{t-1} \\
  x_t
\end{pmatrix}
\]

\[
c_t = f \odot c_{t-1} + i \odot g
\]

\[
h_t = o \odot \tanh(c_t)
\]
Long Short Term Memory (LSTM): Gradient Flow
[Hochreiter et al., 1997]

Uninterrupted gradient flow!

Similar to ResNet!

Possible to learn to set $f = 1$ and use $i$ and $g$ to learn “memory residual”
Summary: LSTM

The LSTM architecture makes it easier for the RNN to preserve information over many timesteps
- e.g. if the f = 1 and the i = 0, then the information of that cell is preserved indefinitely.
- By contrast, it’s harder for vanilla RNN to learn a recurrent weight matrix $Wh$ that preserves info in hidden state

LSTM doesn’t guarantee that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies.

Possible to mitigate vanishing / exploding gradient by learning suitable $i$ and $f$.

RNNs / LSTMs are still forgetful. Hard to represent a long sequence with a compact memory vector.
Attention Mechanism
Example: Machine Translation

we are eating bread

estamos comiendo pan
Machine Translation with RNNs

Encoder: $h_t = f_W(x_t, h_{t-1})$
Machine Translation with RNNs

Encoder: $h_t = f_W(x_t, h_{t-1})$

$s_0 = h_4$

Slide credit: Justin Johnson
Machine Translation with RNNs

Encoder: $h_t = f_W(x_t, h_{t-1})$

Decoder: $s_t = g_U(y_t, s_{t-1}, c)$

Slide credit: Justin Johnson
Machine Translation with RNNs

Encoder: $h_t = f_W(x_t, h_{t-1})$

Decoder: $s_t = g_U(y_t, s_{t-1}, c)$

Slide credit: Justin Johnson
Machine Translation with RNNs

Encoder: $h_t = f_W(x_t, h_{t-1})$

Decoder: $s_t = g_U(y_t, s_{t-1}, c)$

Slide credit: Justin Johnson
Machine Translation with RNNs

Encoder: $h_t = f_W(x_t, h_{t-1})$

Decoder: $s_t = g_U(y_t, s_{t-1}, c)$

Problem: $s_t$ is used to encode input and maintain decoder state

Slide credit: Justin Johnson
Machine Translation with RNNs

Encoder: \( h_t = f_W(x_t, h_{t-1}) \)

Decoder: \( s_t = g_U(y_t, s_{t-1}, c) \)

Solution: add a context vector \( c = h_4 \) and generate \( s_0 \) from \( h_4 \)

Slide credit: Justin Johnson
Machine Translation with RNNs

Encoder: $h_t = f_W(x_t, h_{t-1})$
Decoder: $s_t = g_U(y_t, s_{t-1}, c)$

Solution: add a context vector $c = h_4$ and generate $s_0$ from $h_4$

Encoder:
- $h_t = f_W(x_t, h_{t-1})$
- $h_0$, $h_1$, $h_2$, $h_3$, $h_4$

Decoder:
- $s_t = g_U(y_t, s_{t-1}, c)$
- $s_0$, $s_1$, $s_2$, $s_3$, $s_4$

Input:
- $x_1$, $x_2$, $x_3$, $x_4$, $[START]$
- we, are, eating, bread

Output:
- $y_0$, $y_1$, $y_2$, $y_3$, $y_4$, $[STOP]$
- estamos, comiendo, pan

Slide credit: Justin Johnson
Machine Translation with RNNs

Encoder: \( h_t = f_W(x_t, h_{t-1}) \)
Decoder: \( s_t = g_U(y_t, s_{t-1}, c) \)

Problem: Input sequence bottlenecked through fixed-sized memory vector.

Slide credit: Justin Johnson
Machine Translation with RNNs

Encoder: $h_t = f_W(x_t, h_{t-1})$

Decoder: $s_t = g_U(y_t, s_{t-1}, c)$

Idea: can we "look up" information from the input sequence when making prediction?
Machine Translation with RNNs and Attention

From final hidden state:
Initial decoder state $s_0$

we are eating bread

Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

Slide credit: Justin Johnson
Machine Translation with RNNs and Attention

Compute affinity scores

\[ e_{t,i} = f_{\text{att}}(s_{t-1}, h_i) \] (\( f_{\text{att}} \) is an MLP)

From final hidden state:
Initial decoder state \( s_0 \)

- \( e_1 \)
- \( e_2 \)
- \( e_3 \)
- \( e_4 \)
- \( h_1 \)
- \( h_2 \)
- \( h_3 \)
- \( h_4 \)
- \( s_0 \)
- \( x_1 \)
- \( x_2 \)
- \( x_3 \)
- \( x_4 \)

we are eating bread

Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

Slide credit: Justin Johnson
Machine Translation with RNNs and Attention

Compute affinity scores
\[ e_{t,i} = f_{\text{att}}(s_{t-1}, h_i) \] (\( f_{\text{att}} \) is an MLP)

Normalize to get attention weights
\[ 0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1 \]

Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

Slide credit: Justin Johnson
Machine Translation with RNNs and Attention

Think of how much information needed from each input word to generate the first translated word

From final hidden state:
Initial decoder state $s_0$

Compute affinity scores
$$e_{t,i} = f_{\text{att}}(s_{t-1}, h_i)$$ (f_{\text{att}} is an MLP)

Normalize to get attention weights
$$0 < a_{t,i} < 1 \quad \Sigma_i a_{t,i} = 1$$

we, are, eating, bread

Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

Slide credit: Justin Johnson
Machine Translation with RNNs and Attention

Compute **affinity scores**
\[ e_{t,i} = f_{\text{att}}(s_{t-1}, h_i) \] (\(f_{\text{att}}\) is an MLP)

Normalize to get **attention weights**
\[ 0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1 \]

Set context vector \(c\) to a linear combination of hidden states
\[ c_t = \sum_i a_{t,i} h_i \]

---

Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

Slide credit: Justin Johnson
Machine Translation with RNNs and Attention

Compute **affinity scores**

\[ e_{t,i} = f_{\text{att}}(s_{t-1}, h_i) \quad (f_{\text{att}} \text{ is an MLP}) \]

**estamos**

Normalize to get **attention weights**

\[ 0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1 \]

Set context vector \( c \) to a linear combination of hidden states

\[ c_t = \sum_i a_{t,i} h_i \]

---

Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

Slide credit: Justin Johnson
Machine Translation with RNNs and Attention

Set context vector $c_t$ to a linear combination of hidden states:

$$c_t = \sum_i a_{t,i} h_i$$

Normalize to get attention weights:

$$0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$$

Compute affinity scores:

$$e_{t,i} = f_{\text{att}}(s_{t-1}, h_i) \quad (f_{\text{att}} \text{ is an MLP})$$

Intuition: Context vector attends to the relevant part of the input sequence.

“estamos” = “we are”

This is all differentiable! Do not supervise attention weights – backprop through everything.

Slide credit: Justin Johnson
Machine Translation with RNNs and Attention

Repeat: Use $s_1$ to compute attention and get the new context vector $c_2$.

We are eating bread.

Bahtanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

Slide credit: Justin Johnson
Machine Translation with RNNs and Attention

Repeat: Use $s_1$ to compute attention and get the new context vector $c_2$.

Use $c_2$ to compute $s_2, y_2$.

Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

Slide credit: Justin Johnson
Machine Translation with RNNs and Attention

Use a different context vector in each timestep of decoder
- Input sequence not bottlenecked through single vector
- At each timestep of decoder, context vector “looks at” different parts of the input sequence, i.e., attention.

Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015
Machine Translation with RNNs and Attention

Example: English to French translation

Input: “The agreement on the European Economic Area was signed in August 1992.”


Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015
Machine Translation with RNNs and Attention

Example: English to French translation

Input: “The agreement on the European Economic Area was signed in August 1992.”


Diagonal attention means words correspond in order

Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015
**Example**: English to French translation

**Input**: “The agreement on the European Economic Area was signed in August 1992.”

**Output**: “L’accord sur la zone économique européenne a été signé en août 1992.”
Attention Layer

**Inputs:**
- **State vector:** $s_i$ (Shape: $D_Q$)
- **Hidden vectors:** $h_i$ (Shape: $N \times D_H$)
- **Similarity function:** $f_{\text{att}}$

**Computation:**
- **Similarities:** $e_i$ (Shape: $N \times N$) $e_i = f_{\text{att}}(s_{t-1}, h_i)$
- **Attention weights:** $a = \text{softmax}(e)$ (Shape: $N \times N$)
- **Output vector:** $y = \sum_i a_i h_i$ (Shape: $D_X$)
Attention Layer

Inputs:
- **Query vector**: \( q \) (Shape: \( D_Q \))
- **Input vectors**: \( X \) (Shape: \( N_x \times D_x \))

Similarity function: \( f_{\text{att}} \)

Computation:
- **Similarities**: \( e \) (Shape: \( N_x \)) \( e_i = f_{\text{att}}(q, X_i) \)
- **Attention weights**: \( a = \text{softmax}(e) \) (Shape: \( N_x \))
- **Output vector**: \( y = \sum a_i X_i \) (Shape: \( D_x \))

Slide credit: Justin Johnson
### Attention Layer

**Inputs:**
- **Query vector:** $q$ (Shape: $D_Q$)
- **Input vectors:** $X$ (Shape: $N \times D_Q$)

**Similarity function:** dot product

**Computation:**
- **Similarities:** $e$ (Shape: $N$)  
  $e_i = q \cdot X_i$
- **Attention weights:** $a = \text{softmax}(e)$ (Shape: $N$)
- **Output vector:** $y = \sum_i a_i X_i$ (Shape: $D_X$)

**Changes:**
- Use dot product for similarity

Slide credit: Justin Johnson
**Attention Layer**

**Inputs:**
- Query vector: $q$ (Shape: $D_Q$)
- Input vectors: $X$ (Shape: $N \times D_Q$)

**Similarity function:** scaled dot product

**Computation:**
- Similarities: $e$ (Shape: $N \times 1$)  
  
  \[ e_i = q \cdot X_i \sqrt{D} \]

- Attention weights: $a = \text{softmax}(e)$ (Shape: $N \times 1$)

- Output vector: $y = \sum_i a_i X_i$ (Shape: $D_X$)

**Changes:**
- Use **scaled** dot product for similarity
Attention Layer

Inputs:
Query vectors: $Q$ (Shape: $N_Q \times D_Q$)
Input vectors: $X$ (Shape: $N_X \times D_Q$)

Computation:
Similarities: $E = QX^T / \sqrt{D_Q}$ (Shape: $N_Q \times N_X$)
Attention matrix: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_X$)
Output vectors: $Y = AX$ (Shape: $N_Q \times D_X$) $Y_i = \sum_j A_{i,j} X_j$

Changes:
- Use dot product for similarity
- Multiple query vectors
Attention Layer

Inputs:
Query vectors: $Q$ (Shape: $N_Q \times D_Q$)
Input vectors: $X$ (Shape: $N_x \times D_Q$)

Computation:
Similarities: $E = QX^T / \sqrt{D_Q}$ (Shape: $N_Q \times N_x$)
Attention matrix: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_x$)
Output vectors: $Y = AX$ (Shape: $N_Q \times D_x$) $Y_i = \sum_j A_{i,j} X_j$

Attention matrix (A)
Each row sums up to 1

Changes:
- Use dot product for similarity
- Multiple query vectors

Slide credit: Justin Johnson
**Attention Layer**

**Inputs:**
- **Query vectors:** $Q$ (Shape: $N_Q \times D_Q$)
- **Input vectors:** $X$ (Shape: $N_X \times D_X$)
- **Key matrix:** $W_K$ (Shape: $D_X \times D_Q$)
- **Value matrix:** $W_V$ (Shape: $D_X \times D_V$)

**Computation:**
- **Key vectors:** $K = XW_K$ (Shape: $N_X \times D_Q$)
- **Value vectors:** $V = XW_V$ (Shape: $N_X \times D_V$)
- **Similarities:** $E = QK^T$ (Shape: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_X$)
- **Output vectors:** $Y = AV$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

**Problem:** use the same set of input vectors to compute both affinity and output

**Solution:** project input to two sets of vectors: Keys ($K$) and Values ($V$).

**$Q,K,V$ attention:** Compute attention matrix using Queries ($Q$) and Keys ($K$). Then compute output using attention and Values ($V$).

**Changes:**
- Use dot product for similarity
- Multiple query vectors
- Separate key and value

Slide credit: Justin Johnson
Attention Layer

**Inputs:**
- **Query vectors:** $Q$ (Shape: $N_Q \times D_Q$)
- **Input vectors:** $X$ (Shape: $N_X \times D_X$)
- **Key matrix:** $W_K$ (Shape: $D_X \times D_Q$)
- **Value matrix:** $W_V$ (Shape: $D_X \times D_V$)

**Computation:**
- **Key vectors:** $K = XW_K$ (Shape: $N_X \times D_Q$)
- **Value vectors:** $V = XW_V$ (Shape: $N_X \times D_V$)
- **Similarities:** $E = QK^T$ (Shape: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_X$)
- **Output vectors:** $Y = AV$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} V_j$
Attention Layer

**Inputs:**
- **Query vectors:** $Q$ (Shape: $N_Q \times D_Q$)
- **Input vectors:** $X$ (Shape: $N_X \times D_X$)
- **Key matrix:** $W_K$ (Shape: $D_X \times D_Q$)
- **Value matrix:** $W_V$ (Shape: $D_X \times D_V$)

**Computation:**
- **Key vectors:** $K = XW_K$ (Shape: $N_X \times D_Q$)
- **Value vectors:** $V = XW_V$ (Shape: $N_X \times D_V$)
- **Similarities:** $E = QK^T$ (Shape: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_X$)
- **Output vectors:** $Y = AV$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} V_j$
Attention Layer

**Inputs:**
Query vectors: \( Q \) (Shape: \( N_Q \times D_Q \))
Input vectors: \( X \) (Shape: \( N_X \times D_X \))
Key matrix: \( W_K \) (Shape: \( D_X \times D_Q \))
Value matrix: \( W_V \) (Shape: \( D_X \times D_V \))

**Computation:**
Key vectors: \( K = X W_K \) (Shape: \( N_X \times D_Q \))
Value vectors: \( V = X W_V \) (Shape: \( N_X \times D_V \))
Similarities: \( E = Q K^T \) (Shape: \( N_Q \times N_X \)) \( E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q} \)
Attention weights: \( A = \text{softmax}(E, \text{dim}=1) \) (Shape: \( N_Q \times N_X \))
Output vectors: \( Y = A V \) (Shape: \( N_Q \times D_V \)) \( Y_i = \sum_j A_{i,j} V_j \)

Slide credit: Justin Johnson
Attention Layer

**Inputs:**
- **Query vectors:** $Q$ (Shape: $N_Q \times D_Q$
- **Input vectors:** $X$ (Shape: $N_X \times D_X$
- **Key matrix:** $W_K$ (Shape: $D_X \times D_Q$
- **Value matrix:** $W_V$ (Shape: $D_X \times D_V$

**Computation:**
- **Key vectors:** $K = XW_K$ (Shape: $N_X \times D_Q$
- **Value vectors:** $V = XW_V$ (Shape: $N_X \times D_V$
- **Similarities:** $E = QK^T$ (Shape: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_X$
- **Output vectors:** $Y = AV$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Slide credit: Justin Johnson
**Attention Layer**

**Inputs:**
- Query vectors: $Q$ (Shape: $N_Q \times D_Q$)
- Input vectors: $X$ (Shape: $N_X \times D_X$)
- Key matrix: $W_K$ (Shape: $D_X \times D_Q$)
- Value matrix: $W_V$ (Shape: $D_X \times D_V$)

**Computation:**
- Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
- Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
- Similarities: $E = QK^T$ (Shape: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_X$)
- Output vectors: $Y = AV$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} V_j$
Attention Layer

**Inputs:**
- Query vectors: \( Q \) (Shape: \( N_Q \times D_Q \))
- Input vectors: \( X \) (Shape: \( N_X \times D_X \))
- Key matrix: \( W_K \) (Shape: \( D_X \times D_Q \))
- Value matrix: \( W_V \) (Shape: \( D_X \times D_V \))

**Computation:**
- Key vectors: \( K = XW_K \) (Shape: \( N_X \times D_Q \))
- Value vectors: \( V = XW_V \) (Shape: \( N_X \times D_V \))
- Similarities: \( E = QK^T \) (Shape: \( N_Q \times N_X \)) \( E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q} \)
- Attention weights: \( A = \text{softmax}(E, \text{dim}=1) \) (Shape: \( N_Q \times N_X \))
- Output vectors: \( Y = AV \) (Shape: \( N_Q \times D_V \)) \( Y_i = \sum_j A_{i,j} V_j \)
Attention Layer

**Inputs:**
- Query vectors: \( Q \) (Shape: \( N_Q \times D_Q \) )
- Input vectors: \( X \) (Shape: \( N_X \times D_X \) )
- Key matrix: \( W_K \) (Shape: \( D_X \times D_Q \) )
- Value matrix: \( W_V \) (Shape: \( D_X \times D_V \) )

**Computation:**
- Key vectors: \( K = XW_K \) (Shape: \( N_X \times D_Q \) )
- Value vectors: \( V = XW_V \) (Shape: \( N_X \times D_V \) )
- Similarities: \( E = QK^T \) (Shape: \( N_Q \times N_X \) ) \( E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q} \)
- Attention weights: \( A = \text{softmax}(E, \text{dim}=1) \) (Shape: \( N_Q \times N_X \) )
- Output vectors: \( Y = AV \) (Shape: \( N_Q \times D_V \) ) \( Y_i = \sum_j A_{i,j} V_j \)

Attention seems to be really powerful ...
Do we still need RNN?
RNN is bad at encoding long-range relationships!

Recurrent update can easily “forget” information
Attention Layer

**Inputs:**
- Query vectors: $Q$ (Shape: $N_Q \times D_Q$)
- Input vectors: $X$ (Shape: $N_X \times D_X$)
- Key matrix: $W_K$ (Shape: $D_X \times D_Q$)
- Value matrix: $W_V$ (Shape: $D_X \times D_V$)

**Computation:**
- Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
- Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
- Similarities: $E = QK^T$ (Shape: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_X$)
- Output vectors: $Y = AV$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Attention seems to be really powerful ... Do we still need RNN?

Can we use only **attention layers** to encode an entire sequence?
Self-Attention Layer
Sequence encode -> use each input element as query!

**Inputs:**
**Input vectors:** $X$ (Shape: $N_x \times D_x$)
**Key matrix:** $W_K$ (Shape: $D_x \times D_Q$)
**Value matrix:** $W_V$ (Shape: $D_x \times D_V$)
**Query matrix:** $W_Q$ (Shape: $D_x \times D_Q$)

**Computation:**
**Query vectors:** $Q = XW_Q$
**Key vectors:** $K = XW_K$ (Shape: $N_x \times D_Q$)
**Value vectors:** $V = XW_V$ (Shape: $N_x \times D_V$)
**Similarities:** $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
**Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
**Output vectors:** $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

**Goal:** encode the input sequence with only attention, without a recurrent network.

Slide credit: Justin Johnson
Self-Attention Layer

Sequence encode -> use each input element as query!

**Inputs:**
- **Input vectors:** $X$ (Shape: $N_x \times D_x$)
- **Key matrix:** $W_k$ (Shape: $D_x \times D_Q$)
- **Value matrix:** $W_v$ (Shape: $D_x \times D_V$)
- **Query matrix:** $W_q$ (Shape: $D_x \times D_Q$)

**Computation:**
- Query vectors: $Q = XW_Q$
- Key vectors: $K = XW_k$ (Shape: $N_x \times D_Q$)
- Value vectors: $V = XW_v$ (Shape: $N_x \times D_V$)
- Similarities: $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
- Output vectors: $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

**Goal:** encode the input sequence with only attention, without a recurrent network.

**Encoding only -> no external queries**

*Use each element to query other elements*

Slide credit: Justin Johnson
Self-Attention Layer

Sequence encode -> use each input element as query!

**Inputs:**

**Input vectors:** $X$ (Shape: $N_x \times D_x$)

**Key matrix:** $W_K$ (Shape: $D_x \times D_Q$)

**Value matrix:** $W_V$ (Shape: $D_x \times D_V$)

**Query matrix:** $W_Q$ (Shape: $D_x \times D_Q$)

**Computation:**

**Query vectors:** $Q = XW_Q$

**Key vectors:** $K = XW_K$ (Shape: $N_x \times D_Q$)

**Value vectors:** $V = XW_V$ (Shape: $N_x \times D_V$)

**Similarities:** $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$

**Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)

**Output vectors:** $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$
Self-Attention Layer
Sequence encode -> use each input element as query!

Inputs:
Input vectors: $X$ (Shape: $N_x \times D_x$)
Key matrix: $W_K$ (Shape: $D_x \times D_Q$)
Value matrix: $W_V$ (Shape: $D_x \times D_V$)
Query matrix: $W_Q$ (Shape: $D_x \times D_Q$)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N_x \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_x \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
Output vectors: $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Slide credit: Justin Johnson
Self-Attention Layer
Sequence encode -> use each input element as query!

Inputs:
Input vectors: \( X \) (Shape: \( N_x \times D_x \))
Key matrix: \( W_K \) (Shape: \( D_x \times D_Q \))
Value matrix: \( W_V \) (Shape: \( D_x \times D_V \))
Query matrix: \( W_Q \) (Shape: \( D_x \times D_Q \))

Computation:
Query vectors: \( Q = X W_Q \)
Key vectors: \( K = X W_K \) (Shape: \( N_x \times D_Q \))
Value vectors: \( V = X W_V \) (Shape: \( N_x \times D_V \))
Similarities: \( E = QK^T \) (Shape: \( N_x \times N_x \)) \( E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q} \)
Attention weights: \( A = \text{softmax}(E, \text{dim}=1) \) (Shape: \( N_x \times N_x \))
Output vectors: \( Y = AV \) (Shape: \( N_x \times D_V \)) \( Y_i = \sum_j A_{i,j} V_j \)
Self-Attention Layer

Sequence encode -> use each input element as query!

**Inputs:**
- Input vectors: \( X \) (Shape: \( N_x \times D_x \))
- Key matrix: \( W_k \) (Shape: \( D_x \times D_Q \))
- Value matrix: \( W_v \) (Shape: \( D_x \times D_V \))
- Query matrix: \( W_q \) (Shape: \( D_x \times D_Q \))

**Computation:**
- Query vectors: \( Q = X W_q \)
- Key vectors: \( K = X W_k \) (Shape: \( N_x \times D_Q \))
- Value vectors: \( V = X W_v \) (Shape: \( N_x \times D_V \))
- Similarities: \( E = Q K^T \) (Shape: \( N_x \times N_x \)) \( E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q} \)
- Attention weights: \( A = \text{softmax}(E, \text{dim}=1) \) (Shape: \( N_x \times N_x \))
- Output vectors: \( Y = AV \) (Shape: \( N_x \times D_V \)) \( Y_i = \sum_j A_{i,j} V_j \)

Slide credit: Justin Johnson
Self-Attention Layer

Sequence encode -> use each input element as query!

Inputs:
Input vectors: $X$ (Shape: $N_x \times D_x$)
Key matrix: $W_K$ (Shape: $D_x \times D_Q$)
Value matrix: $W_V$ (Shape: $D_x \times D_V$)
Query matrix: $W_Q$ (Shape: $D_x \times D_Q$)

Computation:
Query vectors: $Q = X W_Q$
Key vectors: $K = X W_K$ (Shape: $N_x \times D_Q$)
Value vectors: $V = X W_V$ (Shape: $N_x \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
Output vectors: $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$
Self-Attention Layer
Sequence encode -> use each input element as query!

Inputs:
Input vectors: $X$ (Shape: $N \times D_x$)
Key matrix: $W_K$ (Shape: $D_x \times D_Q$)
Value matrix: $W_V$ (Shape: $D_x \times D_V$)
Query matrix: $W_Q$ (Shape: $D_x \times D_Q$)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N \times D_V$)
Similarities: $E = QK^T$ (Shape: $N \times N$) $E_{i,j} = Q_i \cdot K_j \big/ \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N \times N$)
Output vectors: $Y = AV$ (Shape: $N \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Slide credit: Justin Johnson
**Self-Attention Layer**

Sequence encode -> use each input element as query!

**Inputs:**
- **Input vectors:** $X$ (Shape: $N_x \times D_x$)
- **Key matrix:** $W_K$ (Shape: $D_x \times D_Q$)
- **Value matrix:** $W_V$ (Shape: $D_x \times D_V$)
- **Query matrix:** $W_Q$ (Shape: $D_x \times D_Q$)

**Computation:**
- **Query vectors:** $Q = XW_Q$
- **Key vectors:** $K = XW_K$ (Shape: $N_x \times D_Q$)
- **Value vectors:** $V = XW_V$ (Shape: $N_x \times D_V$)
- **Similarities:** $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
- **Output vectors:** $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Q: Can we use self-attention to encode an input with specific sequential ordering?

Slide credit: Justin Johnson
**Self-Attention Layer**

**Inputs:**
- **Input vectors:** $X$ (Shape: $N \times D_x$)
- **Key matrix:** $W_K$ (Shape: $D_x \times D_Q$)
- **Value matrix:** $W_V$ (Shape: $D_x \times D_V$)
- **Query matrix:** $W_Q$ (Shape: $D_x \times D_Q$)

**Computation:**
- **Query vectors:** $Q = XW_Q$
- **Key vectors:** $K = XW_K$ (Shape: $N \times D_Q$)
- **Value vectors:** $V = XW_V$ (Shape: $N \times D_V$)
- **Similarities:** $E = \mathbf{QK}^\top$ (Shape: $N \times N$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N \times N$)
- **Output vectors:** $Y = AV$ (Shape: $N \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

---

Consider **permuting** the input vectors:

Slide credit: Justin Johnson
Self-Attention Layer

Consider **permuting** the input vectors:

Queries and Keys will be the same, but permuted

**Inputs:**
- Input vectors: $X$ (Shape: $N_x \times D_x$)
- Key matrix: $W_K$ (Shape: $D_x \times D_Q$)
- Value matrix: $W_V$ (Shape: $D_x \times D_V$)
- Query matrix: $W_Q$ (Shape: $D_x \times D_Q$

**Computation:**
- Query vectors: $Q = XW_Q$
- Key vectors: $K = XW_K$ (Shape: $N_x \times D_Q$)
- Value Vectors: $V = XW_V$ (Shape: $N_x \times D_V$)
- Similarities: $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
- Output vectors: $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$
Self-Attention Layer

Inputs:
Input vectors: $X$ (Shape: $N_x \times D_x$)
Key matrix: $W_K$ (Shape: $D_x \times D_Q$)
Value matrix: $W_V$ (Shape: $D_x \times D_V$)
Query matrix: $W_Q$ (Shape: $D_x \times D_Q$)

Consider **permuting** the input vectors:

Similarities will be the same, but permuted

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N_x \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_x \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \dim=1)$ (Shape: $N_x \times N_x$)
Output vectors: $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$
**Self-Attention Layer**

**Inputs:**
- **Input vectors:** \(X\) (Shape: \(N_x \times D_x\))
- **Key matrix:** \(W_k\) (Shape: \(D_x \times D_Q\))
- **Value matrix:** \(W_v\) (Shape: \(D_x \times D_V\))
- **Query matrix:** \(W_q\) (Shape: \(D_x \times D_Q\))

**Computation:**
- **Query vectors:** \(Q = XW_q\)
- **Key vectors:** \(K = XW_k\) (Shape: \(N_x \times D_Q\))
- **Value vectors:** \(V = XW_v\) (Shape: \(N_x \times D_V\))
- **Similarities:** \(E = QQ^T\) (Shape: \(N_x \times N_x\)) \(E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}\)
- **Attention weights:** \(A = \text{softmax}(E, \dim=1)\) (Shape: \(N_x \times N_x\))
- **Output vectors:** \(Y = AV\) (Shape: \(N_x \times D_V\)) \(Y_i = \sum_j A_{i,j} V_j\)

**Consider permuting the input vectors:**
- Attention weights will be the same, but permuted

---

Slide credit: Justin Johnson
Self-Attention Layer

Consider **permuting** the input vectors:

Values will be the same, but permuted

**Inputs:**
- **Input vectors:** $X$ (Shape: $N_x \times D_x$)
- **Key matrix:** $W_k$ (Shape: $D_x \times D_Q$)
- **Value matrix:** $W_v$ (Shape: $D_x \times D_V$)
- **Query matrix:** $W_q$ (Shape: $D_x \times D_Q$)

**Computation:**
- **Query vectors:** $Q = XW_Q$
- **Key vectors:** $K = XW_k$ (Shape: $N_x \times D_Q$)
- **Value vectors:** $V = XW_v$ (Shape: $N_x \times D_V$)
- **Similarities:** $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
- **Output vectors:** $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$
Consider permuting the input vectors:

Outputs will be the same, but permuted

**Inputs:**
- **Input vectors:** $X$ (Shape: $N \times D_X$)
- **Key matrix:** $W_K$ (Shape: $D_X \times D_Q$)
- **Value matrix:** $W_V$ (Shape: $D_X \times D_V$)
- **Query matrix:** $W_Q$ (Shape: $D_X \times D_Q$)

**Computation:**
- **Query vectors:** $Q = XW_Q$
- **Key vectors:** $K = XW_K$ (Shape: $N \times D_Q$)
- **Value vectors:** $V = XW_V$ (Shape: $N \times D_V$)
- **Similarities:** $E = QK^T$ (Shape: $N \times N_X$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_X \times N_X$)
- **Output vectors:** $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$
Consider **permuting** the input vectors:

Outputs will be the same, but permuted

Self-attention layer is **Permutation Equivariant**

\[ f(s(x)) = s(f(x)) \]

**Inputs:**
- **Input vectors:** \( X \) (Shape: \( N \times D_x \))
- **Key matrix:** \( W_K \) (Shape: \( D_x \times D_Q \))
- **Value matrix:** \( W_V \) (Shape: \( D_x \times D_V \))
- **Query matrix:** \( W_Q \) (Shape: \( D_x \times D_Q \))

**Computation:**
- **Query vectors:** \( Q = XW_Q \)
- **Key vectors:** \( K = XW_K \) (Shape: \( N \times D_Q \))
- **Value vectors:** \( V = XW_V \) (Shape: \( N \times D_V \))
- **Similarities:** \( E = QK^T \) (Shape: \( N \times N \)) \( E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q} \)
- **Attention weights:** \( A = \text{softmax}(E, \ dim=1) \) (Shape: \( N \times N \))
- **Output vectors:** \( Y = AV \) (Shape: \( N \times D_V \)) \( Y_i = \sum_j A_{i,j} V_j \)
Self-Attention Layer

Inputs:
- Input vectors: $X$ (Shape: $N_x \times D_x$)
- Key matrix: $W_K$ (Shape: $D_x \times D_Q$)
- Value matrix: $W_V$ (Shape: $D_x \times D_V$)
- Query matrix: $W_Q$ (Shape: $D_x \times D_Q$)

Self attention doesn’t “know” the order of the vectors it is processing! Not good for sequence encoding.

Computation:
- Query vectors: $Q = XW_Q$
- Key vectors: $K = XW_K$ (Shape: $N_x \times D_Q$)
- Value vectors: $V = XW_V$ (Shape: $N_x \times D_V$)
- Similarities: $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
- Output vectors: $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Slide credit: Justin Johnson
Self-Attention Layer

Inputs:
- Input vectors: $X$ (Shape: $N_x \times D_x$)
- Key matrix: $W_K$ (Shape: $D_x \times D_Q$)
- Value matrix: $W_V$ (Shape: $D_x \times D_V$)
- Query matrix: $W_Q$ (Shape: $D_x \times D_Q$)

Computation:
- Query vectors: $Q = XW_Q$
- Key vectors: $K = XW_K$ (Shape: $N_x \times D_Q$)
- Value vectors: $V = XW_V$ (Shape: $N_x \times D_V$)
- Similarities: $E = QK^T$ (Shape: $N_x \times N_x$) $E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
- Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
- Output vectors: $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

In order to make processing position-aware, concatenate input with positional encoding $E$

$E(i)$ encodes the position of the $i$-th element in a sequence

$E()$ can be a simple function (e.g., linear or sin functions) or a learned lookup table.
Aside: Positional Encoding (PE) for Self-Attention

**Motivation:** Maintain the order of input data since attention mechanisms are permutation invariant. PEs are shared across all input sequences.

**Linear Positional Encoding:** $PE(pos) = a \cdot pos + b$.
Problem: encoding increases with the sequence length, causing gradient problem for long sequences.

**Sin/cos Positional Encoding** (Default):

\[
PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}}) \]
\[
PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}}) \]

PE for each dimension (i) repeats periodically, combine different waveforms at each dimension to get a unique embedding.

**Learned Positional Encoding:** $PE_{\theta}(pos, i)$.
Learn the most suitable position embedding for the training set.
Masked Self-Attention Layer

Inputs:
- **Input vectors:** $X$ (Shape: $N_x \times D_x$)
- **Key matrix:** $W_K$ (Shape: $D_x \times D_Q$)
- **Value matrix:** $W_V$ (Shape: $D_x \times D_V$)
- **Query matrix:** $W_Q$ (Shape: $D_x \times D_Q$)

Don’t let vectors “look ahead” in the sequence

Used for sequence decoding (predict next word)

Computation:
- **Query vectors:** $Q = XW_Q$
- **Key vectors:** $K = XW_K$ (Shape: $N_x \times D_Q$)
- **Value vectors:** $V = XW_V$ (Shape: $N_x \times D_V$)
- **Similarities:** $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
- **Output vectors:** $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Slide credit: Justin Johnson
Multi-headed Self-Attention Layer

**Inputs:**
- **Input vectors:** $X$ (Shape: $N_x \times D_x$)
- **Key matrix:** $W_k$ (Shape: $D_x \times D_Q$)
- **Value matrix:** $W_v$ (Shape: $D_x \times D_V$)
- **Query matrix:** $W_q$ (Shape: $D_x \times D_Q$)

**Computation:**
- **Query vectors:** $Q = XW_q$
- **Key vectors:** $K = XW_k$ (Shape: $N_x \times D_Q$)
- **Value vectors:** $V = XW_v$ (Shape: $N_x \times D_V$)
- **Similarities:** $E = QK^T$ (Shape: $N_x \times N_x$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)
- **Output vectors:** $Y = AV$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} V_j$
Multi-headed Self-Attention Layer

**Inputs:**
- **Input vectors:** $X$ (Shape: $N \times D_X$)
- **Key matrix:** $W_K$ (Shape: $D_K \times D_Q$)
- **Value matrix:** $W_V$ (Shape: $D_V \times D_V$)
- **Query matrix:** $W_Q$ (Shape: $D_Q \times D_Q$)

**Computation:**
- **Query vectors:** $Q = XW_Q$
- **Key vectors:** $K = XW_K$ (Shape: $N \times D_Q$)
- **Value vectors:** $V = XW_V$ (Shape: $N \times D_V$)
- **Similarities:** $E = QK^T$ (Shape: $N \times N$) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
- **Attention weights:** $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N \times N$)
- **Output vectors:** $Y = AV$ (Shape: $N \times D_V$) $Y_i = \sum_j A_{i,j}V_j$

**Use** $H$ independent “Attention Heads” in parallel

**Highly parallelizable:** Can compute attentions for all input element from all head in parallel!
Three Ways of Processing Sequences

Recurrent Neural Network

Works on **Ordered Sequences**
(+) Natural sequential processing: “sees” the input sequence in its original ordering
(-) Forgetful: difficult to handle long-range dependencies.
(-) Not parallelizable: need to compute hidden states sequentially

Slide credit: Justin Johnson
Three Ways of Processing Sequences

Recurrent Neural Network

Works on **Ordered Sequences**

+ Natural sequential processing: “sees” the input sequence in its original ordering

- Forgetful: difficult to handle long-range dependencies.

- Not parallelizable: need to compute hidden states sequentially

1D Convolution

Works on **Multidimensional Grids**

- Bad at long sequences: Need to stack many conv layers for outputs to “see” the whole sequence

+ Highly parallel: Each output can be computed in parallel

Slide credit: Justin Johnson
Three Ways of Processing Sequences

**Recurrent Neural Network**
- Works on **Ordered Sequences**
  (+) Natural sequential processing: “sees” the input sequence in its original ordering
  (-) Forgetful: difficult to handle long-range dependencies.
  (-) Not parallelizable: need to compute hidden states sequentially

**1D Convolution**
- Works on **Multidimensional Grids**
  (+) Good at long sequences: after one self-attention layer, each output “sees” all inputs!
  (+) Highly parallel: Each output can be computed in parallel
  (-) Very memory intensive
  (-) Requires positional encoding

**Self-Attention**
- Works on **Sets of Vectors**
  (+) Good at long sequences: after one self-attention layer, each output “sees” all inputs!
  (+) Highly parallel: Each output can be computed in parallel
  (-) Very memory intensive
  (-) Requires positional encoding

*Slide credit: Justin Johnson*
Three Ways of Processing Sequences

<table>
<thead>
<tr>
<th>Recurrent Neural Network</th>
<th>1D Convolution</th>
<th>Self-Attention</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attention is all you need</strong></td>
<td><strong>Vaswani et al, NeurIPS 2017</strong></td>
<td></td>
</tr>
</tbody>
</table>

Works on **Ordered Sequences**

(+): Natural sequential processing: “sees” the input sequence in its original ordering
(-): Forgetful: difficult to handle long-range dependencies.
(-): Not parallelizable: need to compute hidden states sequentially

Works on **Multidimensional Grids**

(-): Bad at long sequences: Need to stack many conv layers for outputs to “see” the whole sequence
(+): Highly parallel: Each output can be computed in parallel

Works on **Sets of Vectors**

(+): Good at long sequences: after one self-attention layer, each output “sees” all inputs!
(+): Highly parallel: Each output can be computed in parallel
(-): Very memory intensive
(-): Requires positional encoding

Slide credit: Justin Johnson
The Transformer Block

Vaswani et al, “Attention is all you need”, NeurIPS 2017
The Transformer Block

All vectors interact with each other

Vaswani et al, “Attention is all you need”, NeurIPS 2017

Slide credit: Justin Johnson
The Transformer Block

MLP independently on each vector

All vectors interact with each other

Vaswani et al, “Attention is all you need”, NeurIPS 2017

Slide credit: Justin Johnson
The Transformer Block

MLP independently on each vector

Residual connection

All vectors interact with each other

Vaswani et al, "Attention is all you need", NeurIPS 2017

Slide credit: Justin Johnson
The Transformer Block

Recall **Layer Normalization**:
Given \(h_1, \ldots, h_N\) (shape: D)
scale: \(\gamma\) (shape: D)
shift: \(\beta\) (shape: D)
\[
\mu_i = \frac{1}{D}\sum_j h_{i,j} \quad \text{(scalar)}
\]
\[
\sigma_i = \sqrt{\sum_j (h_{i,j} - \mu_i)^2} \quad \text{(scalar)}
\]
\[
z_i = \frac{h_i - \mu_i}{\sigma_i} \quad \text{(shape: D)}
\]
\[
y_i = \gamma * z_i + \beta \quad \text{(shape: D)}
\]

MLP independently on each vector

Residual connection

All vectors interact with each other

Applied **per element**, not across the sequence

---

Vaswani et al, “Attention is all you need”, NeurIPS 2017

Slide credit: Justin Johnson
The Transformer Block

Residual connection
MLP independently on each vector

Residual connection
All vectors interact with each other

Layer Normalization
Self-Attention
MLP

Vaswani et al, “Attention is all you need”, NeurIPS 2017
Slide credit: Justin Johnson
**The Transformer Block**

**Transformer Block:**
- **Input:** Set of vectors $x$
- **Output:** Set of vectors $y$

Self-attention is the only interaction among vectors!

Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable

---

Vaswani et al, “Attention is all you need”, NeurIPS 2017

Slide credit: Justin Johnson
The Transformer

**Transformer Block:**
*Input:* Set of vectors $x$
*Output:* Set of vectors $y$

Self-attention is the only interaction among vectors!

Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable

A **Transformer** is a sequence of transformer blocks

Vaswani et al, "Attention is all you need", NeurIPS 2017

Slide credit: Justin Johnson
The Transformer

Encoder-Decoder

Vaswani et al. “Attention is all you need”, NeurIPS 2017
Visualizing Transformer Attentions

https://github.com/jessevig/bertviz
In a shocking finding, scientists discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

The scientist named the population, after their distinctive horn, Ovid’s Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Pérez and the others then ventured further into the valley. “By the time we reached the top of one peak, the water looked blue, with some crystals on top,” said Pérez.

Source: OpenAI, “Better Language Models and Their Implications”
https://openai.com/blog/better-language-models/
Can Attention/Transformers be used from more than text processing?
Encoding/Decoding Protein Structures (AlphaFold)

https://www.nature.com/articles/s41586-021-03819-2
Predicting Multi-agent Behaviors

Yuan et al., 2021 AgentFormer: Agent-Aware Transformers for Socio-Temporal Multi-Agent Forecasting
ViT: Vision Transformer

An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (Dosovitskiy et al., 2021)
ViT: Vision Transformer

Generally more expensive to train and execute than ConvNets-based models
Formal Algorithms for Transformers

Mary Phuong and Marcus Hutter

1DeepMind

This document aims to be a self-contained, mathematically precise overview of transformer architectures and algorithms (not results). It covers what transformers are, how they are trained, what they are used for, their key architectural components, and a preview of the most prominent models. The reader is assumed to be familiar with basic ML terminology and simpler neural network architectures such as MLPs.

Keywords: formal algorithms, pseudocode, transformers, attention, encoder, decoder, BERT, GPT, Gopher, tokenization, training, inference.

Contents
1 Introduction 1
2 Motivation 3
3 Transformers and Typical Tasks 4
4 Tokenization: How Text is Represented 4
5 Architectural Components 7
6 Transformer Architectures 8
7 Transformer Training and Inference 8
8 Practical Considerations 9
A References 9
B List of Notation 16

A famous colleague once sent an actually very well-written paper he was quite proud of to a famous complexity theorist. His answer: "I can't find a theorem in the paper, I have no idea what this complete, precise and compact overview of transformer architectures and formal algorithms (but not results) is. It covers what Transformers are (Section 6), how they are trained (Section 7), what they're used for (Section 3), their key architectural components (Section 5), tokenization (Section 4), and a preview of practical considerations (Section 8) and the most prominent models.

The essentially complete pseudocode is about 50 lines, compared to thousands of lines of actual real source code. We believe these formal algorithms will be useful for theoreticians who require compact, complete, and precise formulations, experimental researchers interested in implementing a Transformer from scratch, and
Summary

Self-Attention

Transformer Model

Beyond Language
Next time - Training Large Language Models
Instructor: Will Held