CS 784: Computational Linguistics
Lecture 9: Neural Networks II
(for Text Classification)

Freda Shi

School of Computer Science, University of Waterloo
fhs@uwaterloo.ca

February 25, 2025

Check out https://pytorch.org/tutorials/intermediate/nlp_from_scratch_index.html if you aren't familiar with this topic!

Recap: Unified View of Text Classification

$$\text{classify}(s) = \arg\max_{y} \mathit{score}(s, y; \mathbf{\Theta})$$

s: input text, y: class label, $\mathbf{\Theta}$: model parameters.

Recap: Unified View of Text Classification

$$\text{classify}(s) = \arg\max_{y} \mathit{score}(s, y; \boldsymbol{\Theta})$$

s: input text, y: class label, $\boldsymbol{\Theta}$: model parameters.

Model $\mathit{score}(s, y; \boldsymbol{\Theta})$ using a neural network.

Recap: Unified View of Text Classification

$$\text{classify}(s) = \arg\max_{y} \textit{score}(s, y; \mathbf{\Theta})$$

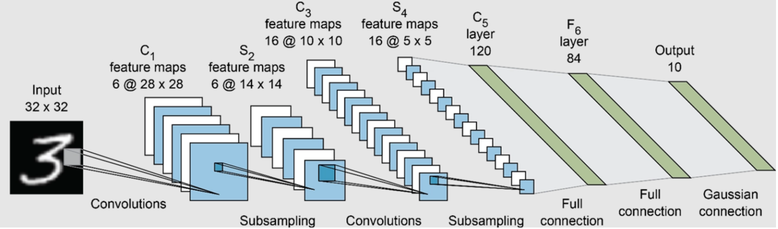
s: input text, y: class label, $\mathbf{\Theta}$: model parameters.

Model $\textit{score}(s, y; \mathbf{\Theta})$ using a neural network.

Last lecture: represent s as a fixed-dimensional vector $\mathbf{x}$ using bag-of-word embeddings.

This lecture: extract more powerful features $\mathbf{x}$ of s using advanced neural network architectures.

Convolutional Neural Networks



Introduced in the context of computer vision, but also used for text classification.

The Convolutional Kernel

$x_{1,5}$	$x_{2,5}$	$x_{3,5}$	$x_{4,5}$	$x_{5,5}$
$x_{1,4}$	$x_{2,4}$	$x_{3,4}$	$x_{4,4}$	$x_{5,4}$
$x_{1,3}$	$x_{2,3}$	$x_{3,3}$	$x_{4,3}$	$x_{5,3}$
$x_{1,2}$	$x_{2,2}$	$x_{3,2}$	$x_{4,2}$	$x_{5,2}$
$x_{1,1}$	$x_{2,1}$	$x_{3,1}$	$x_{4,1}$	$x_{5,1}$

$\longrightarrow$

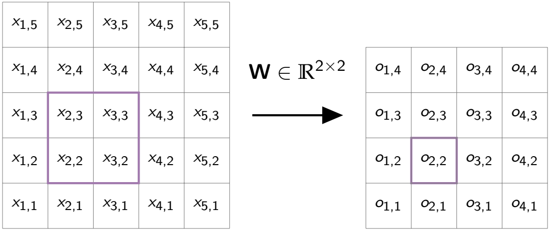
$o_{1,4}$	$o_{2,4}$	$o_{3,4}$	$o_{4,4}$
$o_{1,3}$	$o_{2,3}$	$o_{3,3}$	$o_{4,3}$
$o_{1,2}$	$o_{2,2}$	$o_{3,2}$	$o_{4,2}$
$o_{1,1}$	$o_{2,1}$	$o_{3,1}$	$o_{4,1}$

$$\mathbf{O} = \mathbf{X} * \mathbf{W} \qquad \mathbf{X} \in \mathbb{R}^{n \times m}, \mathbf{W} \in \mathbb{R}^{k \times \ell}, \mathbf{O} \in \mathbb{R}^{(n-k+1)\times(m-\ell+1)}$$

$$o_{i,j} = \sum_{p=1}^{k} \sum_{q=1}^{\ell} x_{i+p-1,j+q-1} \cdot w_{p,q}$$

- A kernel is a small matrix (e.g., 2×2) that slides over the input. At each position, kernel computes element-wise multiplication and sum of the input and kernel.

The Convolutional Kernel

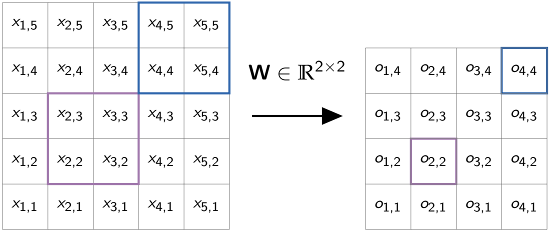


$$\mathbf{O} = \mathbf{X} * \mathbf{W} \qquad \mathbf{X} \in \mathbb{R}^{n\times m}, \mathbf{W} \in \mathbb{R}^{k\times \ell}, \mathbf{O} \in \mathbb{R}^{(n-k+1)\times(m-\ell+1)}$$

$$o_{i,j} = \sum_{p=1}^{k} \sum_{q=1}^{\ell} x_{i+p-1,j+q-1} \cdot w_{p,q}$$

- A kernel is a small matrix (e.g., 2×2) that slides over the input. At each position, kernel computes element-wise multiplication and sum of the input and kernel.

The Convolutional Kernel

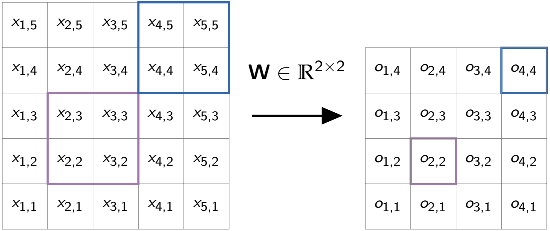


$$\mathbf{O} = \mathbf{X} * \mathbf{W} \qquad \mathbf{X} \in \mathbb{R}^{n \times m}, \mathbf{W} \in \mathbb{R}^{k \times \ell}, \mathbf{O} \in \mathbb{R}^{(n-k+1)\times(m-\ell+1)}$$

$$o_{i,j} = \sum_{p=1}^{k} \sum_{q=1}^{\ell} x_{i+p-1, j+q-1} \cdot w_{p,q}$$

- A kernel is a small matrix (e.g., 2×2) that slides over the input. At each position, kernel computes element-wise multiplication and sum of the input and kernel.

The Convolutional Kernel

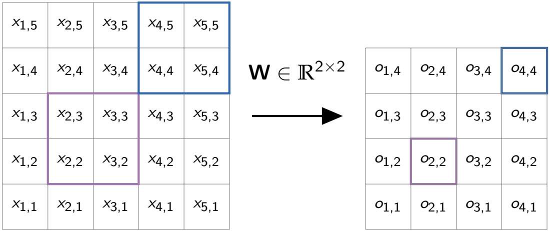


$$\mathbf{O} = \mathbf{X} * \mathbf{W} \qquad \mathbf{X} \in \mathbb{R}^{n\times m}, \mathbf{W} \in \mathbb{R}^{k\times \ell}, \mathbf{O} \in \mathbb{R}^{(n-k+1)\times(m-\ell+1)}$$

$$o_{i,j} = \sum_{p=1}^{k} \sum_{q=1}^{\ell} x_{i+p-1,j+q-1} \cdot w_{p,q}$$

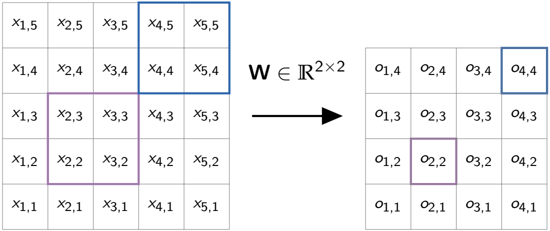
- A kernel is a small matrix (e.g., 2×2) that slides over the input. At each position, kernel computes element-wise multiplication and sum of the input and kernel.
- (Outdated) convention: rotate the kernel by 180 degrees.

Convolutional Neural Networks: Characteristics



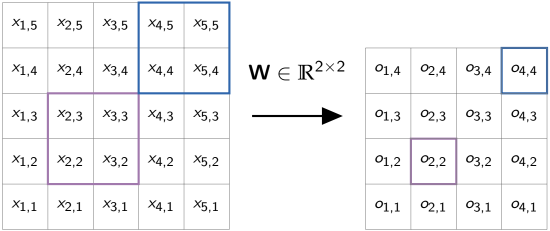
- A convolutional kernel can be thought of as weighted sum over a local region of the input.

Convolutional Neural Networks: Characteristics



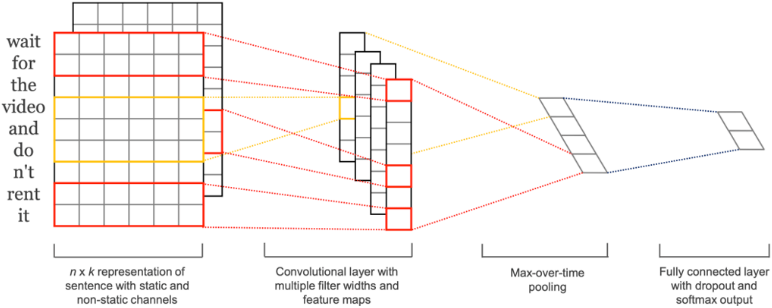
- A convolutional kernel can be thought of as weighted sum over a local region of the input.
- The weights are learnable from data to optimize for downstream task.

Convolutional Neural Networks: Characteristics



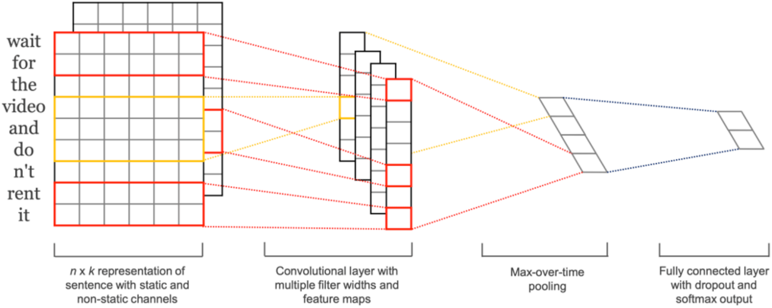
- A convolutional kernel can be thought of as weighted sum over a local region of the input.
- The weights are learnable from data to optimize for downstream task.
- Therefore, a learned kernel is a local feature extractor (e.g., color patterns, edge with a specific shape).

From 2D to 1D: Text CNNs



[Source: Kim, 2014]

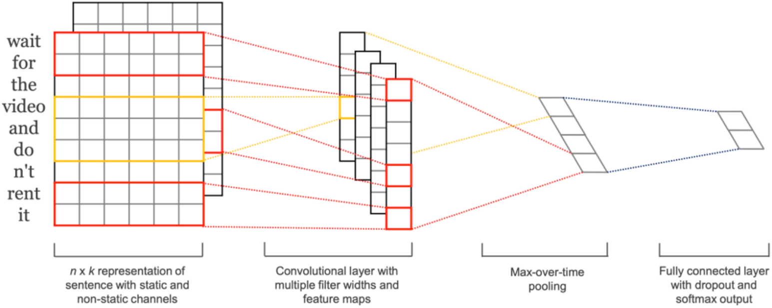
From 2D to 1D: Text CNNs



[Source: Kim, 2014]

- Input $\mathbf{X} \in \mathbb{R}^{n \times d}$.
  n: number of token, d: embedding dimension.

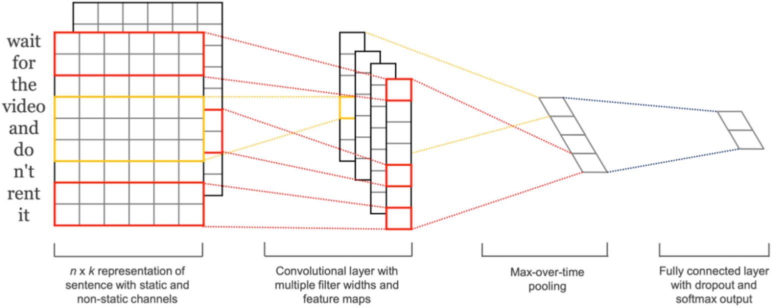
From 2D to 1D: Text CNNs



[Source: Kim, 2014]

- Input $\mathbf{X} \in \mathbb{R}^{n \times d}$.
  n: number of token, d: embedding dimension.
- Kernels $\in \mathbb{R}^{h \times d}$ (kernel size $h << n$).
  Any thoughts on why the second dimension is always d?

From 2D to 1D: Text CNNs



[Source: Kim, 2014]

- Input $\mathbf{X} \in \mathbb{R}^{n \times d}$.
  n: number of token, d: embedding dimension.
- Kernels $\in \mathbb{R}^{h \times d}$ (kernel size $h << n$).
  Any thoughts on why the second dimension is always d?
- Output $\mathbf{O} \in \mathbb{R}^{(n-h+1) \times 1}$:

$$o_i = \sum_{j=1}^{h} \sum_{k=1}^{d} x_{i+j-1,k} \cdot w_{j,k}$$

Obtaining Fixed Dimensional Output

For a convolutional kernel

$$\mathbf{O} = \mathbf{X} * \mathbf{W} \qquad (\in \mathbb{R}^{(n-h+1)\times 1})$$

The output dimension depends on the kernel size h and the input sentence length n.

Obtaining Fixed Dimensional Output

For a convolutional kernel

$$\mathbf{O} = \mathbf{X} * \mathbf{W} \qquad (\in \mathbb{R}^{(n-h+1)\times 1})$$

The output dimension depends on the kernel size h and the input sentence length n.

However, almost all classifiers require a fixed-dimensional input.

Obtaining Fixed Dimensional Output

For a convolutional kernel

$$\mathbf{O} = \mathbf{X} * \mathbf{W} \qquad (\in \mathbb{R}^{(n-h+1)\times 1})$$

The output dimension depends on the kernel size h and the input sentence length n.

However, almost all classifiers require a fixed-dimensional input.

Solution: **pooling** – make the one with variable length fixed!

Obtaining Fixed Dimensional Output

For a convolutional kernel

$$\mathbf{O} = \mathbf{X} * \mathbf{W} \qquad (\in \mathbb{R}^{(n-h+1)\times 1})$$

The output dimension depends on the kernel size h and the input sentence length n.

However, almost all classifiers require a fixed-dimensional input.

Solution: **pooling** – make the one with variable length fixed!

In this case, we will convert $\mathbf{O} \in \mathbb{R}^{(n-h+1)\times 1}$ to a single scalar.

$$\texttt{pooling} : \mathbb{R}^* \to \mathbb{R}$$

Obtaining Fixed Dimensional Output

For a convolutional kernel

$$\mathbf{O} = \mathbf{X} * \mathbf{W} \qquad (\in \mathbb{R}^{(n-h+1)\times 1})$$

The output dimension depends on the kernel size h and the input sentence length n.

However, almost all classifiers require a fixed-dimensional input.

Solution: **pooling** – make the one with variable length fixed!

In this case, we will convert $\mathbf{O} \in \mathbb{R}^{(n-h+1)\times 1}$ to a single scalar.

$$\texttt{pooling} : \mathbb{R}^* \to \mathbb{R}$$

Stacking scalars from **a fixed number of kernels** yields a fixed-dimensional feature vector.

Pooling Mechanisms

$$\texttt{pooling} : \mathbb{R}^* \rightarrow \mathbb{R}$$

More generally, pooling removes one dimension from a **tensor**.

Pooling Mechanisms

$$\texttt{pooling} : \mathbb{R}^* \to \mathbb{R}$$

More generally, pooling removes one dimension from a **tensor**.

Consider the tensor $\mathbf{O} \in \mathbb{R}^{a \times b \times c \times d}$, and we would like to remove the third (c) dimension.

- **Max pooling**: take the maximum from the output of each kernel.

$$\texttt{maxpool}(\mathbf{O})_{i,j,k} = \max_{p=1}^{c} \mathbf{O}_{i,j,p,k}$$

Pooling Mechanisms

$$\texttt{pooling} : \mathbb{R}^* \to \mathbb{R}$$

More generally, pooling removes one dimension from a **tensor**.

Consider the tensor $\mathbf{O} \in \mathbb{R}^{a \times b \times c \times d}$, and we would like to remove the third (c) dimension.

- **Max pooling**: take the maximum from the output of each kernel.

$$\texttt{maxpool}(\mathbf{O})_{i,j,k} = \max_{p=1}^{c} \mathbf{O}_{i,j,p,k}$$

- **Mean pooling**: take the average value from the output of each kernel.

$$\texttt{meanpool}(\mathbf{O})_{i,j,k} = \frac{1}{c} \sum_{p=1}^{c} \mathbf{O}_{i,j,p,k}$$

Pooling Mechanisms

$$\texttt{pooling} : \mathbb{R}^* \to \mathbb{R}$$

More generally, pooling removes one dimension from a **tensor**.

Consider the tensor $\mathbf{O} \in \mathbb{R}^{a \times b \times c \times d}$, and we would like to remove the third (c) dimension.

- **Max pooling**: take the maximum from the output of each kernel.

$$\texttt{maxpool}(\mathbf{O})_{i,j,k} = \max_{p=1}^{c} \mathbf{O}_{i,j,p,k}$$

- **Mean pooling**: take the average value from the output of each kernel.

$$\texttt{meanpool}(\mathbf{O})_{i,j,k} = \frac{1}{c} \sum_{p=1}^{c} \mathbf{O}_{i,j,p,k}$$

- **Attention pooling**: take a weighted average of the output of each kernel.

$$\texttt{attnpool}(\mathbf{O})_{i,j,k} = \sum_{p=1}^{c} \alpha_p \mathbf{O}_{i,j,p,k},$$

where α_p is a data-dependent weight (more on this later).

CNNs as MLPs

The basic form of a 2-layer perceptron:

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$
$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{z}^{(1)} + \mathbf{b}^{(2)}$$

CNNs as MLPs

The basic form of a 2-layer perceptron:

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$
$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{z}^{(1)} + \mathbf{b}^{(2)}$$

The application of one kernel at one position can be expressed as

$$o = \mathbf{W}\mathbf{x},$$

where $\mathbf{W} \in \mathbb{R}^{1\times(h\times d)}$ is the kernel and $\mathbf{x} \in \mathbb{R}^{(h\times d)\times 1}$ is the input.

CNNs as MLPs

The basic form of a 2-layer perceptron:

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$
$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{z}^{(1)} + \mathbf{b}^{(2)}$$

The application of one kernel at one position can be expressed as

$$o = \mathbf{W}\mathbf{x},$$

where $\mathbf{W} \in \mathbb{R}^{1\times(h\times d)}$ is the kernel and $\mathbf{x} \in \mathbb{R}^{(h\times d)\times 1}$ is the input. This corresponds to the first layer without the bias term and activation function—in fact, it is a linear transformation.

Recurrent Neural Networks

Elman (1990), a computational psycholinguist, proposed the simple recurrent neural network (RNN) architecture.

Recurrent Neural Networks

Elman (1990), a computational psycholinguist, proposed the simple recurrent neural network (RNN) architecture.

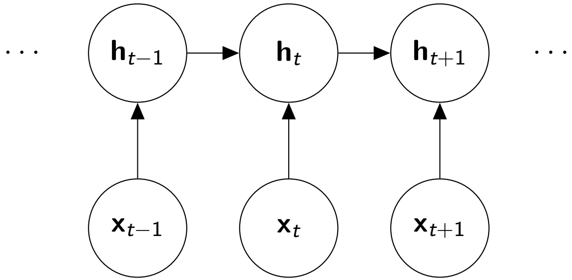
Key idea: apply the same transformation to tokens in time order.

Recurrent Neural Networks

Elman (1990), a computational psycholinguist, proposed the simple recurrent neural network (RNN) architecture.

Key idea: apply the same transformation to tokens in time order.

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$



RNNs: Gradient Update

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$

RNNs: Gradient Update

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$

Suppose $\mathbf{h}_T$ is passed to the classifier as the fixed-dimensional feature vector.

We can easily calculate $\frac{\partial \text{loss}}{\partial \mathbf{h}_T}$, as well as $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ and $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}}$ for each t.

RNNs: Gradient Update

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$

Suppose $\mathbf{h}_T$ is passed to the classifier as the fixed-dimensional feature vector.

We can easily calculate $\frac{\partial \text{loss}}{\partial \mathbf{h}_T}$, as well as $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ and $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}}$ for each t.

What about $\frac{\partial \text{loss}}{\partial \mathbf{W}}$?

RNNs: Gradient Update

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$

Suppose $\mathbf{h}_T$ is passed to the classifier as the fixed-dimensional feature vector.

We can easily calculate $\frac{\partial \text{loss}}{\partial \mathbf{h}_T}$, as well as $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ and $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}}$ for each t.

What about $\frac{\partial \text{loss}}{\partial \mathbf{W}}$?

$$\frac{\partial \text{loss}}{\partial \mathbf{W}} = \sum_{t=1}^{T} \frac{\partial \text{loss}}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}}$$

RNNs: Gradient Update

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$

Suppose $\mathbf{h}_T$ is passed to the classifier as the fixed-dimensional feature vector.

We can easily calculate $\frac{\partial \text{loss}}{\partial \mathbf{h}_T}$, as well as $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ and $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}}$ for each t.

What about $\frac{\partial \text{loss}}{\partial \mathbf{W}}$?

$$\frac{\partial \text{loss}}{\partial \mathbf{W}} = \sum_{t=1}^{T} \frac{\partial \text{loss}}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}}$$

$$\frac{\partial \text{loss}}{\partial \mathbf{h}_t} = \frac{\partial \text{loss}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$$

An Important Issue of Simple RNNs

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$

Suppose $\mathbf{h}_{t+1} = \mathbf{W}\left[\mathbf{h}_t; \mathbf{x}_{t+1}\right] + \mathbf{b} = \alpha \mathbf{h}_t (\alpha \neq 1)$.

What will happen if t goes to $+\infty$?

An Important Issue of Simple RNNs

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$

Suppose $\mathbf{h}_{t+1} = \mathbf{W}\left[\mathbf{h}_t; \mathbf{x}_{t+1}\right] + \mathbf{b} = \alpha \mathbf{h}_t (\alpha \neq 1)$.

What will happen if t goes to $+\infty$?

The norm of $\mathbf{h}_t$ will either explode (if $\alpha > 1$) or vanish (if $\alpha < 1$) as t increases.

An Important Issue of Simple RNNs

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$

Suppose $\mathbf{h}_{t+1} = \mathbf{W}\left[\mathbf{h}_t; \mathbf{x}_{t+1}\right] + \mathbf{b} = \alpha \mathbf{h}_t (\alpha \neq 1)$.

What will happen if t goes to $+\infty$?

The norm of $\mathbf{h}_t$ will either explode (if $\alpha > 1$) or vanish (if $\alpha < 1$) as t increases.

This motivates the development of more advanced RNN architectures.

The Long Short-Term Memory Networks (LSTMs)

Hochreiter & Schmidhuber (1997) proposed the LSTM architecture to address the vanishing/exploding gradient problem.

The Long Short-Term Memory Networks (LSTMs)

Hochreiter & Schmidhuber (1997) proposed the LSTM architecture to address the vanishing/exploding gradient problem.

Forget gate $$\mathbf{f}_t = \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_f\right)$$

The Long Short-Term Memory Networks (LSTMs)

Hochreiter & Schmidhuber (1997) proposed the LSTM architecture to address the vanishing/exploding gradient problem.

$$
\begin{aligned}
\text{Forget gate} \qquad & \mathbf{f}_t = \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_f\right) \\
\text{Input gate} \qquad & \mathbf{i}_t = \sigma\left(\mathbf{W}_i\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_i\right)
\end{aligned}
$$

The Long Short-Term Memory Networks (LSTMs)

Hochreiter & Schmidhuber (1997) proposed the LSTM architecture to address the vanishing/exploding gradient problem.

$$\begin{aligned}
\text{Forget gate} \qquad & \mathbf{f}_t = \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_f\right) \\
\text{Input gate} \qquad & \mathbf{i}_t = \sigma\left(\mathbf{W}_i\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_i\right) \\
\text{Cell} \qquad & \tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_c\right)
\end{aligned}$$

The Long Short-Term Memory Networks (LSTMs)

Hochreiter & Schmidhuber (1997) proposed the LSTM architecture to address the vanishing/exploding gradient problem.

$$
\begin{aligned}
\text{Forget gate} \quad & \mathbf{f}_t = \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_f\right) \\
\text{Input gate} \quad & \mathbf{i}_t = \sigma\left(\mathbf{W}_i\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_i\right) \\
\text{Cell} \quad & \tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_c\right) \\
\text{Update} \quad & \mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t
\end{aligned}
$$

$\odot$: element-wise multiplication.

The Long Short-Term Memory Networks (LSTMs)

Hochreiter & Schmidhuber (1997) proposed the LSTM architecture to address the vanishing/exploding gradient problem.

$$
\begin{aligned}
\text{Forget gate} \qquad & \mathbf{f}_t = \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_f\right) \\
\text{Input gate} \qquad & \mathbf{i}_t = \sigma\left(\mathbf{W}_i\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_i\right) \\
\text{Cell} \qquad & \tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_c\right) \\
\text{Update} \qquad & \mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \\
\text{Output gate} \qquad & \mathbf{o}_t = \sigma\left(\mathbf{W}_o\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_o\right)
\end{aligned}
$$

$\odot$: element-wise multiplication.

The Long Short-Term Memory Networks (LSTMs)

Hochreiter & Schmidhuber (1997) proposed the LSTM architecture to address the vanishing/exploding gradient problem.

Forget gate	$\mathbf{f}_t = \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_f\right)$
Input gate	$\mathbf{i}_t = \sigma\left(\mathbf{W}_i\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_i\right)$
Cell	$\tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_c\right)$
Update	$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$
Output gate	$\mathbf{o}_t = \sigma\left(\mathbf{W}_o\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_o\right)$
Hidden state	$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$

$\odot$: element-wise multiplication.

The Long Short-Term Memory Networks (LSTMs)

Hochreiter & Schmidhuber (1997) proposed the LSTM architecture to address the vanishing/exploding gradient problem.

Forget gate	$\mathbf{f}_t = \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_f\right)$
Input gate	$\mathbf{i}_t = \sigma\left(\mathbf{W}_i\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_i\right)$
Cell	$\tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_c\right)$
Update	$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$
Output gate	$\mathbf{o}_t = \sigma\left(\mathbf{W}_o\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_o\right)$
Hidden state	$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$

$\odot$: element-wise multiplication.

Key idea: keep entries in $\tilde{\mathbf{c}}_t$ and $\mathbf{h}_t$ in the range $[-1, 1]$.

A Simplified Version: Gated Recurrent Units (GRUs)

GRUs (Cho et al., 2014) can be viewed as simplified LSTMs from a practical perspective.

A Simplified Version: Gated Recurrent Units (GRUs)

GRUs (Cho et al., 2014) can be viewed as simplified LSTMs from a practical perspective.

$$\text{Update gate} \qquad \mathbf{z}_t = \sigma\left(\mathbf{W}_z\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_z\right)$$

A Simplified Version: Gated Recurrent Units (GRUs)

GRUs (Cho et al., 2014) can be viewed as simplified LSTMs from a practical perspective.

$$\begin{aligned} \text{Update gate} \qquad \mathbf{z}_t =& \sigma\left(\mathbf{W}_z\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_z\right) \\ \text{Reset gate} \qquad \mathbf{r}_t =& \sigma\left(\mathbf{W}_r\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_r\right) \end{aligned}$$

A Simplified Version: Gated Recurrent Units (GRUs)

GRUs (Cho et al., 2014) can be viewed as simplified LSTMs from a practical perspective.

$$
\begin{aligned}
\text{Update gate} \quad \mathbf{z}_t &= \sigma\left(\mathbf{W}_z\left[\mathbf{h}_{t-1};\mathbf{x}_t\right]+\mathbf{b}_z\right)\\
\text{Reset gate} \quad \mathbf{r}_t &= \sigma\left(\mathbf{W}_r\left[\mathbf{h}_{t-1};\mathbf{x}_t\right]+\mathbf{b}_r\right)\\
\text{Update rule} \quad \mathbf{h}_t &= (1-\mathbf{z}_t)\odot\mathbf{h}_{t-1}\\
&\quad + \mathbf{z}_t\odot\tanh\left(\mathbf{W}_h\left[\mathbf{r}_t\odot\mathbf{h}_{t-1};\mathbf{x}_t\right]+\mathbf{b}_h\right)
\end{aligned}
$$

A Simplified Version: Gated Recurrent Units (GRUs)

GRUs (Cho et al., 2014) can be viewed as simplified LSTMs from a practical perspective.

$$
\begin{aligned}
\text{Update gate} \quad \mathbf{z}_t &= \sigma\left(\mathbf{W}_z\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_z\right) \\
\text{Reset gate} \quad \mathbf{r}_t &= \sigma\left(\mathbf{W}_r\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_r\right) \\
\text{Update rule} \quad \mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} \\
&\quad + \mathbf{z}_t \odot \tanh\left(\mathbf{W}_h\left[\mathbf{r}_t \odot \mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_h\right)
\end{aligned}
$$

Works well with fewer parameters and less computation.

Theoretical Motivation vs. Practical Approaches

Even with LSTM and GRU architectures, RNNs usually require **gradient clipping** to stabilize training.

Theoretical Motivation vs. Practical Approaches

Even with LSTM and GRU architectures, RNNs usually require **gradient clipping** to stabilize training.

- If the L_2 norm exceeds a threshold, scale down the gradients before updating the parameters.

Theoretical Motivation vs. Practical Approaches

Even with LSTM and GRU architectures, RNNs usually require **gradient clipping** to stabilize training.

- If the L_2 norm exceeds a threshold, scale down the gradients before updating the parameters.

Even RNNs theoretically preserve information from the beginning of the sequence, in practice, they are not very good at it.

Khandelwal et al. (2018). Sharp Nearby, Fuzzy Far Away: How Neural Language Models Use Context.

Theoretical Motivation vs. Practical Approaches

Even with LSTM and GRU architectures, RNNs usually require **gradient clipping** to stabilize training.

- If the L_2 norm exceeds a threshold, scale down the gradients before updating the parameters.

Even RNNs theoretically preserve information from the beginning of the sequence, in practice, they are not very good at it.

Khandelwal et al. (2018). Sharp Nearby, Fuzzy Far Away: How Neural Language Models Use Context.

Bidirectional modeling typically gives more powerful features.

Theoretical Motivation vs. Practical Approaches

Even with LSTM and GRU architectures, RNNs usually require **gradient clipping** to stabilize training.

- If the L_2 norm exceeds a threshold, scale down the gradients before updating the parameters.

Even RNNs theoretically preserve information from the beginning of the sequence, in practice, they are not very good at it.

Khandelwal et al. (2018). Sharp Nearby, Fuzzy Far Away: How Neural Language Models Use Context.

Bidirectional modeling typically gives more powerful features.

To obtain the fixed-dimensional output as RNN features for classification, we may use the hidden states at the last time step, or pooling over all hidden states (Lin et al., 2017).

Pretrained RNNs

In earlier years, people pretrained RNNs on large corpora!

Peters et al. (2018). Deep contextualized word representations.

(Also known as ELMo; Embeddings from Language Models)

Pretrained RNNs

In earlier years, people pretrained RNNs on large corpora!

Peters et al. (2018). Deep contextualized word representations.

(Also known as ELMo; Embeddings from Language Models)

ELMo trains a bidirectional LSTM on a large corpus, and use the hidden states as text features.

Pretrained RNNs

In earlier years, people pretrained RNNs on large corpora!

Peters et al. (2018). Deep contextualized word representations.

(Also known as ELMo; Embeddings from Language Models)

ELMo trains a bidirectional LSTM on a large corpus, and use the hidden states as text features.

The hidden states are also referred to as **contextualized word embeddings**.

Recursive Neural Networks

Generalized RNNs that support tree-structured computation graph.

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}\right)$$

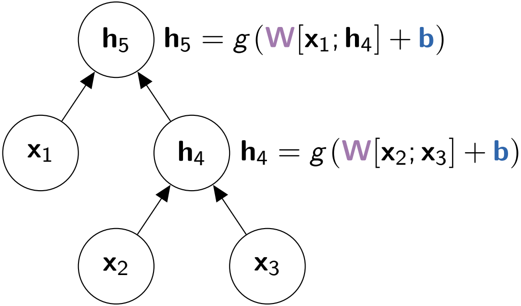
Run constituency parser on sentence and construct vector recursively (Socher et al., 2011 & 2013).

Recursive Neural Networks

Generalized RNNs that support tree-structured computation graph.

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$

Run constituency parser on sentence and construct vector recursively (Socher et al., 2011 & 2013).

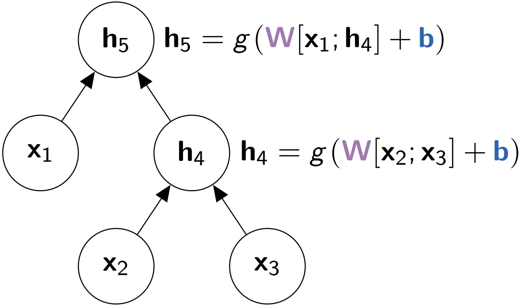


Recursive Neural Networks

Generalized RNNs that support tree-structured computation graph.

$$\mathbf{h}_t = g\left(\mathbf{W}\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}\right)$$

Run constituency parser on sentence and construct vector recursively (Socher et al., 2011 & 2013).



We may use complicated cells (e.g., LSTMs) to compute $\mathbf{h}_i$ (Zhu et al., 2015, Tai et al. 2015).

From LSTMs to Tree LSTMs

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_f\right)$$
$$\mathbf{i}_t = \sigma\left(\mathbf{W}_i\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_i\right)$$
$$\tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_c\right)$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$
$$\mathbf{o}_t = \sigma\left(\mathbf{W}_o\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_o\right)$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

$$\mathbf{l}_n = \sigma\left(\mathbf{W}_\ell[\mathbf{h}_\ell; \mathbf{h}_r] + \mathbf{b}_\ell\right)$$
$$\mathbf{r}_n = \sigma\left(\mathbf{W}_r[\mathbf{h}_\ell; \mathbf{h}_r] + \mathbf{b}_r\right)$$
$$\tilde{\mathbf{c}}_n = \tanh\left(\mathbf{W}_c[\mathbf{h}_l; \mathbf{h}_r] + \mathbf{b}_c\right)$$
$$\mathbf{c}_n = \mathbf{l}_n \odot \mathbf{c}_l + \mathbf{r}_n \odot \mathbf{c}_r + \tilde{\mathbf{c}}_n$$
$$\mathbf{o}_n = \sigma\left(\mathbf{W}_o[\mathbf{h}_l; \mathbf{h}_r] + \mathbf{b}_o\right)$$
$$\mathbf{h}_n = \mathbf{o}_n \odot \tanh(\mathbf{c}_n)$$

From LSTMs to Tree LSTMs

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_f\right)$$
$$\mathbf{i}_t = \sigma\left(\mathbf{W}_i\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_i\right)$$
$$\tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_c\right)$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$
$$\mathbf{o}_t = \sigma\left(\mathbf{W}_o\left[\mathbf{h}_{t-1};\mathbf{x}_t\right] + \mathbf{b}_o\right)$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

$$\mathbf{l}_n = \sigma\left(\mathbf{W}_\ell[\mathbf{h}_\ell;\mathbf{h}_r] + \mathbf{b}_\ell\right)$$
$$\mathbf{r}_n = \sigma\left(\mathbf{W}_r[\mathbf{h}_\ell;\mathbf{h}_r] + \mathbf{b}_r\right)$$
$$\tilde{\mathbf{c}}_n = \tanh\left(\mathbf{W}_c[\mathbf{h}_l;\mathbf{h}_r] + \mathbf{b}_c\right)$$
$$\mathbf{c}_n = \mathbf{l}_n \odot \mathbf{c}_l + \mathbf{r}_n \odot \mathbf{c}_r + \tilde{\mathbf{c}}_n$$
$$\mathbf{o}_n = \sigma\left(\mathbf{W}_o[\mathbf{h}_l;\mathbf{h}_r] + \mathbf{b}_o\right)$$
$$\mathbf{h}_n = \mathbf{o}_n \odot \tanh(\mathbf{c}_n)$$

Recursive networks with left-branching trees shares a lot in common with RNNs.

From LSTMs to Tree LSTMs

$$\begin{aligned}
\mathbf{f}_t &= \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1};\mathbf{x}_t\right]+\mathbf{b}_f\right)\\
\mathbf{i}_t &= \sigma\left(\mathbf{W}_i\left[\mathbf{h}_{t-1};\mathbf{x}_t\right]+\mathbf{b}_i\right)\\
\tilde{\mathbf{c}}_t &= \tanh\left(\mathbf{W}_c\left[\mathbf{h}_{t-1};\mathbf{x}_t\right]+\mathbf{b}_c\right)\\
\mathbf{c}_t &= \mathbf{f}_t\odot\mathbf{c}_{t-1}+\mathbf{i}_t\odot\tilde{\mathbf{c}}_t\\
\mathbf{o}_t &= \sigma\left(\mathbf{W}_o\left[\mathbf{h}_{t-1};\mathbf{x}_t\right]+\mathbf{b}_o\right)\\
\mathbf{h}_t &= \mathbf{o}_t\odot\tanh(\mathbf{c}_t)
\end{aligned}$$

$$\begin{aligned}
\mathbf{l}_n &= \sigma\left(\mathbf{W}_\ell[\mathbf{h}_\ell;\mathbf{h}_r]+\mathbf{b}_\ell\right)\\
\mathbf{r}_n &= \sigma\left(\mathbf{W}_r[\mathbf{h}_\ell;\mathbf{h}_r]+\mathbf{b}_r\right)\\
\tilde{\mathbf{c}}_n &= \tanh\left(\mathbf{W}_c[\mathbf{h}_l;\mathbf{h}_r]+\mathbf{b}_c\right)\\
\mathbf{c}_n &= \mathbf{l}_n\odot\mathbf{c}_l+\mathbf{r}_n\odot\mathbf{c}_r+\tilde{\mathbf{c}}_n\\
\mathbf{o}_n &= \sigma\left(\mathbf{W}_o[\mathbf{h}_l;\mathbf{h}_r]+\mathbf{b}_o\right)\\
\mathbf{h}_n &= \mathbf{o}_n\odot\tanh(\mathbf{c}_n)
\end{aligned}$$

Recursive networks with left-branching trees shares a lot in common with RNNs.

Syntactically meaningful parse trees are not necessary for good representations: instead, size balanced trees work well for most tasks (Shi et al., 2018).

RNNs and RvNNs as MLPs

All the gates in advanced RNN architectures are linear transformations followed by an activation function.

Taking LSTMs as an example,

Forget gate	$\mathbf{f}_t = \sigma\left(\mathbf{W}_f\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_f\right)$
Input gate	$\mathbf{i}_t = \sigma\left(\mathbf{W}_i\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_i\right)$
Cell	$\tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_c\right)$
Output gate	$\mathbf{o}_t = \sigma\left(\mathbf{W}_o\left[\mathbf{h}_{t-1}; \mathbf{x}_t\right] + \mathbf{b}_o\right)$
Update	$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$
Hidden state	$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$

Gates are combined linearly to form intermediate results.

Recap: Attention Pooling

Removing the third dimension of a tensor using a weighted average:

$$\texttt{attnpool}(\mathbf{O})_{i,j,k} = \sum_{p=1}^{c} \alpha_p \mathbf{O}_{i,j,p,k}$$

Recap: Attention Pooling

Removing the third dimension of a tensor using a weighted average:

$$\texttt{attnpool}(\mathbf{O})_{i,j,k} = \sum_{p=1}^{c} \alpha_p \mathbf{O}_{i,j,p,k}$$

How do we compute the weights α_p?

Recap: Attention Pooling

Removing the third dimension of a tensor using a weighted average:

$$\mathtt{attnpool}(\mathbf{O})_{i,j,k} = \sum_{p=1}^{c} \alpha_p \mathbf{O}_{i,j,p,k}$$

How do we compute the weights α_p?

$$\alpha_p = \mathtt{softmax}(\mathbf{s})_p = \frac{\exp(s_p)}{\sum_{q=1}^{c} \exp(s_q)}$$

$$s_p = \mathbf{w}^T \mathbf{v}_p$$

where $\mathbf{v}_p$ is the (stretched) vector of $\mathbf{O}_{*,*,p,*}$.

Recap: Attention Pooling

Removing the third dimension of a tensor using a weighted average:

$$\texttt{attnpool}(\mathbf{O})_{i,j,k} = \sum_{p=1}^{c} \alpha_p \mathbf{O}_{i,j,p,k}$$

How do we compute the weights α_p?

$$\alpha_p = \texttt{softmax}(\mathbf{s})_p = \frac{\exp(s_p)}{\sum_{q=1}^{c} \exp(s_q)}$$

$$s_p = \mathbf{w}^T \mathbf{v}_p$$

where $\mathbf{v}_p$ is the (stretched) vector of $\mathbf{O}_{*,*,p,*}$.

We may calculate $\mathbf{s}$ with more complicated neural architectures.

Recap: Attention Pooling

Removing the third dimension of a tensor using a weighted average:

$$\mathtt{attnpool}(\mathbf{O})_{i,j,k} = \sum_{p=1}^{c} \alpha_p \mathbf{O}_{i,j,p,k}$$

How do we compute the weights α_p?

$$\alpha_p = \mathtt{softmax}(\mathbf{s})_p = \frac{\exp(s_p)}{\sum_{q=1}^{c} \exp(s_q)}$$
$$s_p = \mathbf{w}^T \mathbf{v}_p$$

where $\mathbf{v}_p$ is the (stretched) vector of $\mathbf{O}_{*,*,p,*}$.

We may calculate $\mathbf{s}$ with more complicated neural architectures.

In (most) machine learning context, attention is just weighted sum!

The Transformer Architecture

Vaswani et al. (2017). Attention is All You Need.

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

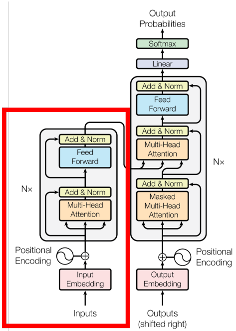
Aidan N. Gomez[*] [†]
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser[*]
Google Brain
lukaszkaiser@google.com

Illia Polosukhin[*] [‡]
illia.polosukhin@gmail.com

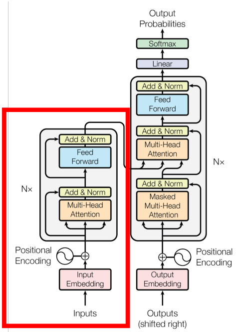
Transformers

- Introduced for sequence-to-sequence tasks, but could be more accessible understood as a feature extractor.



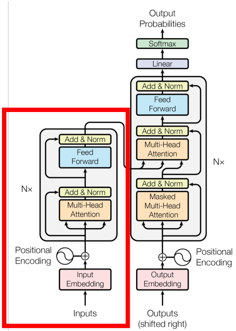
Transformers

- Introduced for sequence-to-sequence tasks, but could be more accessible understood as a feature extractor.
- **Key idea**: every token has **attention** to every other token.
  In slightly more CS/math words, after passing through one transformer layer, the representation of one token should contain information from all context tokens.



Transformers

- Introduced for sequence-to-sequence tasks, but could be more accessible understood as a feature extractor.
- **Key idea**: every token has **attention** to every other token.
  In slightly more CS/math words, after passing through one transformer layer, the representation of one token should contain information from all context tokens.



For sentence with tokens $w_i, \ldots, w_n$, a transformer computes

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

Transformer Encoder

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

In database terms, **K**, **Q**, and **V** are motivated by the functions of **key**, **query**, and **value**, respectively.

Transformer Encoder

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

In database terms, **K**, **Q**, and **V** are motivated by the functions of **key**, **query**, and **value**, respectively.

The next layer representations are given by

$$\tilde{\mathbf{E}} = \mathbf{V}\texttt{softmax}\left(\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{d_2}}\right) \in \mathbb{R}^{d_3 \times n}$$

Transformer Encoder: Variance Normalization

$$\tilde{\mathbf{E}} = \mathbf{V}\texttt{softmax}\left(\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{d_2}}\right) \in \mathbb{R}^{d_3 \times n}$$

What is $\sqrt{d_2}$ for?

Transformer Encoder: Variance Normalization

$$\tilde{\mathbf{E}} = \mathbf{V}\texttt{softmax}\left(\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{d_2}}\right) \in \mathbb{R}^{d_3 \times n}$$

What is $\sqrt{d_2}$ for?

Consider the dot product between vectors **a** and **b**: if each entry in both vector is drawn from a distribution with zero mean and unit variance, what happens if the dimensionality grows?

Transformer Encoder: Variance Normalization

$$\tilde{\mathbf{E}} = \mathbf{V}\texttt{softmax}\left(\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{d_2}}\right) \in \mathbb{R}^{d_3 \times n}$$

What is $\sqrt{d_2}$ for?

Consider the dot product between vectors **a** and **b**: if each entry in both vector is drawn from a distribution with zero mean and unit variance, what happens if the dimensionality grows?

The variance of the dot product grows **linearly** with the dimensionality.

Transformer Encoder: Variance Normalization

$$\tilde{\mathbf{E}} = \mathbf{V}\texttt{softmax}\left(\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{d_2}}\right) \in \mathbb{R}^{d_3 \times n}$$

What is $\sqrt{d_2}$ for?

Consider the dot product between vectors **a** and **b**: if each entry in both vector is drawn from a distribution with zero mean and unit variance, what happens if the dimensionality grows?

The variance of the dot product grows **linearly** with the dimensionality.

Recall:

$$\texttt{softmax}([1, -1]) = [0.88, 0.12]$$

$$\texttt{softmax}([10, -10]) \approx [1, 2.0612 \times 10^{-9}]$$

Transformer Encoder: Variance Normalization

$$\tilde{\mathbf{E}} = \mathbf{V}\texttt{softmax}\left(\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{d_2}}\right) \in \mathbb{R}^{d_3 \times n}$$

What is $\sqrt{d_2}$ for?

Consider the dot product between vectors **a** and **b**: if each entry in both vector is drawn from a distribution with zero mean and unit variance, what happens if the dimensionality grows?

The variance of the dot product grows **linearly** with the dimensionality.

Recall:

$$\texttt{softmax}\left([1, -1]\right) = [0.88, 0.12]$$

$$\texttt{softmax}\left([10, -10]\right) \approx [1, 2.0612 \times 10^{-9}]$$

The scaling factor $\sqrt{d_2}$ stabilizes the variance of the dot product. See also Xavier initialization (Glorot & Bengio, 2010).

Position Encoding

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

This formulation isn't so different from weighted bag of words.

Position Encoding

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

This formulation isn't so different from weighted bag of words.

Key idea: add position encoding $\mathbf{p}$ to the input embeddings.

Position Encoding

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

This formulation isn't so different from weighted bag of words.

Key idea: add position encoding $\mathbf{p}$ to the input embeddings.

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d_1}}\right) \qquad p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d_1}}\right)$$

Position Encoding

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

This formulation isn't so different from weighted bag of words.

Key idea: add position encoding **p** to the input embeddings.

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d_1}}\right) \qquad p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d_1}}\right)$$

Despite the arbitrary choice of the constant 10,000, the theoretical motivation is to make the add-δ relation in position encoding representable by a linear transformation.

$$\forall \delta \in \mathbb{N}_+, \exists \mathbf{M}_\delta s.t. \forall i, \mathbf{p}_{i+\delta} = \mathbf{M}_\delta \mathbf{p}_i$$

Position Encoding

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k \mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q \mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v \mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

This formulation isn't so different from weighted bag of words.

Key idea: add position encoding $\mathbf{p}$ to the input embeddings.

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d_1}}\right) \qquad p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d_1}}\right)$$

Despite the arbitrary choice of the constant 10,000, the theoretical motivation is to make the add-δ relation in position encoding representable by a linear transformation.

$$\forall \delta \in \mathbb{N}_+, \exists \mathbf{M}_\delta s.t. \forall i, \mathbf{p}_{i+\delta} = \mathbf{M}_\delta \mathbf{p}_i$$

Now: learnable position encoding (Shaw et al., 2018, inter alia).

Multi-Head Attention

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

The equation above is called one **head** of attention.

Multi-Head Attention

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

The equation above is called one **head** of attention.

To capture different aspects of the input, we concatenate multiple heads to form the feature.

Remember these heads should be **initialized differently**.

Stacking Multiple Layers of Transformers

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

$$\tilde{\mathbf{E}} = \mathbf{V}\texttt{softmax}\left(\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{d_2}}\right) \in \mathbb{R}^{d_3 \times n}$$

Stacking Multiple Layers of Transformers

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

$$\mathbf{K} = \mathbf{W}_k\mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{K} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{Q} = \mathbf{W}_q\mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1} \qquad \mathbf{Q} \in \mathbb{R}^{d_2 \times n}$$

$$\mathbf{V} = \mathbf{W}_v\mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1} \qquad \mathbf{V} \in \mathbb{R}^{d_3 \times n}$$

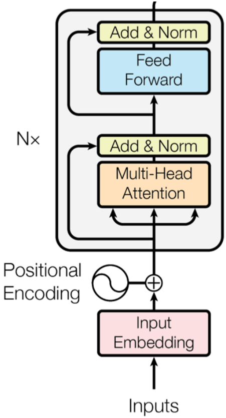
$$\tilde{\mathbf{E}} = \mathbf{V}\texttt{softmax}\left(\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{d_2}}\right) \in \mathbb{R}^{d_3 \times n}$$

The output of one transformer layer $\tilde{\mathbf{E}}$ is fed into the next layer as the input $\mathbf{E}$.

The Residual Connections in Transformers

After processing in each transformer component, the output is added to the input.

$$\mathbf{x} \leftarrow \mathbf{x} + \mathcal{F}(\mathbf{x})$$

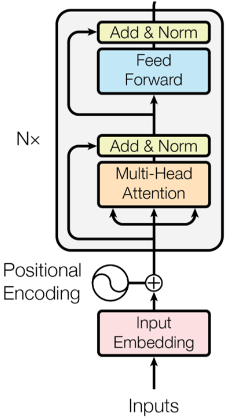


The Residual Connections in Transformers

After processing in each transformer component, the output is added to the input.

$$\mathbf{x} \leftarrow \mathbf{x} + \mathcal{F}(\mathbf{x})$$

Residual connection (He et al., 2016).
Designed for easier training in computer vision: there is always a component for the input to linearly contribute to the output.



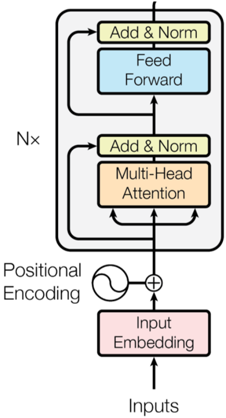
The Residual Connections in Transformers

After processing in each transformer component, the output is added to the input.

$$\mathbf{x} \leftarrow \mathbf{x} + \mathcal{F}(\mathbf{x})$$

Residual connection (He et al., 2016). Designed for easier training in computer vision: there is always a component for the input to linearly contribute to the output.

Another interpretation: the residual connection is the main flow of information, while other results are added to the main flow.



Transformers as MLPs

$$\mathbf{E} = \texttt{Embedding}(w_i, \ldots, w_n) \in \mathbb{R}^{d_1 \times n}$$

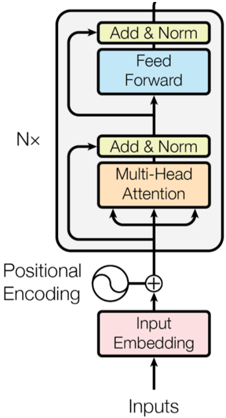
$$\mathbf{K} = \mathbf{W}_k \mathbf{E} \qquad \mathbf{W_k} \in \mathbb{R}^{d_2 \times d_1}$$

$$\mathbf{Q} = \mathbf{W}_q \mathbf{E} \qquad \mathbf{W_q} \in \mathbb{R}^{d_2 \times d_1}$$

$$\mathbf{V} = \mathbf{W}_v \mathbf{E} \qquad \mathbf{W_v} \in \mathbb{R}^{d_3 \times d_1}$$

$$\tilde{\mathbf{E}} = \mathbf{V}\texttt{softmax}\left(\frac{\mathbf{K}^T\mathbf{Q}}{\sqrt{d_2}}\right) \in \mathbb{R}^{d_3 \times n}$$

All above are generalized linear operations, coupled with a some real MLP in each Transformer layer.



Caveat: Attention is Not Explanation

Suppose $\mathbf{H} \in \mathbb{R}^{n \times d}$ is the final hidden state of a transformer.

If we calculate attention weights α on $\mathbf{H}$, does it mean that the model is attending to the corresponding tokens?

Caveat: Attention is Not Explanation

Suppose $\mathbf{H} \in \mathbb{R}^{n \times d}$ is the final hidden state of a transformer.

If we calculate attention weights α on $\mathbf{H}$, does it mean that the model is attending to the corresponding tokens?

Not really. Token representation $\mathbf{h}_i$ is affected by all other tokens.

Caveat: Attention is Not Explanation

Suppose $\mathbf{H} \in \mathbb{R}^{n \times d}$ is the final hidden state of a transformer.

If we calculate attention weights α on $\mathbf{H}$, does it mean that the model is attending to the corresponding tokens?

Not really. Token representation $\mathbf{h}_i$ is affected by all other tokens.

Are there better way to extract the positions that the model "thinks" are important?

Caveat: Attention is Not Explanation

Suppose $\mathbf{H} \in \mathbb{R}^{n \times d}$ is the final hidden state of a transformer.

If we calculate attention weights α on $\mathbf{H}$, does it mean that the model is attending to the corresponding tokens?

Not really. Token representation $\mathbf{h}_i$ is affected by all other tokens.

Are there better way to extract the positions that the model "thinks" are important? Yes!

We have $\text{loss}(\mathbf{x}, y, \widehat{y}; \mathbf{\Theta})$

For optimizing the model, we compute $\frac{\partial \text{loss}}{\partial \mathbf{\Theta}}$

Caveat: Attention is Not Explanation

Suppose $\mathbf{H} \in \mathbb{R}^{n \times d}$ is the final hidden state of a transformer.

If we calculate attention weights α on $\mathbf{H}$, does it mean that the model is attending to the corresponding tokens?

Not really. Token representation $\mathbf{h}_i$ is affected by all other tokens.

Are there better way to extract the positions that the model "thinks" are important? Yes!

We have $\text{loss}(\mathbf{x}, y, \widehat{y}; \mathbf{\Theta})$

For optimizing the model, we compute $\dfrac{\partial \text{loss}}{\partial \mathbf{\Theta}}$

For input-based explanation, compute $\dfrac{\partial \text{loss}}{\partial \mathbf{x}}$

Simonyan et al. 2013. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps

What's Not Covered (Much) in the Lecture

- Initialization and normalization techniques for stabilizing training.
  `https://pytorch.org/docs/stable/nn.init.html`
  Ba et al. (2016). Layer Normalization.

What's Not Covered (Much) in the Lecture

- Initialization and normalization techniques for stabilizing training.
  `https://pytorch.org/docs/stable/nn.init.html`
  Ba et al. (2016). Layer Normalization.
- Dropout (Srivastava et al., 2014): a simple regularization technique that randomly sets some the input units (of a neural layer) to zero at each update during training.
  Remember to turn off dropout during evaluation!

Next

Language Modeling