Lectures on Natural Language Processing

5. Neural Architectures for Language Processing

Karl Stratos

Review: Deep Learning and Universality

- A system that employs a hierarchy of features of the input, learned end-to-end jointly with the predictor.

$$f(x; \theta_1, \theta_2, \ldots, \theta_L) = F_L(F_{L-1}(\cdots F_2(F_1(x; \theta_1); \theta_2) \cdots); \theta_L)$$

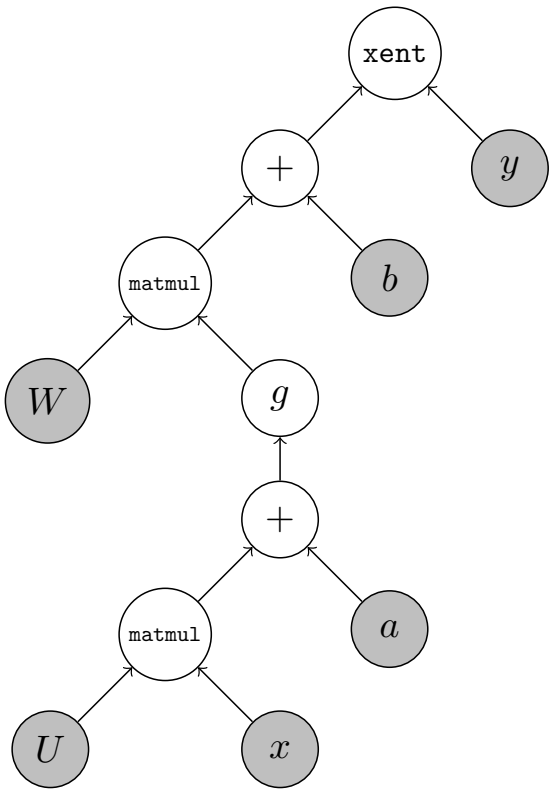
- We will refer to F_k as **layer** k
- E.g., deep learning for classification:

$$f_c(x; \mathbf{w}, \mathbf{b}, \theta_1, \theta_2, \ldots, \theta_L) = \mathbf{w}_c \cdot f(x; \theta_1, \theta_2, \ldots, \theta_L) + b_c$$

  - All parameters $(\mathbf{w}, \mathbf{b}, \theta_1, \theta_2, \ldots, \theta_L)$ are learned jointly
  - We can think of $f(x; \theta_1, \theta_2, \ldots, \theta_L)$ as **learned features** for x or a **learned representation** of x (doesn't depend on the class being scored)
- **Universality.** Feedforward with a nonlinear layer can express any mapping (given enough hidden units).

Review: Computation Graph

Express any differentiable function as a directed acyclic graph (DAG) and automatically calculate gradients for all nodes.



- **Forward.** Populate values in topological order.
- **Backward.** Populate gradients in reverse topological order by the chain rule.

$$z^i = \sum_{j \in \mathbf{ch}(i)} z^j \times \underbrace{\frac{\partial f^j(x_I^j)}{\partial x^i}}_{\text{Jacobian of } f^j \text{ wrt. } x^i} \Bigg|_{x_I^j = a_I^j}$$

Use the stored gradients to update parameters.

Training Tips

- Regularization: Dropout, label smoothing, layer normalization
  - In addition to the usual early stopping based on validation performance
- Initialization: Uniform, normal, Xavier, Kaiming, and others
- Optimization: Appropriate learning rates, gradients with momentum, gradient clipping
- Ensembling: Average many stochastically trained neural models for variance reduction and improved generalization.
- More tricks:
  - **Gradient accumulation**: Make batch size G times larger without using more memory by accumulating gradients over G batches before updating weights.
  - **Residual connection**: Use $\mathbf{enc}_\theta(x) + x$ to propagate gradient directly to x, useful with deep networks (hidden dim must equal input dim).

Review: Need for Specialized Neural Architectures

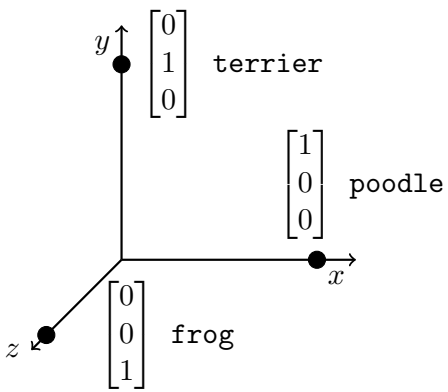
- Feedforward implicitly assumes the input is a single vector.
- NLP: Input is a *sequence*!
- Option 1: BOW representation
  - Loses lots of information (e.g., ordering), high-dimensional
- Option 2: Giant feedforward with input dimension = max sequence length
  - Computationally intractable, too many parameters to learn
- Solution: Develop specialized architectures that can handle *variable* input lengths.
- Starting point: **word embeddings**

Word Embeddings: Idea

Represent word type $x \in \mathcal{V}$ as a continuous vector $e_x \in \mathbb{R}^{d_w}$

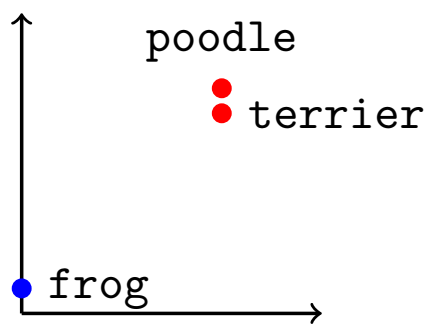
- The loss will be a differentiable function of e_x.
- As we optimize the loss, the word embeddings tend to implicitly capture inter-word relations

(Word identity indicator vector)



$$||\texttt{poodle} - \texttt{terrier}||$$
$$= ||\texttt{terrier} - \texttt{frog}||$$
$$= ||\texttt{frog} - \texttt{poodle}||$$

(Continuous word representations after learning)



$$||\texttt{poodle} - \texttt{terrier}||$$
$$\ll ||\texttt{poodle} - \texttt{frog}||$$

Embedding Matrix in Practice

- Part of model parameter θ to learn (aka. "lookup table")
- $E \in \mathbb{R}^{d_w \times V}$ where $V = |\mathcal{V}|$ is the vocabulary size and d_w is the word embedding dimension (e.g., 128, 256, 512)
  - More generally can embed any discrete **features**
  - Example: n-grams, special indicators (language, beginning/end of a span, etc.)
- Can view as a "lookup" function $E : \mathcal{V}^T \to \mathbb{R}^{T \times d_w}$

$$E([\text{the}, \text{dog}, \text{laughed}]) = E\left(\begin{bmatrix} 0 \\ 3 \\ 2 \end{bmatrix}\right) = \begin{bmatrix} 0.7 & -0.1 & 0.3 & 0.1 \\ 0.2 & 0.1 & 0.1 & 0.7 \\ 0.0 & 0.8 & -0.4 & 0.6 \end{bmatrix}$$

- Very sparse updates: only a few columns of E updated on a single batch
- Special **padding token**: $E(\texttt{<pad>}) = (0, \ldots, 0)$

Example Architecture: Continuous Bag-of-Words (CBOW)

- $\mathbf{enc}_\theta : \mathcal{V}^+ \to \mathbb{R}^{d_w}$ defined by

$$\mathbf{enc}_\theta(x_1 \dots x_T) = \frac{1}{T} \sum_{t=1}^{T} E(x_t)$$

- Optionally apply additional layers (e.g., $\mathbf{enc}_\theta(x_1 \dots x_T) = \tanh(W \frac{1}{T} \sum_{t=1}^{T} E(x_t) + b)$).
- Differentiable in E and can be fed into a linear classifier to learn end-to-end
- **Pros.** Simple, natural continuous extension of bag-of-words (BOW) representation (= feedforward on count-based BOW)
- **Cons.** Like BOW, CBOW is incapable of modeling word ordering.
  - Maybe try to use n-grams for $n > 1$?

Convolutional Layer

- Idea: Slide an n-gram filter (aka. "kernel") across text to identify a certain aspect from local patterns
- Example: Trigram filter F_3 "activates" at negative sentiment
  - **[the movie was]** not super good $\mapsto -0.2$
  - the **[movie was not]** super good $\mapsto -0.1$
  - the movie **[was not super]** good $\mapsto 0.3$
  - the movie was **[not super good]** $\mapsto 1.8$

  Take 1.8 is the final output of F_3 on the sentence (large value means the filter is activated).
- Sliding can be implemented efficiently in parallel.
- Learnable parameter: $U \in \mathbb{R}^{3 \times d_w}$ defining $F_3 : \mathcal{V}^+ \to \mathbb{R}$ by

$$F_3(x_1 \dots x_T) = \max_{t=1}^{T-2} g\left(\sum_{i,j}[\underbrace{U}_{3\times d_w} \underbrace{\odot}_{\text{elt-wise multi.}} \underbrace{E(x_t, x_{t+1}, x_{t+2})}_{3\times d_w}]_{i,j}\right)$$

g is a nonlinear function (e.g., ReLU). U reused for all inputs

Convolutional Neural Networks (CNNs)

- Learn H "types" of trigram filter $F_3 = (F_3^{(1)} \dots F_3^{(H)})$
- Each type expected to learn different aspects
  - Depends on the learning problem (e.g., positive and negative sentiments for sentiment classification)
  - Certain key phrases activate certain filters ("is good .")
- Treated as a H-dimensional **encoder** $F_3 : \mathcal{V}^+ \to \mathbb{R}^H$.
- General CNNs: Multiple n-grams (e.g., $n \in \{3, 4, 5\}$) and concatenate outputs

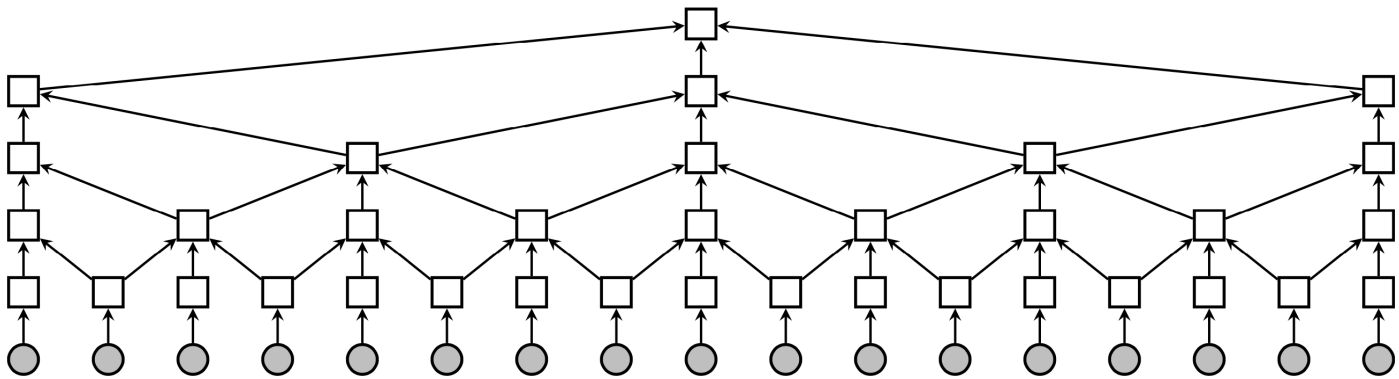
$$\textbf{CNN}_{n\in\{3,4,5\},H}(x_1 \dots x_T) = \begin{bmatrix} F_3(x_1 \dots x_T) \\ F_4(x_1 \dots x_T) \\ F_5(x_1 \dots x_T) \end{bmatrix} \in \mathbb{R}^{3H}$$

At the end of the day, it's just a text encoder with output dimension (number of n-gram types) $\times H$

Variations of CNNs

- **Stacking**: multiple convolutional layers stacked on each other
- **Stride**: Skip ahead when sliding (previously stride 1), reduces output dim
- **Mean pooling**: Instead of taking max activation ("max pooling"), average all activations?

Stride > 1 + stacking: learn progressively "higher-level" patterns

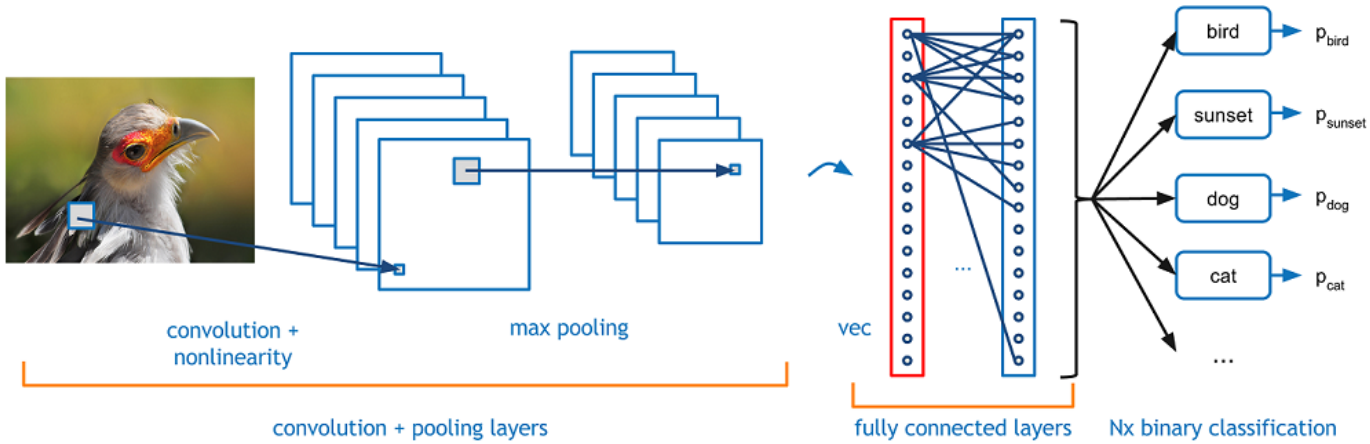


(Image credit: Eisenstein)

"Dilated" CNN with 3 conv layers, stride 2

Computer Vision and CNNs

- CNNs originated from image processing.



- Motivation: translation invariance (can have bird appear anywhere in the image)
- Applicable to NLP by treating text as 1-dimensional image (Kim, 2014)
  - Same motivation: can have "not good" appear anywhere
- **Cons.** Still cannot model word ordering beyond filter sizes.

Recurrent Neural Networks (RNNs)

- Idea: Read text $x_1 \dots x_T$ (already word embeddings for notational convenience) left-to-right, updating internal state

$$\begin{aligned} h_0 &= (0, \dots, 0) \\ h_1 &= \mathbf{RNN}_\theta(h_0, x_1) \\ &\vdots \\ h_T &= \mathbf{RNN}_\theta(h_{T-1}, x_T) \end{aligned}$$

- $\mathbf{RNN}_\theta : \mathbb{R}^{d_h} \times \mathbb{R}^{d_w} \to \mathbb{R}^{d_h}$ is a feedforward ("RNN cell"), e.g.,

$$\mathbf{RNN}_\theta(h_{t-1}, x_t) = \tanh\left(W x_t + V h_{t-1} + b\right)$$

  Important: parameters (W, V, b here) reused for all time steps

- $h_t \in \mathbb{R}^{d_h}$: function of $x_1 \dots x_t$

$$h_t = \mathbf{RNN}_\theta(\cdots \mathbf{RNN}_\theta(\mathbf{RNN}_\theta(0_d, x_1), x_2), \cdots, x_t)$$

Stacked RNNs

- Number of RNN layers L
- Stacked RNN cell $\mathbf{RNN}_\theta : \mathbb{R}^{Ld_h} \times \mathbb{R}^{d_w} \to \mathbb{R}^{Ld_h}$

$$\mathbf{RNN}_\theta(x_t, h_{t-1} = (h_{t-1}^{(1)} \dots h_{t-1}^{(L)})) = (h_t^{(1)} \dots h_t^{(L)})$$

consisting of L RNN cells (typically each with its own parameters)

$$\begin{aligned}
h_t^{(1)} &= \mathbf{RNN}_\theta^{(1)}(h_{t-1}^{(1)}, x_t) & \mathbf{RNN}_\theta^{(1)} &: \mathbb{R}^{d_h} \times \mathbb{R}^{d_w} \to \mathbb{R}^{d_h} \\
h_t^{(2)} &= \mathbf{RNN}_\theta^{(2)}(h_{t-1}^{(2)}, h_t^{(1)}) & \mathbf{RNN}_\theta^{(2)} &: \mathbb{R}^{d_h} \times \mathbb{R}^{d_h} \to \mathbb{R}^{d_h} \\
&\vdots \\
h_t^{(L)} &= \mathbf{RNN}_\theta^{(L)}(h_{t-1}^{(L)}, h_t^{(L-1)}) & \mathbf{RNN}_\theta^{(L)} &: \mathbb{R}^{d_h} \times \mathbb{R}^{d_h} \to \mathbb{R}^{d_h}
\end{aligned}$$

- Maps $x_1 \dots x_T$ to $h_1 \dots h_T \in \mathbb{R}^{Ld_h}$
  - Can get a single vector by averaging the states in the final layer $h_1^{(L)} \dots h_T^{(L)} \in \mathbb{R}^{d_h}$

Exploding/Vanishing Gradient Problems

- In general, deep networks with multiplicative weights get gradients in exponential form

$$\left|\frac{\partial w^n x}{\partial x}\right| = |w^n| \approx \begin{cases} 0 & \text{if } |w| < 1 \text{ ("vanishes")} \\ \infty & \text{if } |w| > 1 \text{ ("explodes")} \end{cases}$$

- RNN: if $x_1 \dots x_T$ is long, gradient of $\mathrm{Loss}(h_T)$ wrt. variables with small t will either vanish or explode.
- Solutions
  1. Gradient clipping
  2. Architectural modifications: maintain an extra "cell" state c_t that we carry over without losing signals (e.g., LSTM)

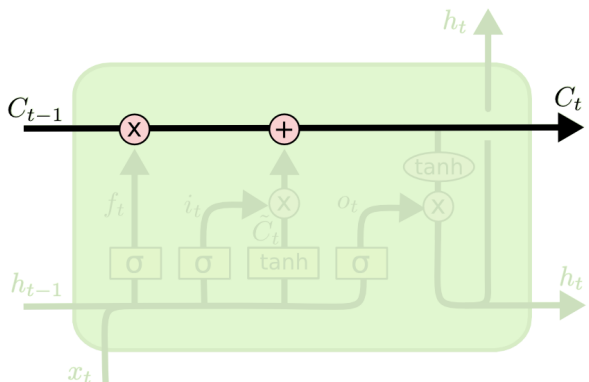
  Typically best to do both 1 and 2

Long Short-Term Memory (LSTM) Cell

- Parameters (omitting bias terms) $U^q, U^c, U^o \in \mathbb{R}^{d_h \times d_w}$, $V^q, V^c, V^o, W^q, W^o \in \mathbb{R}^{d_h \times d_h}$

$$
\begin{aligned}
q_t &= \sigma\left(U^q x_t + V^q h_{t-1} + W^q c_{t-1}\right) \\
c_t &= (1 - q_t) \odot c_{t-1} + q_t \odot \tanh\left(U^c x_t + V^c h_{t-1}\right) \\
o_t &= \sigma\left(U^o x_t + V^o h_{t-1} + W^o c_t\right) \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

- Idea: Memory cells c_t can carry long-range information (image credit: Colah's blog)

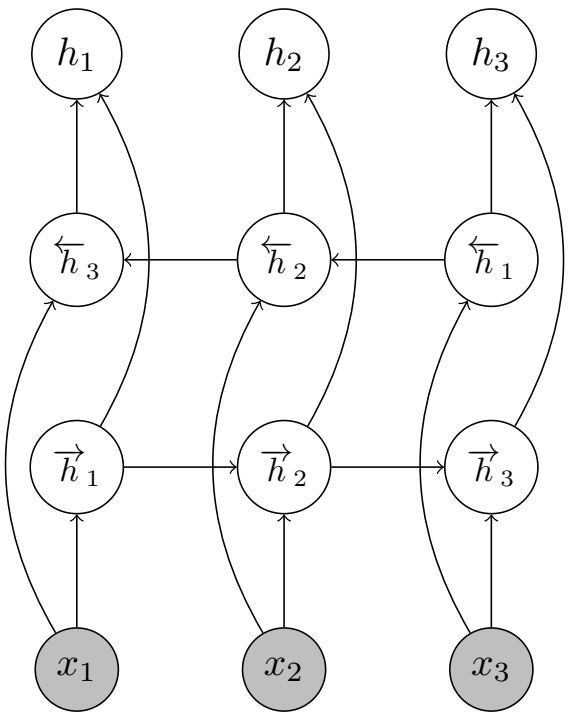


Network can choose to make $q_t = 0$!

- Cell states typically ignored: loss still defined using h_t

Bidirectional RNNs

Left-to-right RNN: h_t is a function of $x_{t'}$ for $t' \leq t$ only
Bidirectional RNN: Make h_t a function of all $x_1 \dots x_T$ as follows.



- Left-to-right RNN:
  $\overrightarrow{\textbf{RNN}}_\theta(x_1 \dots x_T) = \overrightarrow{h}_1 \dots \overrightarrow{h}_T$
- Right-to-left RNN:
  $\overleftarrow{\textbf{RNN}}_\theta(x_T \dots x_1) = \overleftarrow{h}_T \dots \overleftarrow{h}_1$
- New representation of x_t

$$h_t = (\overrightarrow{h}_t, \overleftarrow{h}_{T-t+1}) \in \mathbb{R}^{2d_h}$$

- Both RNNs learned jointly to optimize some loss (in h_t).

To get single vector, can average $h_1 \dots h_T \in \mathbb{R}^{2d_h}$.

Problems with Recurrent Architectures

Recurrent architectures are intuitive and effective.

- Compact recurrent cell updating an internal state is similar to how humans read text.
- Sensitive to word ordering
- Performs well, especially with LSTM cells and bidirectional variants

BUT

- **Slow.** Computation of $h_1 \dots h_T$ cannot be parallelized
  - Must compute h_{t-1} first before computing h_t.
- **Shallow bidirectionality.** Even if bidirectional, only a function of one side until the end of RNN computation

Attention

- Key recent progress in deep learning
  - Origin: NLP, specifically for sequence-to-sequence learning
- Idea: Let the model select which input vectors to use, by defining a distribution over them
- Three types of vector
  - **Query vector**: $q \in \mathbb{R}^d$
  - **Key/value vectors**: $(k_1, v_1) \dots (k_T, v_T) \in \mathbb{R}^d \times \mathbb{R}^d$
- Sometimes key/value collectively called **memory bank** M
- Compute an embedding from M "attended" by q

$$(p_1 \dots p_T) = \text{softmax}(q^\top k_1, \dots, q^\top k_T)$$

$$\textbf{Attn}(q, M) := \sum_{t=1}^{T} p_t v_t \in \mathbb{R}^d$$

Attention in Matrix Form

Input

- $Q = (q_1 \dots q_{T'}) \in \mathbb{R}^{T' \times d}$: T' query vectors as rows
- $K = (k_1 \dots k_T) \in \mathbb{R}^{T \times d}$: T key vectors as rows
- $V = (v_1 \dots v_T) \in \mathbb{R}^{T \times d}$: T value vectors as rows

Output

- $A = (a_1 \dots a_{T'}) \in \mathbb{R}^{T' \times d}$: $a_t = \textbf{Attn}(q_t, M = (K, V))$

Compute $A = \textbf{Attn}(Q, K, V)$ efficiently in matrix form:

$$\underbrace{A}_{T' \times d} = \underbrace{\text{softmax}}_{\text{over each row}}\left(\underbrace{Q}_{T' \times d}\underbrace{K^\top}_{d \times T}\right)\underbrace{V}_{T \times d}$$

Multi-Head Attention

- Idea: Make H types of attention ("heads")
  - May learn different attention behaviors.

$$(\mathbf{0.7}, 0.1, 0.1, 0.0) \qquad \text{(head 1)}$$
$$(0.0, 0.1, \mathbf{0.5}, 0.4) \qquad \text{(head 2)}$$

- Unlike raw attention, there are learnable parameters.
  1. For each type $\tau \in \{q, k, v\}$: Linear function $f_\theta^{(\tau,i)} : \mathbb{R}^d \to \mathbb{R}^{d/H}$ for $i = 1 \dots H$ (assume d is divisible by H)
  2. Linear function $g_\theta : \mathbb{R}^d \to \mathbb{R}^d$
- Given $Q \in \mathbb{R}^{T' \times d}$ and $K, V \in \mathbb{R}^{T \times d}$, compute

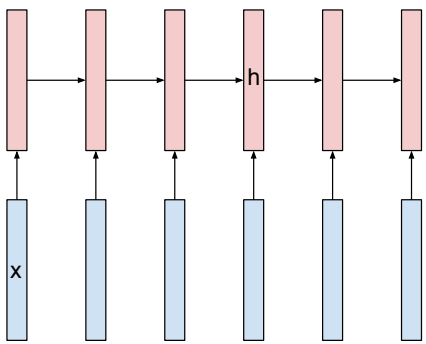
$$\underbrace{\mathbf{Attn}_\theta^H(Q,K,V)}_{d} = g_\theta\Big(\underbrace{\bigoplus_{i=1}^{H}}_{\text{concat}} \mathbf{Attn}\big(\underbrace{f_\theta^{(q,i)}(Q)}_{d/H}, \underbrace{f_\theta^{(k,i)}(K)}_{d/H}, \underbrace{f_\theta^{(v,i)}(V)}_{d/H}\big)\Big)$$

Example Architecture: Self-Attention Encoder

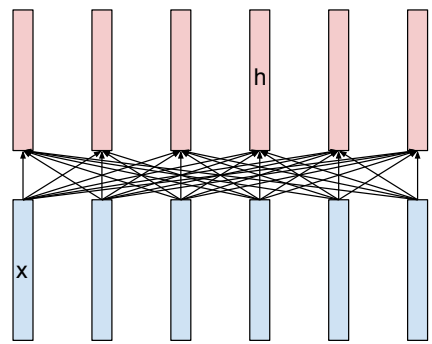
- Initial word embeddings $X = (x_1 \dots x_T) \in \mathbb{R}^{T \times d}$
- Want new embeddings $Z = (h_1 \dots h_T) \in \mathbb{R}^{T \times d}$ such that h_t is a function of all $x_1 \dots x_T$
- **Self-attention**: Use X as key, query, value at the same time!

$$Z = \textbf{Attn}_\theta^H(X, X, X)$$

- Multi-head attention parameters will "specialize" X internally.
- Unlike recurrent, h_t has direct connection to every input.



recurrent



self-attention

Transformer Encoder (Vaswani et al., 2017)

- Do self-attention many times, each time with layer normalization, nonlinear transformation, and residual connection
- There is no "right" implementation, other than the general importance of multiple applications of the above
- Example: Given $X = Z_0 \in \mathbb{R}^{T \times d}$, for layer $l = 1 \dots 6$, compute using layer-specific parameters

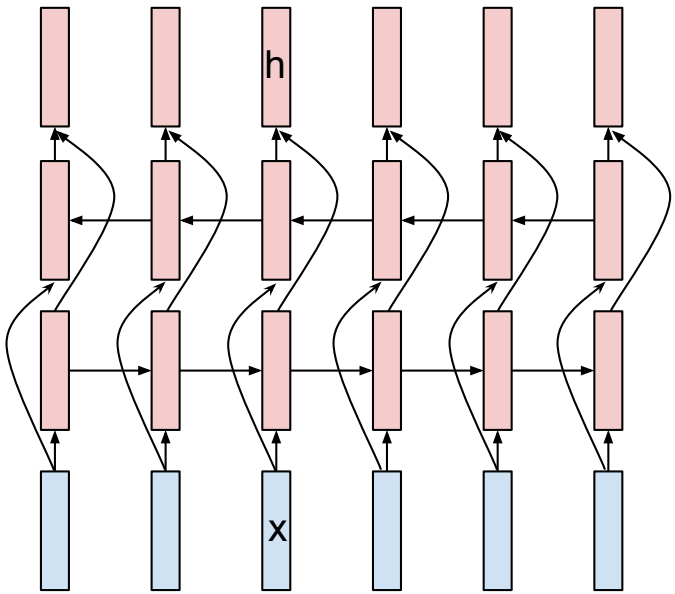
$$
\begin{aligned}
N_{l-1} &= \textbf{LayerNormalization}_\theta(Z_{l-1}) \\
H_l &= \textbf{Attn}^H_\theta(N_{l-1}, N_{l-1}, N_{l-1}) \\
\widehat{Z}_l &= \textbf{Dropout}_{0.1}(H_l) + Z_{l-1} \\
Z_l &= \textbf{NonlinearTransformation}(\widehat{Z}_l)
\end{aligned}
$$

(typically with large hidden dimension $D > d$)

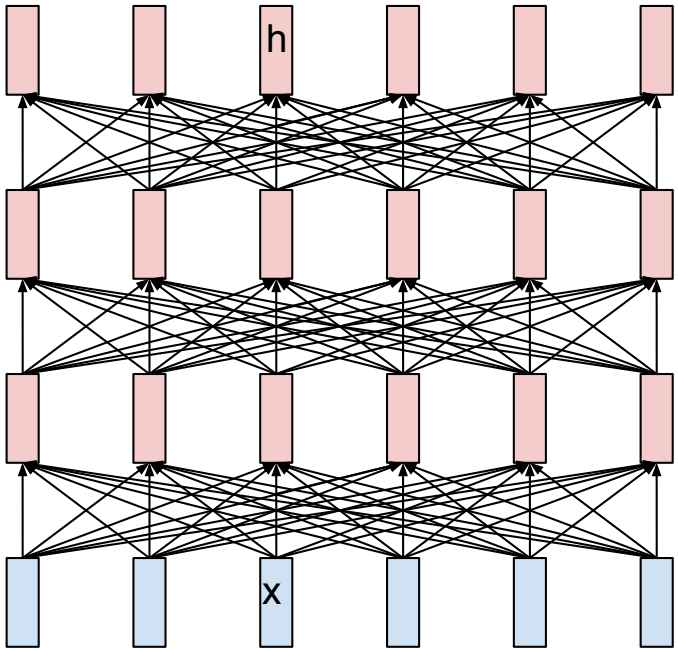
Use $Z_6 \in \mathbb{R}^{T \times d}$ as final d-dimensional embeddings of the input tokens.

- To get a single vector, can either average or take the first row.

Bidirectional RNNs vs Transformers



not bidirectional until later

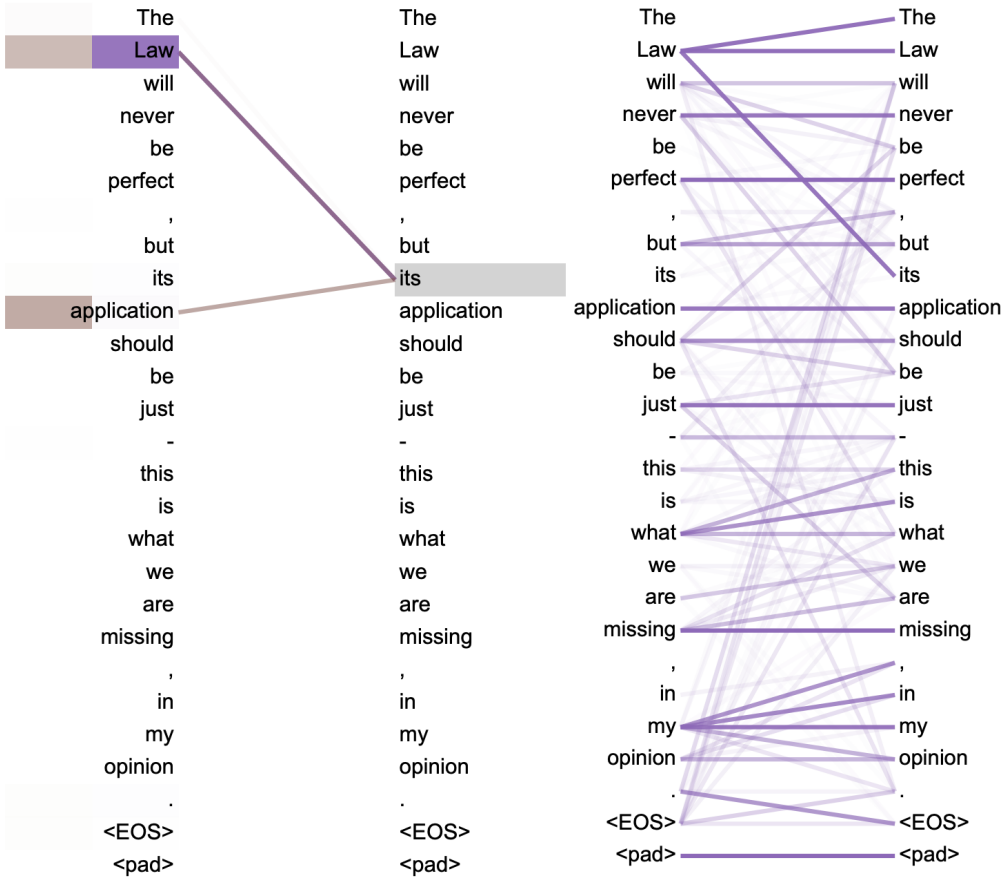


deeply bidirectional

Transformer intuition: refine its state for many rounds (multi-hop reasoning)

Self-Attention Visualization (Vaswani et al., 2017)

Layer 5 and 6, one of the "heads"



Different heads learn different weights

Details: Position Encodings, Masking

- Attention does not differentiate positions.

$$\textbf{Attn}(q, ((k_1, k_2), (v_1, v_2))) = \textbf{Attn}(q, ((k_2, k_1), (v_2, v_1)))$$

- Solution: Explicitly model positions at input level. Various approaches:
  - **Absolute positions**: Introduce an embedding $\pi_i \in \mathbb{R}^d$ for positions $i = 1, 2, \ldots,$ and use $(x_1 + \pi_1, \ldots, x_T + \pi_T)$ as input.
  - **Relative positions**: Introduce an embedding $\pi_i \in \mathbb{R}^d$ for offsets $i = -k, \ldots, k$ between key and query and use it when computing logits between key-query pairs.
- If we don't want q to attend to certain position i, can specify that by "masking" logits

$$\text{Mask}_{\{2,3\}}(q^\top k_1, q^\top k_2, q^\top k_3) = (q^\top k_1, -\infty, -\infty)$$

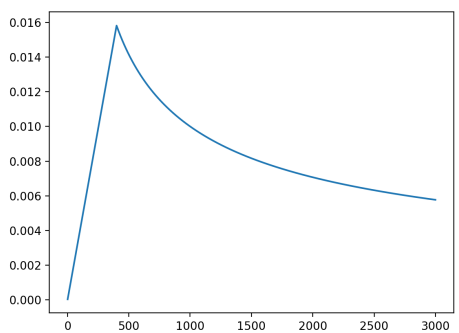
Details: Scaled Dot Product, Training

- In practice we scale dot products by $1/\sqrt{d}$

$$\textbf{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

  Helps stabilize the variance of dot products when d is large.

- Training transformers can be nontrivial. Original work needed a very specific training setting
  - Xavier uniform initialization
  - Adam optimizer with non-default hyperparameters
  - Learning rate schedule: *manually* change learning rate during training (on top of Adam's adaptive learning rate)



Summary

Input: **word embeddings**, learnable vectors for distinct word types

- **Averaging word embeddings**: Simplest way to get a vector out of text, but cannot model word ordering
- **CNNs**: Learn n-gram filters, still unable to model general word ordering
- **RNNs**: Recurrent updates naturally model word ordering, but cannot be parallelized and one-sided (or shallowly bidirectional)
- **Transformers**: Make all inputs attend to all inputs directly via self-attention, stacked to capture "deep" patterns (deeply bidirectional), can be parallelized and made sensitive to word ordering with position encodings, but can be tricky to train

Any of these encoders can be "plugged in" to optimize a task-specific loss!