(Self-)Attention and the Transformer

Deep Learning for NLP: Lecture 8

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilians-Universität München
poerner@cis.uni-muenchen.de

December 23, 2020

Outline

1. Recap: Attention

2. Transformer
   - Cross-attention and self-attention
   - Parallelized attention
   - Multi-layer attention
   - Multi-head attention
   - Masked self-attention
   - Residual connections and layer normalization
   - The Transformer architecture
   - Position encodings

Outline

Limitations of RNNs

- In an RNN, at a given point in time j, the information about all past inputs $x^{(1)} \dots x^{(j)}$ is "crammed" into the state vector $\boldsymbol{h}^{(j)}$ (and $\boldsymbol{c}^{(j)}$ for an LSTM)
- So for long sequences, the state becomes a bottleneck
- Especially problematic in encoder-decoder models (e.g., for Machine Translation)
- Solution: Attention (Bahdanau et al., 2015) – an architectural modification of the RNN encoder-decoder that allows the model to "attend to" past encoder states

Attention: The basic recipe

- **Ingredients:**

- One query vector: $\mathbf{q} \in \mathbb{R}^{d_q}$
- J key vectors: $\mathbf{K} \in \mathbb{R}^{J \times d_k}; (\boldsymbol{k}_1 \ldots \boldsymbol{k}_J)$
- J value vectors: $\mathbf{V} \in \mathbb{R}^{J \times d_v}; (\boldsymbol{v}_1 \ldots \boldsymbol{v}_J)$
- Scoring function $a : \mathbb{R}^{d_q} \times \mathbb{R}^{d_k} \rightarrow \mathbb{R}$
  - Maps a query-key pair to a scalar ("score")
  - a may be parametrized by parameters θ_a

Attention: The basic recipe

- **Step 1**: Apply a to $\boldsymbol{q}$ and all keys $\boldsymbol{k}_j$ to get scores (one per key):

$$\boldsymbol{e} = \begin{bmatrix} e_1 \\ \vdots \\ e_J \end{bmatrix} = \begin{bmatrix} a(\boldsymbol{q}, \boldsymbol{k}_1; \theta_a) \\ \vdots \\ a(\boldsymbol{q}, \boldsymbol{k}_J; \theta_a) \end{bmatrix}$$

- **Step 2**: Turn **e** into a probability distribution with the softmax function

$$\alpha_j = \frac{\exp(e_j)}{\sum_{j'=1}^{J} \exp(e_{j'})}$$

  - Note that $\sum_j \alpha_j = 1$

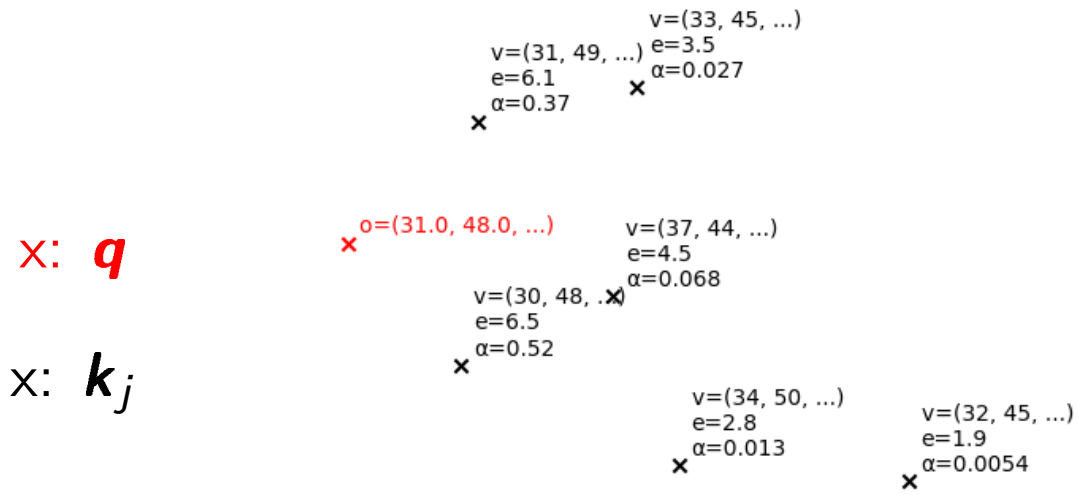
- **Step 3**: α-weighted sum over $\boldsymbol{V}$ yields one d_v-dimensional output vector:

$$\boldsymbol{o} = \sum_{j=1}^{J} \alpha_j \boldsymbol{v}_j$$

  - Intuition: α_j is how much "attention" the model pays to $\boldsymbol{v}_j$ when computing $\boldsymbol{o}$.

Attention: An analogy

- We have J weather stations on a map
- $\boldsymbol{K} \in \mathbb{R}^{J \times 2}$ are their geolocations (x,y coordinates)
- $\boldsymbol{V} \in \mathbb{R}^{J \times d_v}$ are their current weather conditions (temperature, humidity, etc.)
- $\boldsymbol{q} \in \mathbb{R}^2$ is a new geolocation for which we want to estimate weather conditions
- e_j is the relevance of the j'th station (e.g., $e_j = a(\boldsymbol{q}, \boldsymbol{k}_j) = \frac{1}{||\boldsymbol{q} - \boldsymbol{k}_j||_2}$), and α_j is e_j as a probability
- $\boldsymbol{o}$: a weighted sum of all known weather conditions, where stations that have a small distance (high α) have a higher weight



Attention in neural networks

- Contrary to our geolocation example, the $\boldsymbol{q}$, $\boldsymbol{k}_j$ and $\boldsymbol{v}_j$ vectors of a neural network are produced as a function of the input and some trainable parameters
- So the *model* learns which keys are relevant for which queries, based on the training data and loss function

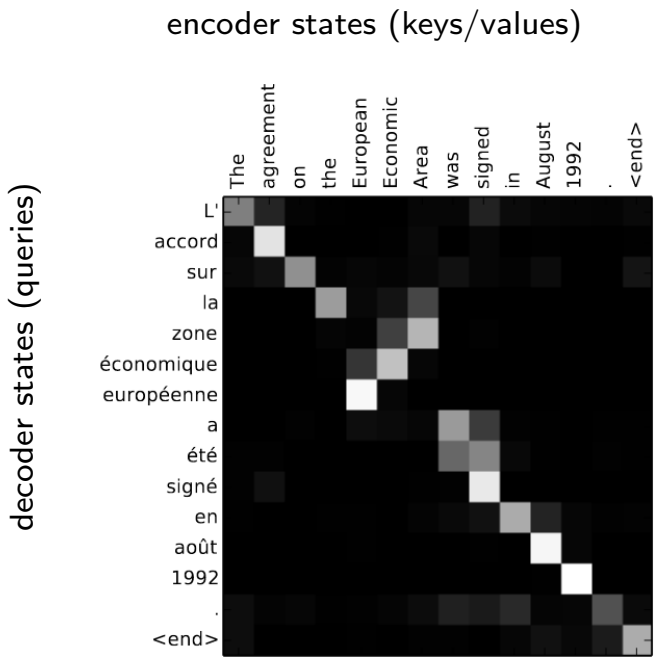


Figure from Bahdanau et al. 2015: Neural Machine Translation by Jointly Learning to Align and Translate.

Attention in neural networks

- No (or few) assumptions are baked into the architecture (no notion of which words are neighbors in the sentence, sequentiality, etc.)
- The lack of prior knowledge often means that the Transformer requires more training data than an RNN/CNN to achieve a certain performance
- But when presented with sufficient data, it usually outperforms them
- After Christmas: Transfer learning as a way to pretrain Transformers on **lots** of data

Questions?

Take a moment to write down any questions you have for the QA session!

Outline

1. Recap: Attention

2. Transformer
   - Cross-attention and self-attention
   - Parallelized attention
   - Multi-layer attention
   - Multi-head attention
   - Masked self-attention
   - Residual connections and layer normalization
   - The Transformer architecture
   - Position encodings

- The Bahdanau model is still an RNN, just with attention on top.
- Architecture that consists of attention only: Transformer (Vaswani et al. (2017), "Attention is all you need")

The Transformer architecture (sequence-to-sequence)

(For simpler problems (e.g., classification, tagging), you would simply use the encoder.)

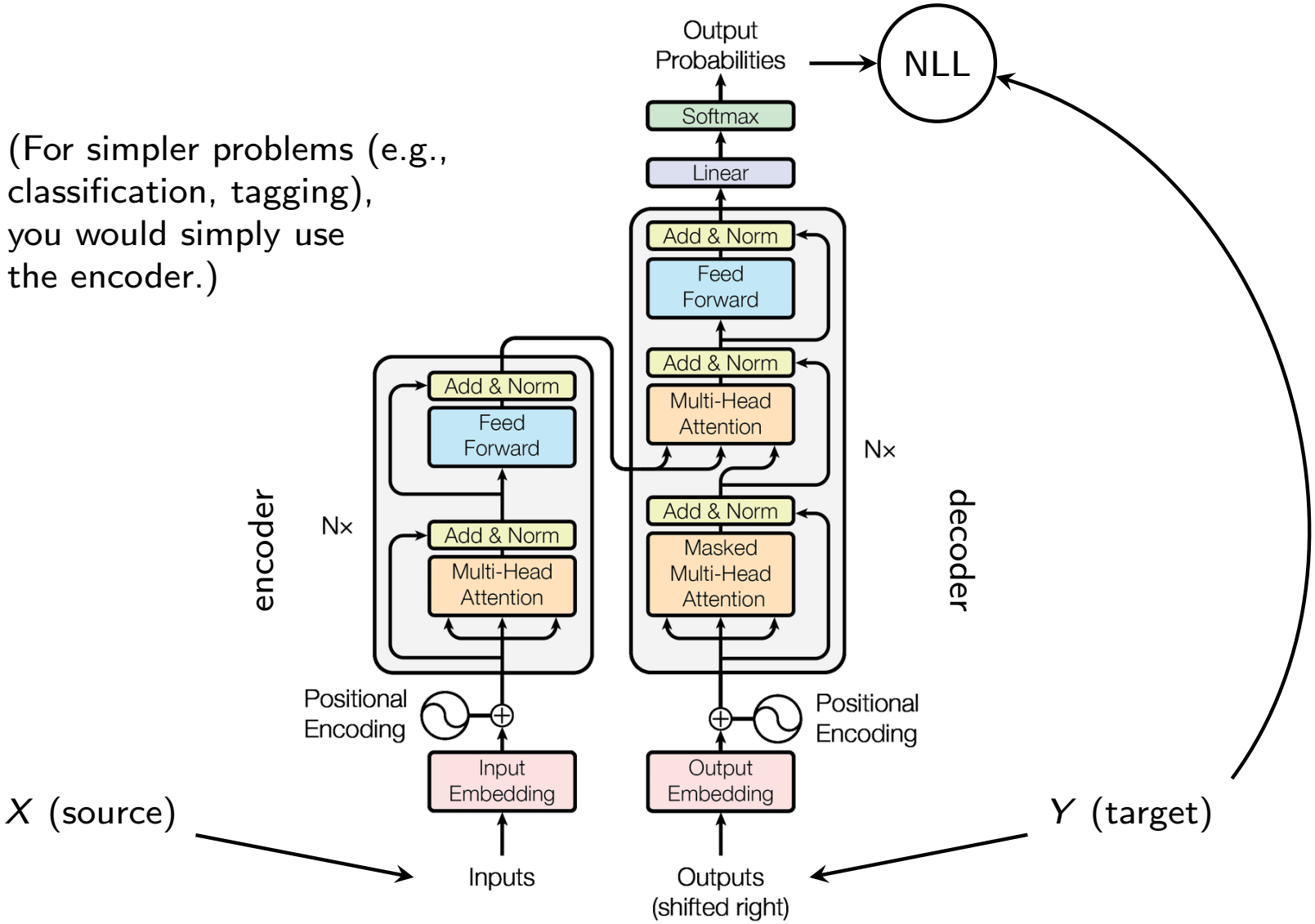


Figure from Vaswani et al. 2017: Attention is all you need

Outline

Cross-attention and self-attention

- We can use attention on many different "things", including:
  - The pixels of images
  - The nodes of knowledge graphs
  - The words of a vocabulary
  - ...
- Here, we focus on scenarios where the query, key and value vectors represent tokens (e.g., words, characters, etc.) in sequences (e.g., sentences, paragraphs, etc.).
- Cross-attention:
  - Let $X = (x_1 \dots x_{J_x})$, $Y = (y_1 \dots y_{J_y})$ be two sequences (e.g., source and target in a sequence-to-sequence problem)
  - The query vectors represent tokens in Y and the key/value vectors represent tokens in X ("Y attends to X")
- Self-attention:
  - There is only one sequence $X = (x_1 \dots x_J)$
  - The query, key and value vectors represent tokens in X ("X attends to itself")

Cross-attention

- Here, we describe cross-attention. Self-attention can easily be derived by assuming $\boldsymbol{X} = \boldsymbol{Y}$.
- Let $\boldsymbol{X} \in \mathbb{R}^{J_x \times d_x}$, $\boldsymbol{Y} \in \mathbb{R}^{J_y \times d_y}$ be representations of X, Y (e.g., stacked word embeddings, or the outputs of a previous layer)
- Let $\theta = \{\boldsymbol{W}^{(q)} \in \mathbb{R}^{d_y \times d_q}, \boldsymbol{W}^{(k)} \in \mathbb{R}^{d_x \times d_k}, \boldsymbol{W}^{(v)} \in \mathbb{R}^{d_x \times d_v}\}$ be trainable weight matrices
- We transform $\boldsymbol{Y}$ into a matrix of query vectors:

$$\boldsymbol{Q} = \boldsymbol{Y}\boldsymbol{W}^{(q)}$$

- We transform $\boldsymbol{X}$ into matrices of key and value vectors:

$$\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}^{(k)}; \qquad \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}^{(v)}$$

Cross-attention

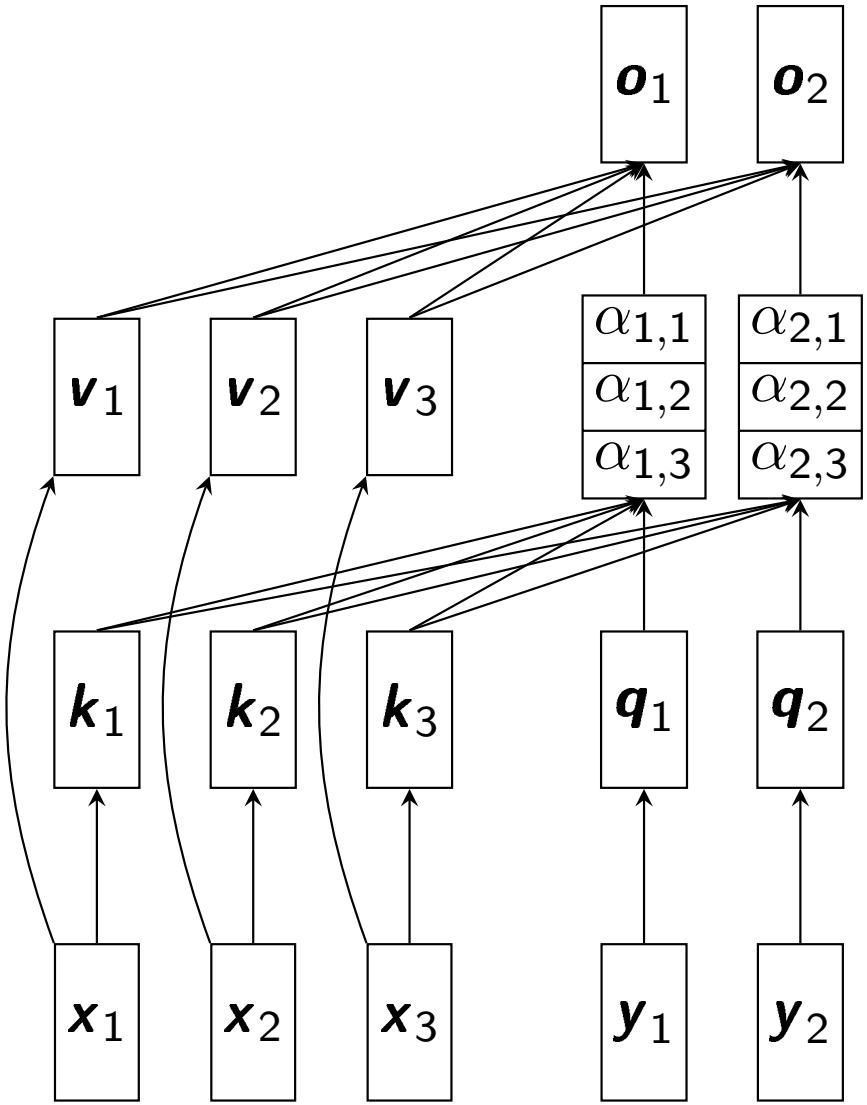
- To calculate the e scores (step 1 of the basic recipe), Vaswani et al. use a parameter-less scaled dot product instead of Bahdanau's complicated FFN:

$$e_{j,j'} = a(\boldsymbol{q}_j, \boldsymbol{k}_{j'}) = \frac{\boldsymbol{q}_j^T \boldsymbol{k}_{j'}}{\sqrt{d_k}}$$

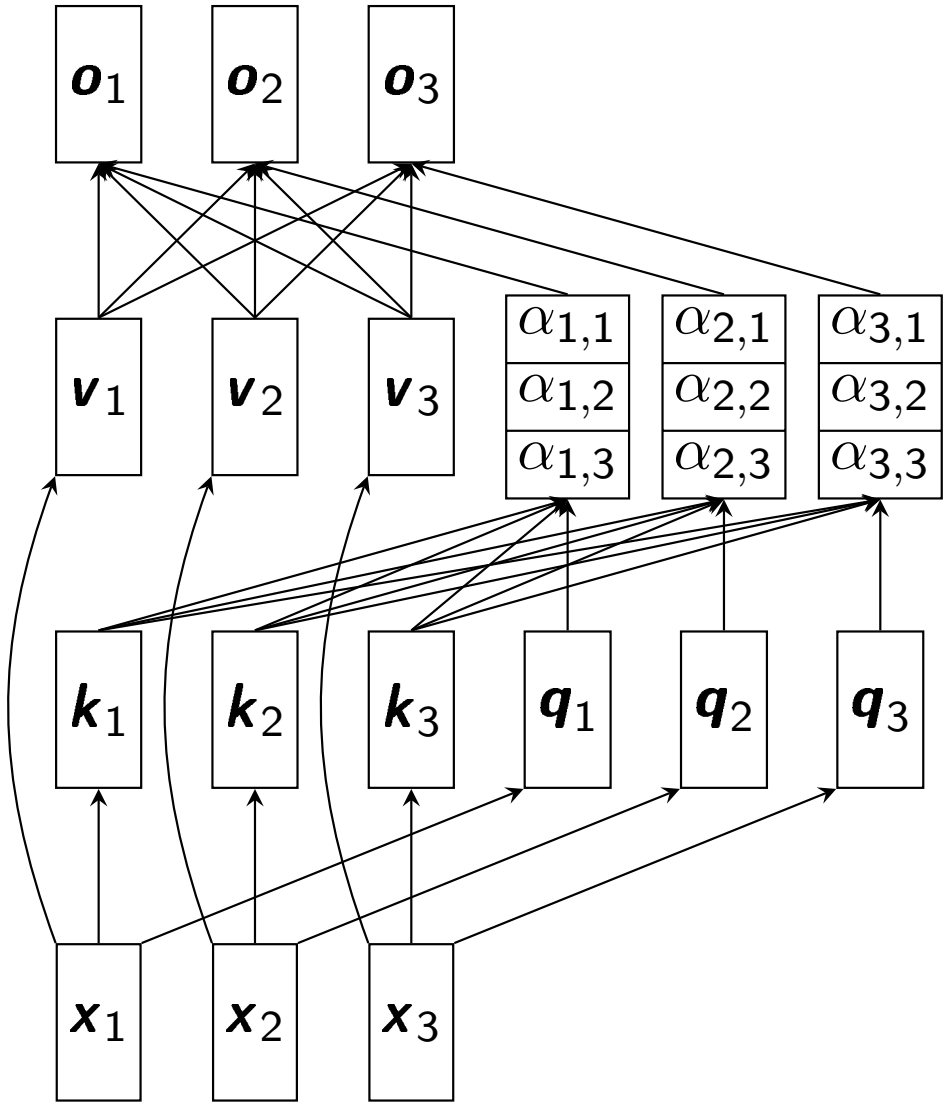
- Note: This requires that $d_q = d_k$
- Attention weights and outputs are defined like before (steps 2 and 3 of the basic recipe):

$$\alpha_{j,j'} = \frac{\exp(e_{j,j'})}{\sum_{j''=1}^{J_x} \exp(e_{j,j''})}$$

$$\boldsymbol{o}_j = \sum_{j'=1}^{J_x} \alpha_{j,j'} \boldsymbol{v}_{j'}$$



Cross-attention



Self-attention

Outline

Parallelized attention

- We want to apply our attention recipe to every query vector $\boldsymbol{q}_j$
- We could simply loop over all time steps $1 \leq j \leq J_y$ and calculate each $\boldsymbol{o}_j$ independently.
- Then stack all $\boldsymbol{o}_j$ into an output matrix $\boldsymbol{O} \in \mathbb{R}^{J_y \times d_v}$
- But a loop does not use the GPU's capacity for parallelization
- So it might be unnecessarily slow

Parallelized attention

- Do some inputs (e.g., $\boldsymbol{q}_j$) depend on previous outputs (e.g., $\boldsymbol{o}_{j-1}$)? If not, we can parallelize the loop into a single function:

$$\boldsymbol{O} = \mathcal{F}^{\text{attn}}(\boldsymbol{X}, \boldsymbol{Y}; \theta)$$

- Attention in Transformers is usually parallelizable, unless we are doing autoregressive inference (more on that later).
- By the way: The Bahdanau model is not parallelizable in this way, because s_i (a.k.a. the query of the $i+1$'st step) depends on c_i (a.k.a. the attention output of the i'th step), see last lecture:

The hidden state s_i of the decoder given the annotations from the encoder is computed by

$$s_i = (1 - z_i) \circ s_{i-1} + z_i \circ \tilde{s}_i,$$

where

$$\begin{aligned}
\tilde{s}_i &= \tanh\left(WEy_{i-1} + U\left[r_i \circ s_{i-1}\right] + Cc_i\right) \\
z_i &= \sigma\left(W_z Ey_{i-1} + U_z s_{i-1} + C_z c_i\right) \\
r_i &= \sigma\left(W_r Ey_{i-1} + U_r s_{i-1} + C_r c_i\right)
\end{aligned}$$

Parallelized scaled dot product attention

- **Step 1**: The parallel application of the scaled dot product to all query-key pairs can be written as:

$$\boldsymbol{E} = \frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}; \quad \boldsymbol{E} \in \mathbb{R}^{J_y \times J_x}$$

$$\begin{array}{c} \\ \downarrow \\ \textit{queries} \\ \downarrow \end{array} \overset{\rightarrow \textit{keys} \rightarrow}{\begin{bmatrix} e_{1,1} & \dots & e_{1,J_x} \\ \vdots & \ddots & \vdots \\ e_{J_y,1} & \dots & e_{J_y,J_x} \end{bmatrix}} = \frac{1}{\sqrt{d_k}} \begin{bmatrix} - & \boldsymbol{q}_1 & - \\ & \vdots & \\ - & \boldsymbol{q}_{J_y} & - \end{bmatrix} \begin{bmatrix} | & & | \\ \boldsymbol{k}_1 & \dots & \boldsymbol{k}_{J_x} \\ | & & | \end{bmatrix}$$

Parallelized scaled dot product attention

- **Step 2**: Softmax with normalization over the second axis (key axis):

$$\alpha_{j,j'} = \frac{\exp(e_{j,j'})}{\sum_{j''=1}^{J_x} \exp(e_{j,j''})}$$

```
>>> A = np.exp(E) / np.exp(E).sum(axis=-1, keepdims=True)
```

- Let's call this new normalized matrix $\boldsymbol{A} \in (0,1)^{J_y \times J_x}$
- The rows of $\boldsymbol{A}$, denoted $\boldsymbol{\alpha}_j$, are probability distributions (one $\boldsymbol{\alpha}_j$ per $\boldsymbol{q}_j$)
- **Step 3**: Weighted sum

$$\boldsymbol{O} = \boldsymbol{A}\boldsymbol{V}; \boldsymbol{O} \in \mathbb{R}^{J_y \times d_v}$$

$$\begin{array}{c} \\ \downarrow \\ \textit{queries} \\ \downarrow \end{array} \overset{\rightarrow d_v \text{(value dims)} \rightarrow}{\begin{bmatrix} o_{1,1} & \cdots & o_{1,d_v} \\ \vdots & \ddots & \vdots \\ o_{J_y,1} & \cdots & o_{J_y,d_v} \end{bmatrix}} = \begin{bmatrix} - & \boldsymbol{\alpha}_1 & - \\ & \vdots & \\ - & \boldsymbol{\alpha}_{J_y} & - \end{bmatrix} \begin{bmatrix} | & & | \\ \boldsymbol{v}_{:,1} & \cdots & \boldsymbol{v}_{:,d_v} \\ | & & | \end{bmatrix}$$

... as a one-liner

$$\boldsymbol{O} = \mathcal{F}^{\text{attn}}(\boldsymbol{X}, \boldsymbol{Y}; \theta) = \text{softmax}\left(\frac{(\boldsymbol{Y}\boldsymbol{W}^{(q)})(\boldsymbol{X}\boldsymbol{W}^{(k)})^T}{\sqrt{d_k}}\right)(\boldsymbol{X}\boldsymbol{W}^{(v)})$$

- GPUs like matrix multiplications $\rightarrow$ usually a lot faster than RNN!
- But: The memory requirements of $\boldsymbol{E}$ and $\boldsymbol{A}$ are $\mathcal{O}(J_y J_x)$
- A length up to about 500 is usually ok on a medium-sized GPU (and most sentences are shorter than that anyway).
- But when we consider inputs that span several sentences (e.g., paragraphs or whole documents), we need tricks to reduce memory. These are beyond the scope of this lecture.

Questions?

Take a moment to write down any questions you have for the QA session!

Outline

Multi-layer attention

- Sequential application of several attention layers, with separate parameters $\{\theta^{(1)} \dots \theta^{(N)}\}$
- In Transformer: sequential application of Transformer blocks
- There are some additional position-wise layers inside the Transformer block, i.e., $\boldsymbol{O}^{(n)}$ undergoes some additional transformations before becoming the input to the next Transformer block $n+1$

Outline

Multi-head attention

- Application of several attention layers ("heads") in parallel
- M sets of parameters $\{\theta^{(1)}, \ldots, \theta^{(M)}\}$, with $\theta^{(m)} = \{\boldsymbol{W}^{(m,q)}, \boldsymbol{W}^{(m,k)}, \boldsymbol{W}^{(m,v)}\}$
- For every head, compute in parallel:

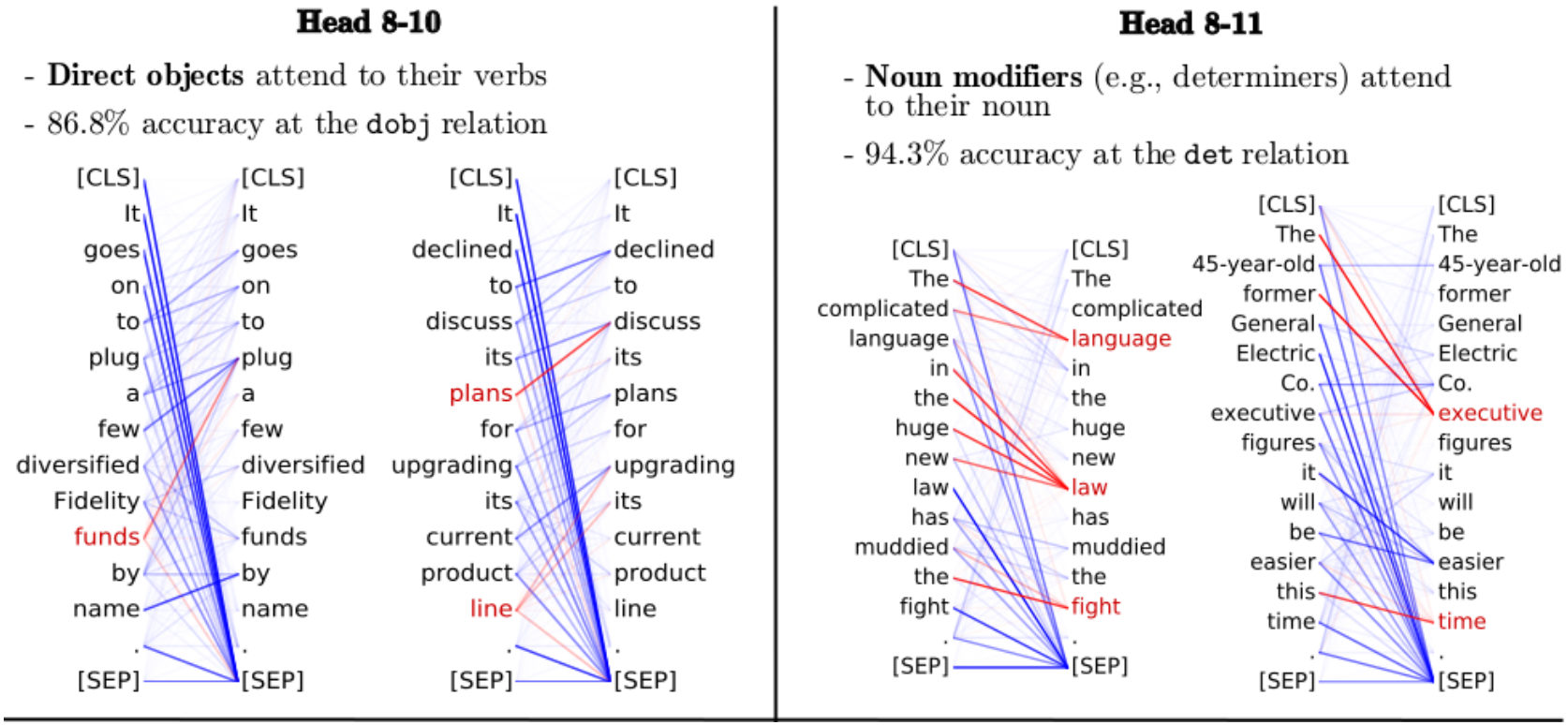
$$\boldsymbol{O}^{(m)} = \mathcal{F}^{\text{attn}}(\boldsymbol{X}, \boldsymbol{Y}; \theta^{(m)})$$

- Concatenate all $\boldsymbol{O}^{(m)}$ along their last axis; then down-project the concatenation with an additional parameter matrix $\boldsymbol{W}^{(o)} \in \mathbb{R}^{Md_v \times d_v}$:

$$\boldsymbol{O} = [\boldsymbol{O}^{(1)}; \ldots; \boldsymbol{O}^{(M)}]\boldsymbol{W}^{(o)}$$

Multi-head attention

- Conceptually, multi-head attention is to single-head attention like a filter bank is to a single filter (Lecture 6 on CNNs)
- Division of labor: different heads model different kinds of inter-word relationships

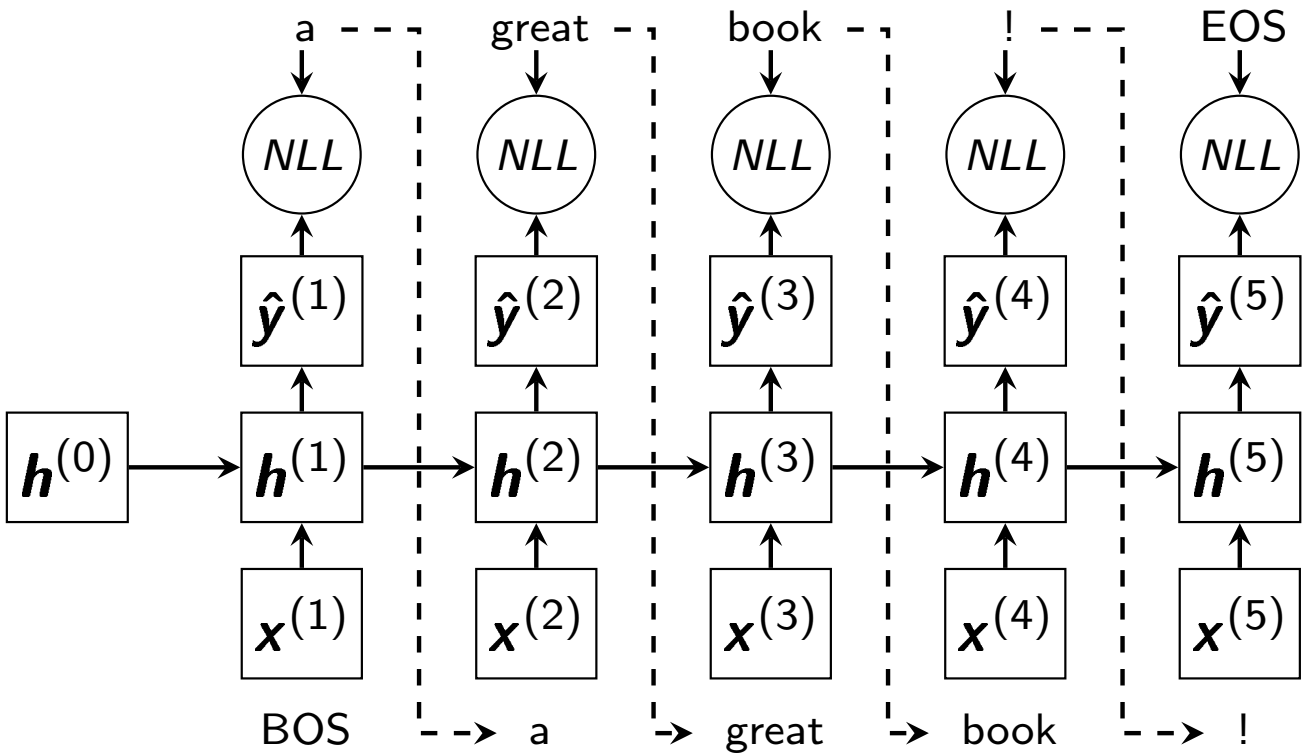


Clark et al. (2018): What Does BERT Look At? An Analysis of BERT's Attention

Outline

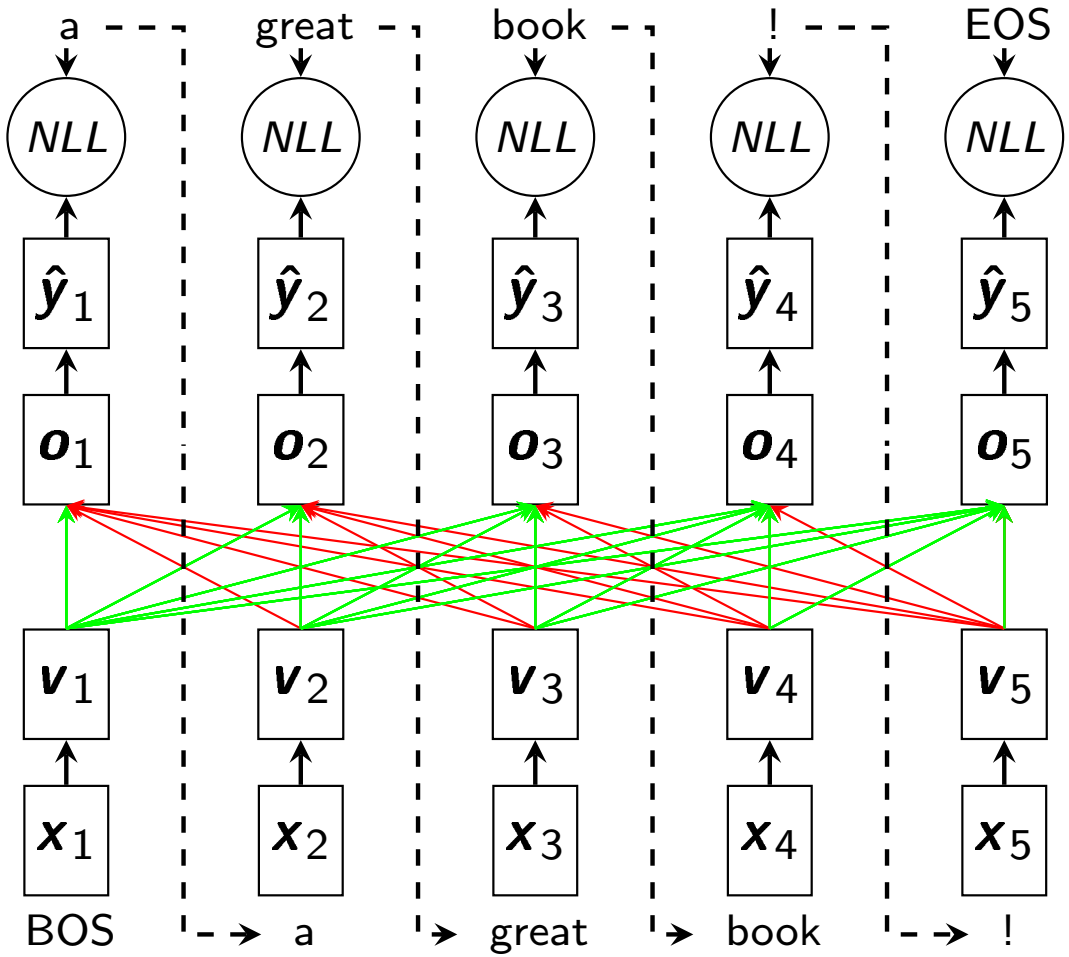
Recap: RNNs for autoregressive language modeling or decoding

- In autoregressive language modeling, or in the decoder of a sequence-to-sequence model, the task is to always predict the next word
- In an RNN, a given state $\boldsymbol{h}^{(j)}$ depends on past inputs $x^{(1)} \ldots x^{(j)}$
- Thus, the RNN is unable to "cheat":



Self-attention for autoregressive LM & decoding

- With attention, all $\boldsymbol{o}_j$ depend on all $\boldsymbol{v}_{j'}$ (and by extension, all $\boldsymbol{x}_{j'}$).
- This means that the model can easily cheat by looking at future words (red connections)



Masked self-attention

- So when we use self-attention for language modeling or in a sequence-to-sequence decoder, we have to prevent $\boldsymbol{o}_j$ from attending to any $\boldsymbol{v}_{j'}$ where $j' > j$.
- **Question:** How can we do that?
- Remember:

$$\boldsymbol{o}_j = \sum_{j'=1}^{J} \alpha_{j,j'} \boldsymbol{v}_{j'}$$

$$\alpha_{j,j'} = \frac{\exp(e_{j,j'})}{\sum_{j''=1}^{J} \exp(e_{j,j''})}$$

  - By hardcoding $e_{j,j'} = -\infty$ when $j' > j$ (in practice, "∞" is just a large constant)
  - That way, $\exp(e_{j,j'}) = \alpha_{j,j'} = 0$, so $\boldsymbol{v}_{j'}$ has no impact on $\boldsymbol{o}_j$

Parallelized masked self-attention

- Step 1: Calculate $\boldsymbol{E}$ like we usually would
- Step 1B:

$$\boldsymbol{E}^{\text{masked}} = \boldsymbol{E} \odot \boldsymbol{M} + \infty \boldsymbol{M} - \infty; \qquad m_{j,j'} = \begin{cases} 1 & \text{if } j' \leq j \\ 0 & \text{otherwise} \end{cases}$$

- Example:

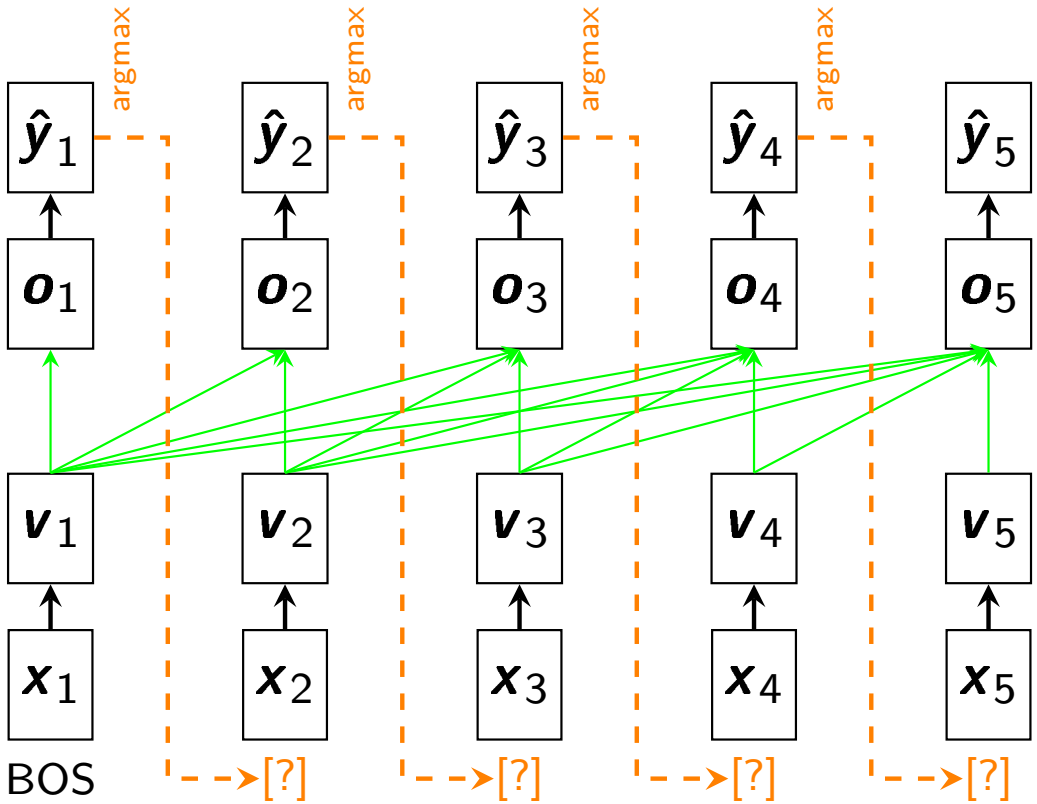
$$\boldsymbol{E} = \begin{bmatrix} e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,1} & e_{2,2} & e_{2,3} \\ e_{3,1} & e_{3,2} & e_{3,3} \end{bmatrix}; \qquad \boldsymbol{M} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\boldsymbol{E}^{\text{masked}} = \begin{bmatrix} e_{1,1} & -\infty & -\infty \\ e_{2,1} & e_{2,2} & -\infty \\ e_{3,1} & e_{3,2} & e_{3,3} \end{bmatrix}; \qquad \boldsymbol{A}^{\text{masked}} = \begin{bmatrix} 1 & 0 & 0 \\ \alpha_{2,1} & \alpha_{2,2} & 0 \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} \end{bmatrix}$$

$$\boldsymbol{o}_1 = \boldsymbol{v}_1; \qquad \boldsymbol{o}_2 = \alpha_{2,1}\boldsymbol{v}_1 + \alpha_{2,2}\boldsymbol{v}_2; \qquad \boldsymbol{o}_3 = \alpha_{3,1}\boldsymbol{v}_1 + \alpha_{3,2}\boldsymbol{v}_2 + \alpha_{3,3}\boldsymbol{v}_3$$

Autoregressive Transformer at inference time

- During training, when targets are known, we use parallelized masked attention
- At inference time, when we don't know what the targets are, we have to decode the prediction in a loop
- Slower, but at least we don't have to worry about masking anymore



Questions?

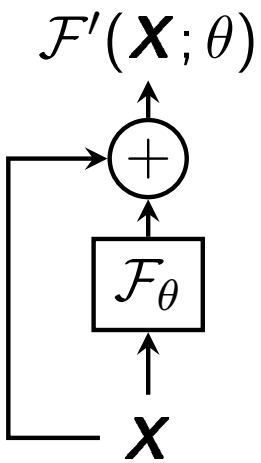
Take a moment to write down any questions you have for the QA session!

Outline

Residual connections

- Let $\mathcal{F}$ be a function with parameters θ
- $\mathcal{F}$ with a residual connection:

$$\mathcal{F}'(\boldsymbol{X};\theta) = \mathcal{F}(\boldsymbol{X};\theta) + \boldsymbol{X}$$



- Benefits: Information retention (we add to $\boldsymbol{X}$ but don't replace it)

Layer normalization

- Let $\theta = \{\boldsymbol{\gamma} \in \mathbb{R}^d, \boldsymbol{\beta} \in \mathbb{R}^d\}$ be trainable parameters
- Let $\boldsymbol{h} \in \mathbb{R}^d$ be an output vector of some layer (e.g., an $\boldsymbol{o}_j$ vector from an attention layer)
- Then layer normalization calculates:

$$\boldsymbol{\gamma} \odot \frac{\boldsymbol{h} - \mu}{\sigma} + \boldsymbol{\beta}$$

- where μ, σ are mean and standard deviation over the dimensions of $\boldsymbol{h}$:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} h_i; \qquad \sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(h_i - \mu)^2}$$

- Benefits: Allows us to normalize vectors after every layer; helps against exploding activations on the forward pass
- In the Transformer, layer normalization is applied position-wise, i.e., every $\boldsymbol{o}_j$ is normalized independently

Outline

The Transformer architecture (sequence-to-sequence)

(For simpler problems (e.g., classification, tagging), you would simply use the encoder.)

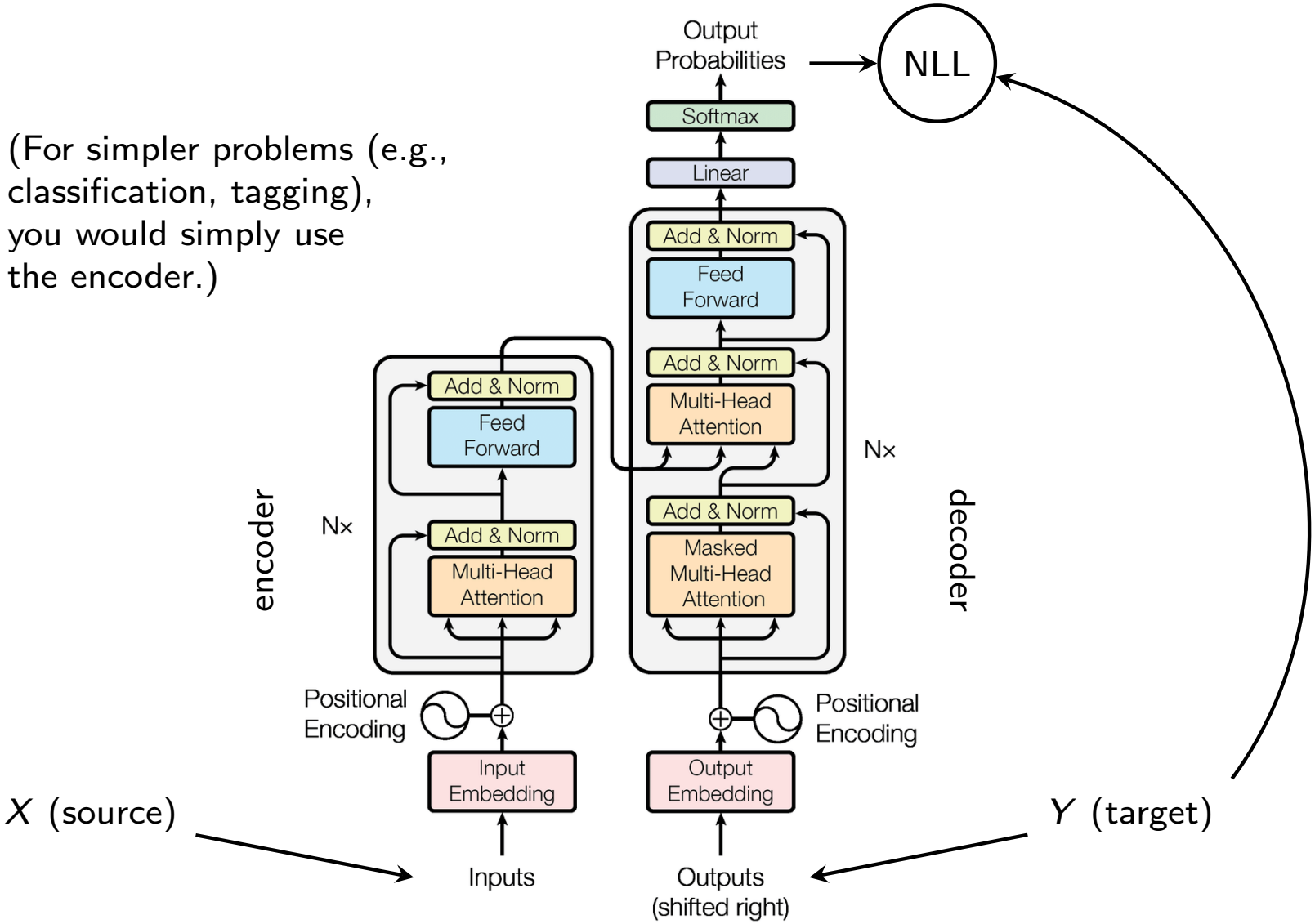


Figure from Vaswani et al. 2017: Attention is all you need

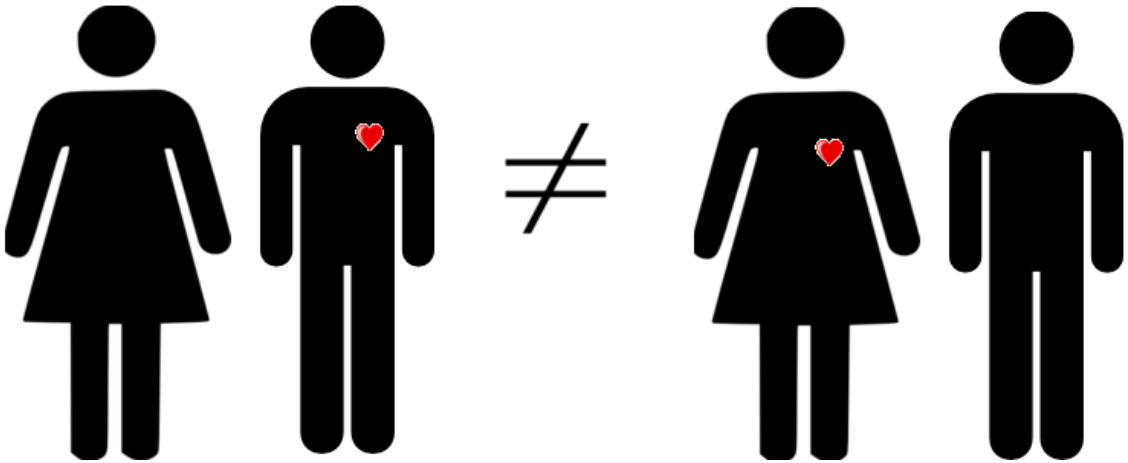
Questions?

Take a moment to write down any questions you have for the QA session!

Outline

Can self-attention model word order?

- Our model consists of a self-attention layer on top of a simple word embedding lookup layer. (For simplicity, we only consider one head, but this applies to multi-head attention as well.)
- Let $X^{(1)}$, $X^{(2)}$ be two sentences of the same length J, which contain the same words in a different order
- Example: "john loves mary" vs. "mary loves john"



Can self-attention model word order?

- Let $\boldsymbol{g} \in \{1, ..., J\}^J$ be the permutation of the word order between $X^{(1)}$ and $X^{(2)}$. So we know that $j \rightarrow g_j$ is bijective, and $\forall_{j \in \{1,...,J\}} x_j^{(1)} = x_{g_j}^{(2)}$
  - In our example: $\boldsymbol{g} = [3, 2, 1]^T$
- Can our self-attention layer differentiate between word orders, or is $\forall_{j \in \{1,...,J\}} \boldsymbol{o}_j^{(1)} = \boldsymbol{o}_{g_j}^{(2)}$?
- (For example, do mary and mary get the same vector representation? If so, that is bad news for very basic Natural Language Understanding capabilities, such as figuring out what is the subject or object of an English verb.)

Can self-attention model word order?

- We want to show that:

$$\forall_{j \in \{1,\ldots,J\}} \boldsymbol{o}_j^{(1)} = \boldsymbol{o}_{g_j}^{(2)}$$

- Let's start with $\boldsymbol{x}_j$.

- Our word embedding lookup layer represents x_j as $\boldsymbol{x}_j = \boldsymbol{w}_{\mathcal{I}(x_j)}$ (see lecture 5), where $\mathcal{I}$ is a bijective indexing function.
- So it is trivially true that:

$$x_j^{(1)} = x_{g_j}^{(2)} \implies \boldsymbol{w}_{\mathcal{I}(x_j^{(1)})} = \boldsymbol{w}_{\mathcal{I}(x_{g_j}^{(2)})} \implies \boldsymbol{x}_j^{(1)} = \boldsymbol{x}_{g_j}^{(2)}$$

Can self-attention model word order?

- Definition of $\boldsymbol{o}_j$:

$$\boldsymbol{o}_j = \sum_{j'=1}^{J} \alpha_{j,j'} \boldsymbol{v}_{j'}$$

- Since addition is commutative, and the permutation is bijective, it is sufficient to show that:

$$\forall_{j \in \{1,\ldots,J\}, j' \in \{1,\ldots,J\}} \alpha^{(1)}_{j,j'} \boldsymbol{v}^{(1)}_{j'} = \alpha^{(2)}_{g_j,g_{j'}} \boldsymbol{v}^{(2)}_{g_{j'}}$$

- Step 1: Let's show that $\forall_j \boldsymbol{v}^{(1)}_j = \boldsymbol{v}^{(2)}_{g_j}$
- Definition of $\boldsymbol{v}_j$:

$$\boldsymbol{v}_j = \boldsymbol{W}^{(v)T} \boldsymbol{x}_j$$

- Then:

$$\boldsymbol{x}^{(1)}_j = \boldsymbol{x}^{(2)}_{g_j} \implies \boldsymbol{W}^{(v)T} \boldsymbol{x}^{(1)}_j = \boldsymbol{W}^{(v)T} \boldsymbol{x}^{(2)}_{g_j} \implies \boldsymbol{v}^{(1)}_j = \boldsymbol{v}^{(2)}_{g_j}$$

Can self-attention model word order?

- Step 2: Let's show that $\forall_{j \in \{1,\ldots,J\}, j' \in \{1,\ldots,J\}} \alpha^{(1)}_{j,j'} = \alpha^{(2)}_{g_j, g_{j'}}$
- Definition of $\alpha_{j,j'}$:

$$\alpha_{j,j'} = \frac{\exp(e_{j,j'})}{\sum_{j''=1}^{J} \exp(e_{j,j''})}$$

- Since the sum in the denominator is commutative, and the permutation is bijective, it is sufficient to show that

$$\forall_{j \in \{1,\ldots,J\}, j' \in \{1,\ldots,J\}} e^{(1)}_{j,j'} = e^{(2)}_{g_j, g_{j'}}$$

Can self-attention model word order?

- Definition of $e_{j,j'}$:

$$e_{j,j'} = \frac{1}{\sqrt{d_k}} \boldsymbol{q}_j^T \boldsymbol{k}_{j'} = \frac{1}{\sqrt{d_k}} (\boldsymbol{W}^{(q)T} \boldsymbol{x}_j)^T (\boldsymbol{W}^{(k)T} \boldsymbol{x}_{j'})$$

- Then:

$$\begin{aligned}
& \boldsymbol{x}_j^{(1)} = \boldsymbol{x}_{g_j}^{(2)} \wedge \boldsymbol{x}_{j'}^{(1)} = \boldsymbol{x}_{g_{j'}}^{(2)} \\
\implies & \boldsymbol{W}^{(q)T} \boldsymbol{x}_j^{(1)} = \boldsymbol{W}^{(q)T} \boldsymbol{x}_{g_j}^{(2)} \wedge \boldsymbol{W}^{(k)T} \boldsymbol{x}_{j'}^{(1)} = \boldsymbol{W}^{(k)T} \boldsymbol{x}_{g_{j'}}^{(2)} \\
\implies & \boldsymbol{q}_j^{(1)} = \boldsymbol{q}_{g_j}^{(2)} \wedge \boldsymbol{k}_{j'}^{(1)} = \boldsymbol{k}_{g_{j'}}^{(2)} \\
\implies & \boldsymbol{q}_j^{(1)T} \boldsymbol{k}_{j'}^{(1)} = \boldsymbol{q}_{g_j}^{(2)T} \boldsymbol{k}_{g_{j'}}^{(2)} \\
\implies & \frac{1}{\sqrt{d_k}} \boldsymbol{q}_j^{(1)T} \boldsymbol{k}_{j'}^{(1)} = \frac{1}{\sqrt{d_k}} \boldsymbol{q}_{g_j}^{(2)T} \boldsymbol{k}_{g_{j'}}^{(2)} \\
\implies & e_{j,j'}^{(1)} = e_{g_j,g_{j'}}^{(2)}
\end{aligned}$$

Can self-attention model word order?

- So, $\forall_j \boldsymbol{o}_j^{(1)} = \boldsymbol{o}_{g_j}^{(2)}$
- In other words: The representation of mary is identical to that of mary, and the representation of john is identical to that of john
- **Question:** Can the other layers in the Transformer architecture (feed-forward net, layer normalization) help with the problem?
  - No, because they are apply the same function to all positions.
- **Question:** Would it help to apply more self-attention layers?
  - No. Since the representations of identical words are still identical in $\boldsymbol{O}$, the next self-attention layer will have the same problem.
- So... does that mean the Transformer is unusable?
- Luckily not. We just need to ensure that input embeddings of identical words at different positions are not identical.

Position embeddings

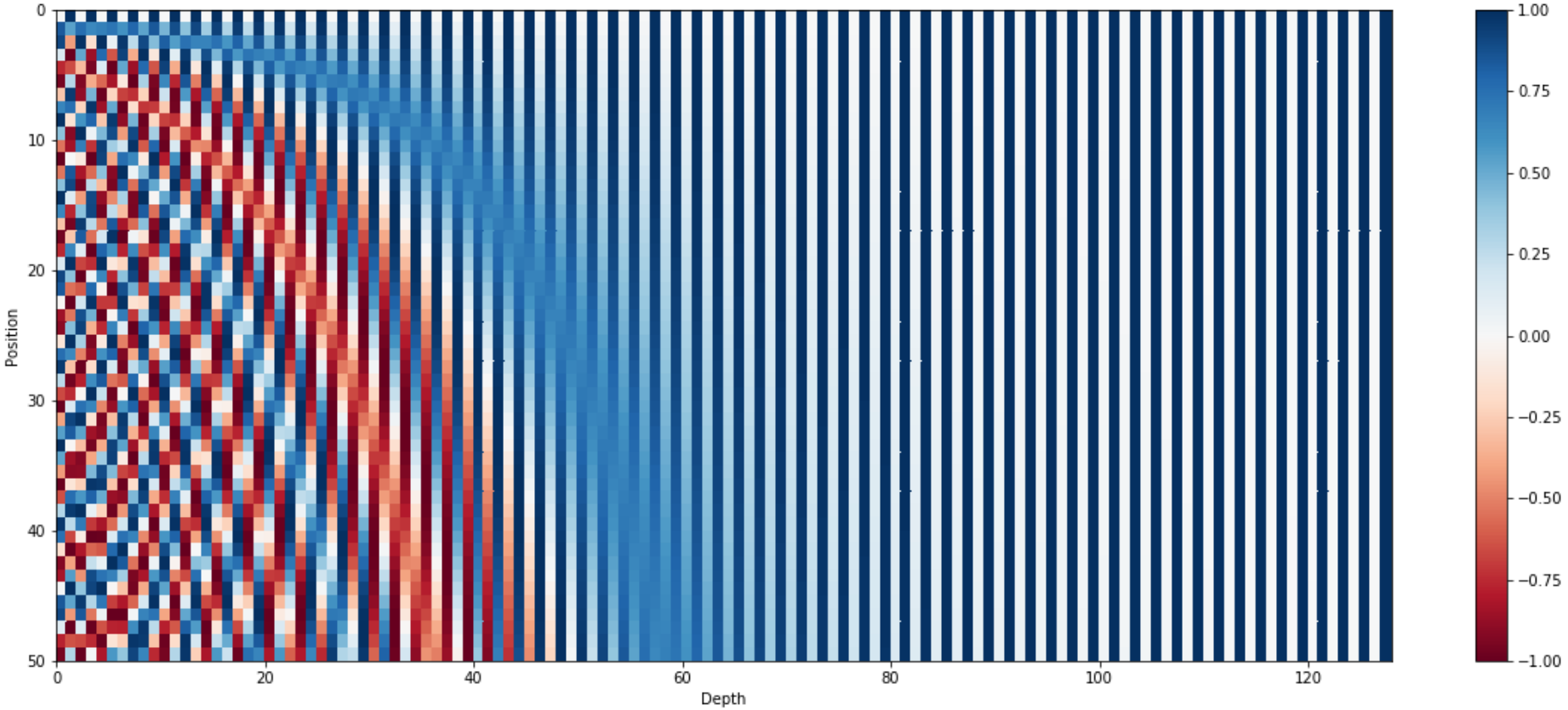
- Add to every input word embedding a position embedding $\boldsymbol{p} \in \mathbb{R}^d$:
- Input embedding of word "mary" in position j: $\boldsymbol{x}_j = \boldsymbol{w}_{\mathcal{I}(\text{mary})} + \mathbf{p}_j$

$$\boldsymbol{w}_{\mathcal{I}(\text{mary})} + \boldsymbol{p}_j \neq \boldsymbol{w}_{\mathcal{I}(\text{mary})} + \boldsymbol{p}_{j'} \text{ if } j \neq j'$$

- Option 1 (Devlin et al., 2018): Trainable position embeddings: $\boldsymbol{P} \in \mathbb{R}^{J^{\max} \times d}$
  - Disadvantage: Cannot deal with sentences that are longer than $J^{\max}$
- Option 2 (Vaswani et al., 2017): Sinusoidal position embeddings (deterministic):

$$p_{j,i} = \begin{cases} \sin\left(\frac{j}{10000^{\frac{i}{d}}}\right) & \text{if } i \text{ is even} \\ \cos\left(\frac{j}{10000^{\frac{i-1}{d}}}\right) & \text{if } i \text{ is odd} \end{cases}$$

Sinusoidal position embeddings



https://kazemnejad.com/blog/transformer_architecture_positional_encoding

Questions?

Take a moment to write down any questions you have for the QA session!

Hyperparameter Optimization

Deep Learning for NLP: Lecture 8b

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilians-Universität München
poerner@cis.uni-muenchen.de

December 23, 2020

Outline

Outline

What are hyperparameters?

- Things that may affect the performance of a model on a task, without being a trainable parameter in θ

Hyperparameters related to the model

- input (word embedding) size
- hidden state size (RNN)
- number of filters (CNN)
- filter size (CNN)
- number of heads (Transformer)
- hidden size of queries/keys/values (Transformer)
- number of layers/blocks
- choice of nonlinearity (ReLU vs. tanh vs. ...)
- ...

Hyperparameters related to training

- batch size
- factor for L1, L2 regularization
- dropout probability
- learning rate
- choice of optimizer (SGD vs. Adam vs. Adagrad vs. ...)
- parameters of optimizer (e.g., momentum, ...)
- number of epochs (early stopping)
- ...

What is hyperparameter optimization?

- Also known as "hyperparameter tuning"
- The process of choosing the best hyperparameters for a given architecture and task
- Different from training your regular parameters, because hyperparameters are not optimized by gradient descent

Outline

Datasets

- Needed: separate train/validation*/test sets
  - *a.k.a. dev(elopment), heldout, valid(ation)
- Many existing datasets come with a predefined train/validation/test split; if that is the case, use the predefined split
- Otherwise, you should split the data yourself (randomly!); 80/10/10 is usually fine, unless there is too little data (see cross-validation)
- You should always report the absolute and relative sizes of the sets

Using the validation set

- For every hyperparameter configuration:
  - Create model
  - Train model on the training set
  - Evaluate model on validation set
- Keep the model that had the best performance on the validation set
- Evaluate that model on the test set

Cross-validation

- Useful when there is very little data, so that you cannot afford separate train and validation sets.
- Split training set into N subsets ("folds")
- For every hyperparameter configuration:
  - For every fold $1 \leq n \leq N$:
    - ★ Fold n is your validation set, the other folds are your training set
    - ★ Create, train and evaluate the model
  - Average the validation performance over all folds
- Keep the configuration that had the best average validation performance
- Train a model with that configuration on the full training set (all folds together)
- Measure the model's performance on the test set

The importance of the test set

- Hyperparameter optimization means overfitting to the validation set
- You should never do hyperparameter optimization on the test set
- The test set should only be used once, to evaluate your final model. Otherwise, your test set is "spoiled".

Outline

- A *hyperparameter configuration* is a specific choice of hyperparameters, e.g.,
  - batch size $= 16$
  - optimizer $=$ Adagrad
  - hidden size $= 50$
  - dropout $= 0.5$
  - ...
- Every hyperparameter has a *range*, or a set of permissible values, e.g.,
  - batch size $\in \{8, 16, 32...\}$
  - optimizer $\in \{\text{SGD}, \text{Adagrad}, ...\}$
  - ...

Defining hyperparameter ranges

- Usually, the range will be informed by your domain expertise:
  - Lots of data $\rightarrow$ low risk of overfitting $\rightarrow$ try big hidden sizes / more layers...
  - Little data $\rightarrow$ high risk of overfitting $\rightarrow$ try low hidden sizes / fewer layers ...
  - Some values just don't make sense (learning rate of 10000, filter size 1 in a CNN)
- In practice, your range will also be limited by your resources (e.g., GPU memory)
- Always report the range, so that others can reproduce your evaluation.

Discretizing continuous ranges

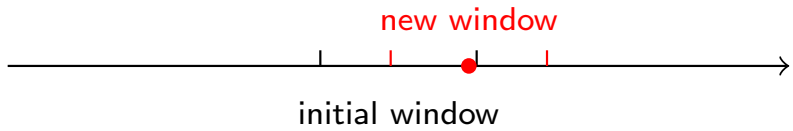
- Continuous hyperparameters (e.g., hidden size, batch size, number of filters) should be discretized.
- For open-ended hyperparameters, use an (approximate) log scale, e.g.,
  - hidden size $\in 50, 100, 200, 500, 1000$
  - batch size $\in 16, 32, 64, 128$
- Categorical hyperparameters (e.g., optimizer or nonlinearity) can be treated as a finite set

Outline

How to search

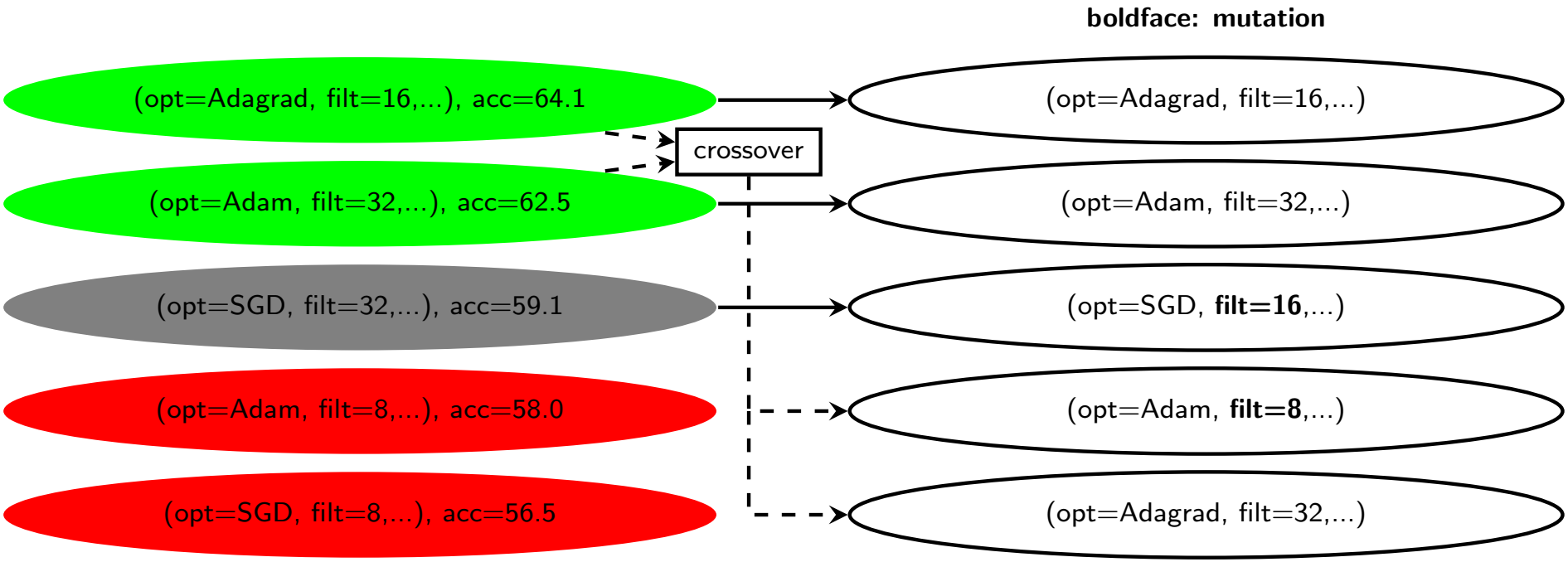
- Brute Force?
  - Pro: Exhaustive
  - Con: Search space grows exponentially.
- Intuition: Some hyperparameters have a big effect, others have a small effect. A configuration that gets the first group right will be almost as good as the optimal configuration.
- Uniform sampling?
  - Pro: Easy to implement, good results in practice
  - Con: Will waste resources on configurations that have little chance of success (e.g., if we have previously found that all configurations with a hidden size of 20 produce bad results, we should stop sampling that particular value)

Shifting window search (credit to Ben Roth)



- For every continuous hyperparameter, define a small window of the range
- Example: full range $= \{4, 8, 16, 32, 64, 128, 256\}$, window $= \{16, 32, 64\}$
- For N iterations:
  - Use the random sampling method N' times, but sample only from the windows.
  - For each hyperparameter, identify the value that led to the best performance.
  - Shift that window, so that the best value lies in the center
- Not applicable to hyperparameters whose values are unordered sets (i.e., categorical hyperparameters)

Evolutionary algorithm



- Start with a random set of configurations (population)
- For N iterations:
  - Delete the worst-performing configurations (survival of the fittest)
  - Randomly combine best-performing hyperparameter configurations to produce "children" (crossover).
  - Inject random changes to explore new possibilities (mutations).

How to search

- For reproducibility, you should always provide details about your method of hyperparameter search
- That includes how many iterations you did, and any search-specific parameters (e.g., mutation probability, ...)

Questions?

Take a moment to write down any questions you have for the QA session!