Alternative Architectures

Philipp Koehn

10 October 2024

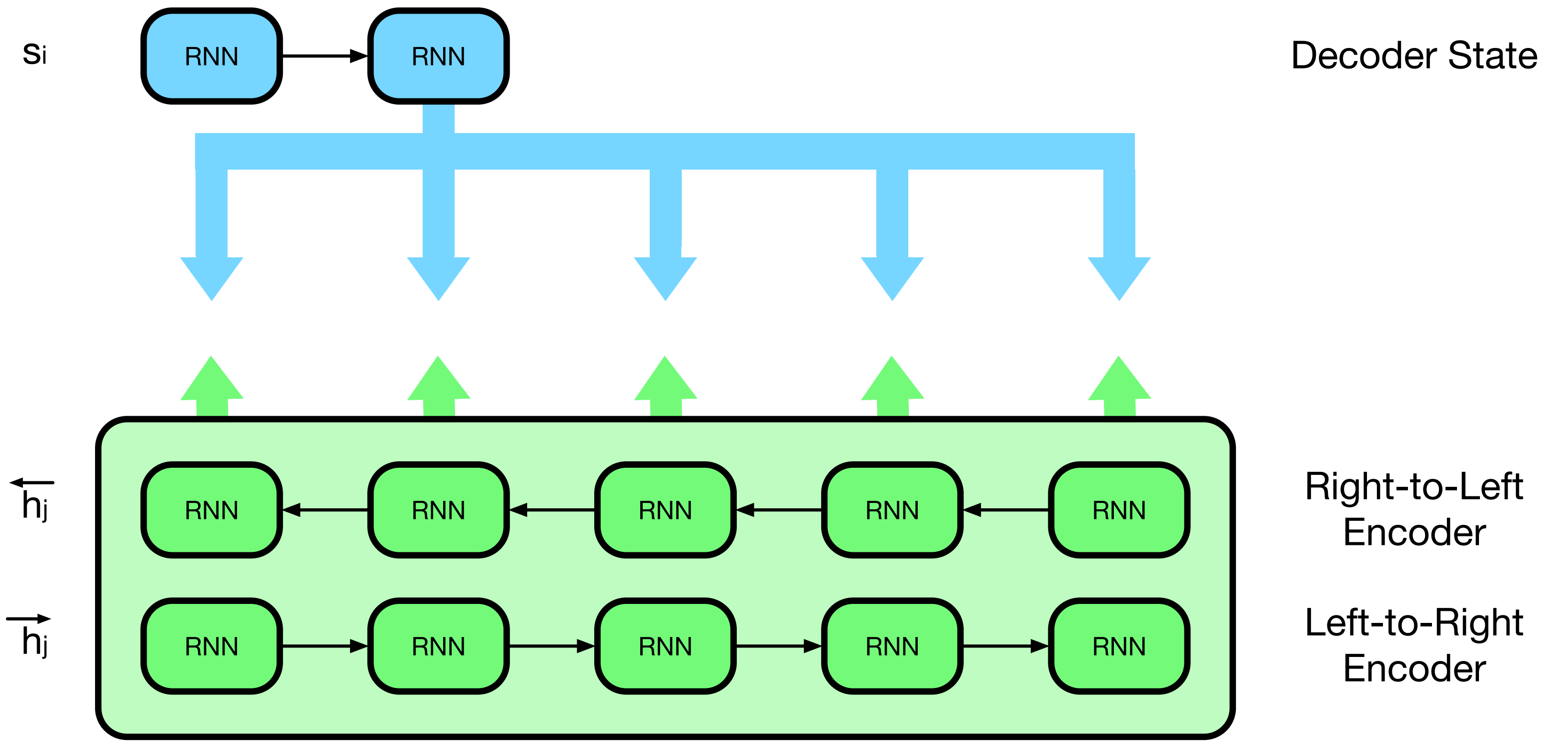
attention

Attention

- Machine translation is a structured prediction task
  - output is not a single label
  - output structure needs to be built, word by word
- Relevant information for each word prediction varies
- Human translators pay attention to different parts of the input sentence when translating

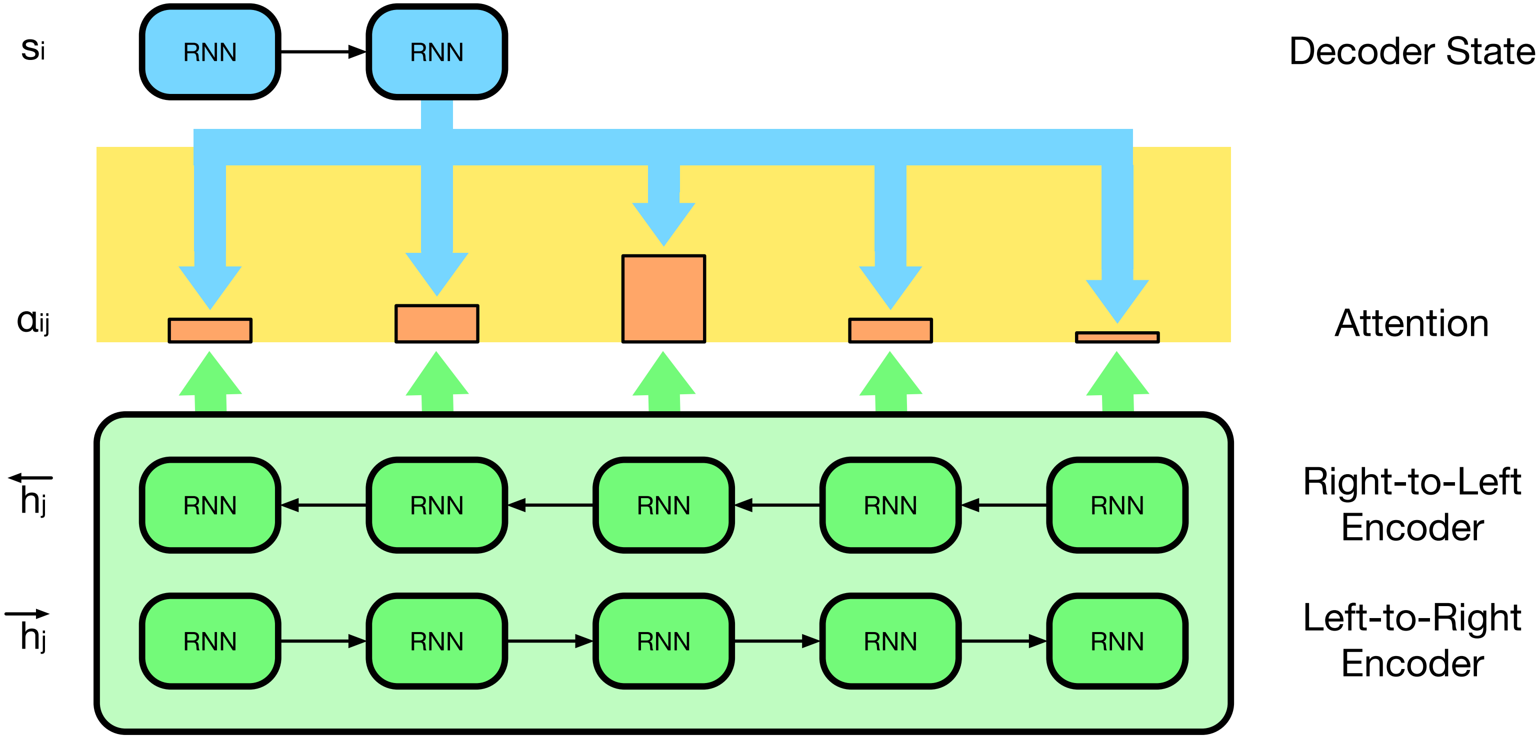
⇒ Attention mechanism

Attention



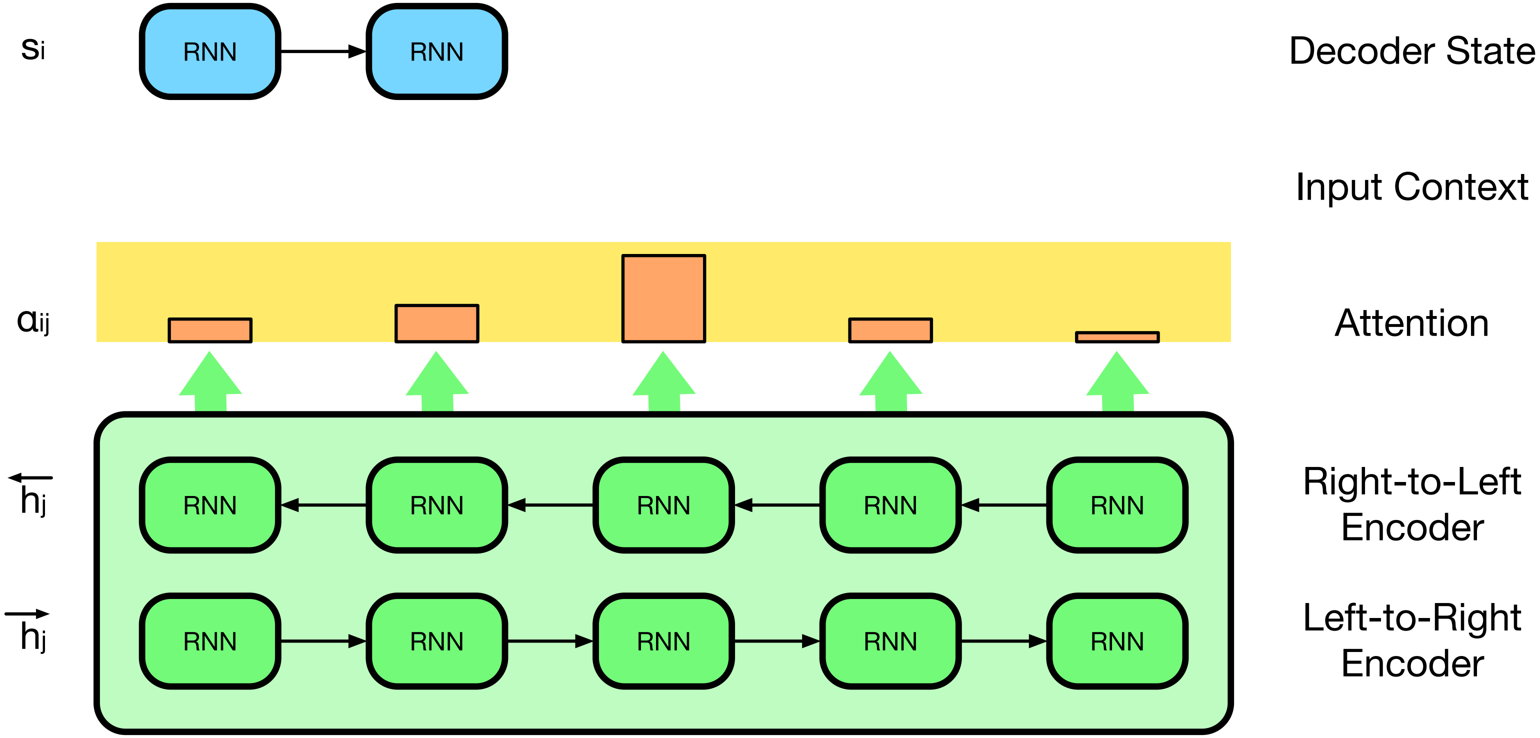
- Given what we have generated so far (decoder hidden state)
- ... which words in the input should we pay attention to (encoder states)?

Attention



- Given: – the previous hidden state of the decoder s_{i-1}
  – the representation of input words $h_j = (\overleftarrow{h_j}, \overrightarrow{h_j})$
- Predict an alignment probability $a(s_{i-1}, h_j)$ to each input word j (modeled with with a feed-forward neural network layer)

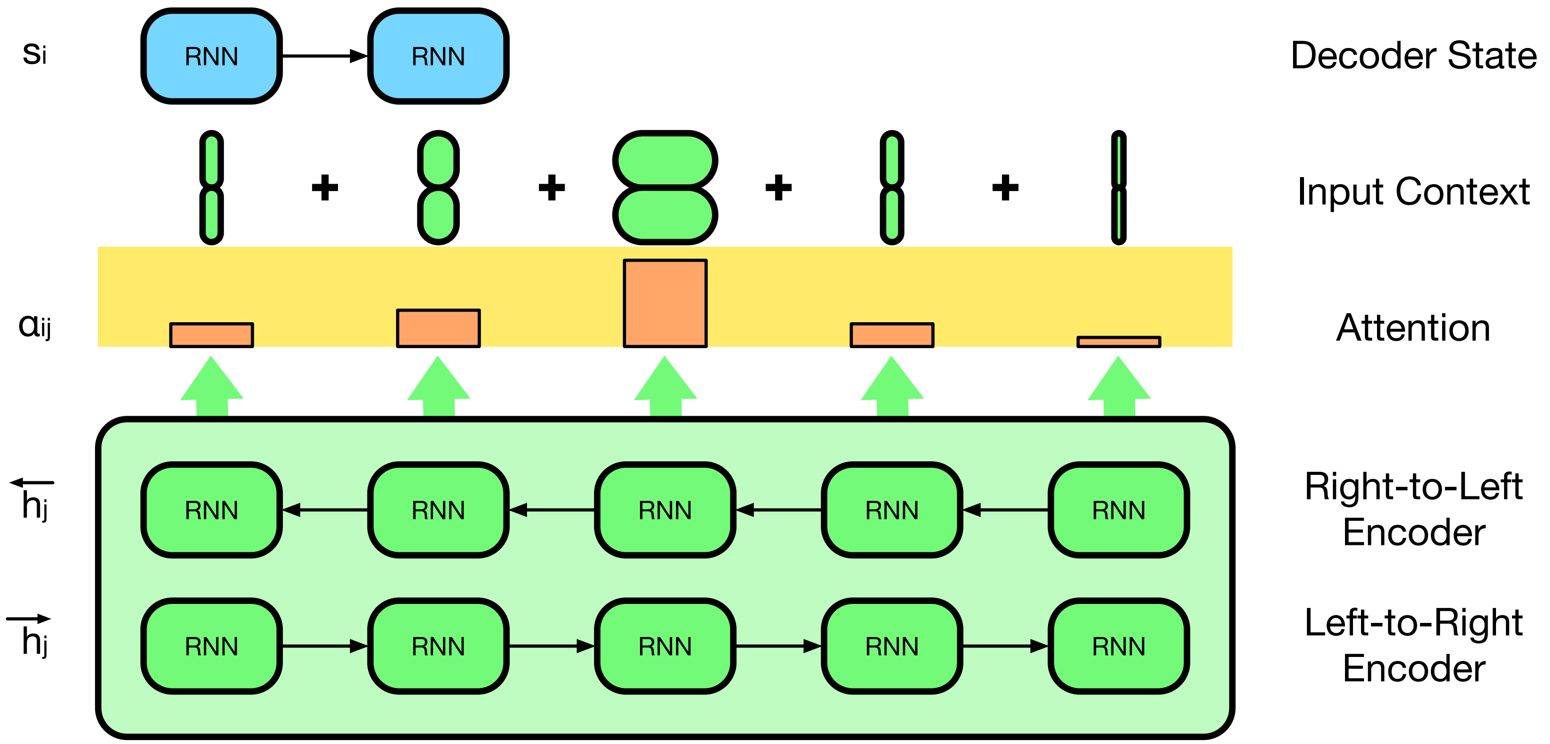
Attention



- Normalize attention (softmax)

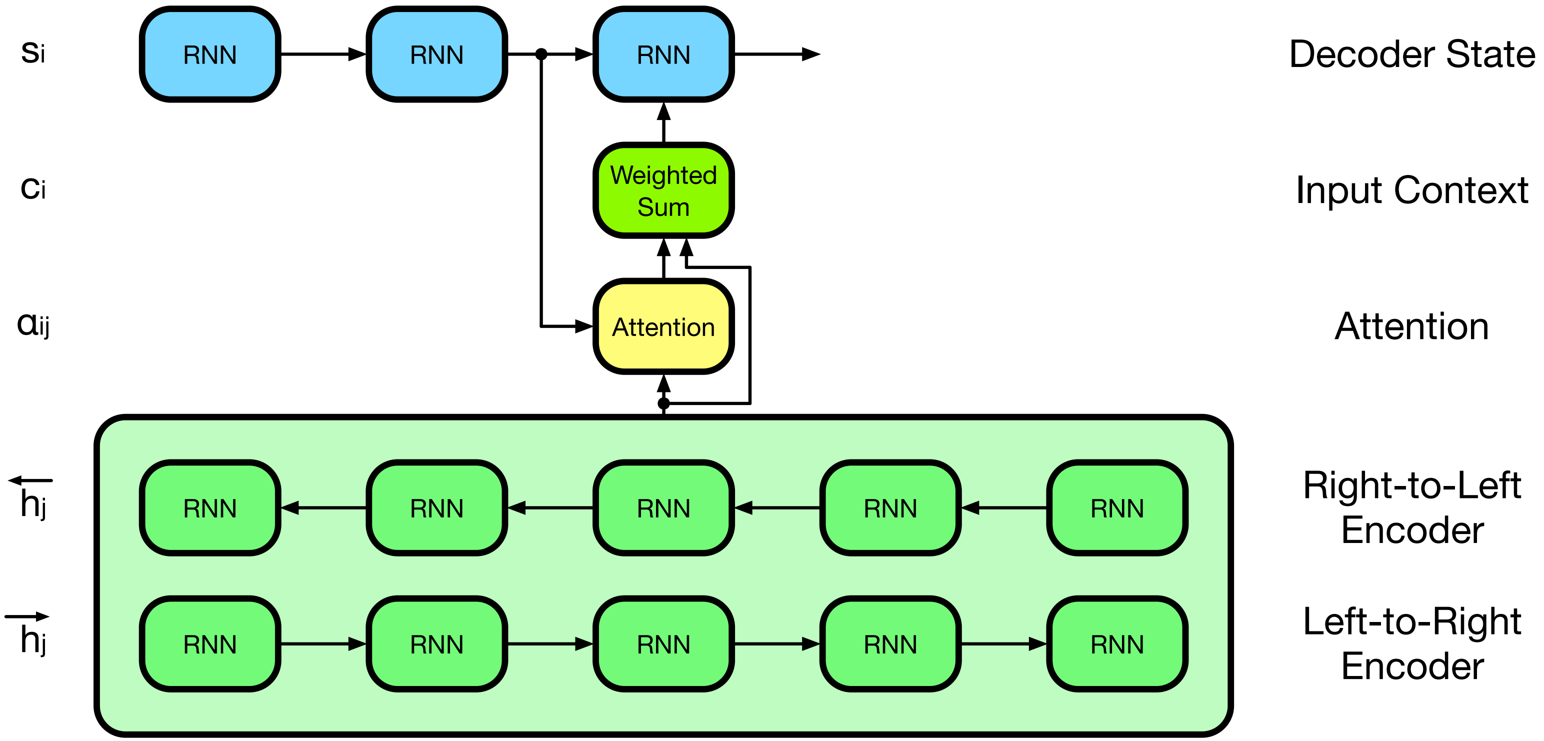
$$\alpha_{ij} = \frac{\exp(a(s_{i-1}, h_j))}{\sum_k \exp(a(s_{i-1}, h_k))}$$

Attention



- Relevant input context: weigh input words according to attention: $c_i = \sum_j \alpha_{ij} h_j$

Attention



- Use context to predict next hidden state and output word

Computing Attention

- Attention mechanism in neural translation model (Bahdanau et al., 2015)
  - previous hidden state s_{i-1}
  - input word embedding h_j
  - trainable parameters b, W_a, U_a, v_a

$$a(s_{i-1}, h_j) = v_a^T \tanh(W_a s_{i-1} + U_a h_j + b)$$

- Other ways to compute attention (Luong et al., 2015)
  - Dot product: $a(s_{i-1}, h_j) = s_{i-1}^T h_j$
  - Scaled dot product: $a(s_{i-1}, h_j) = \frac{1}{\sqrt{|h_j|}} s_{i-1}^T h_j$
  - General: $a(s_{i-1}, h_j) = s_{i-1}^T W_a h_j$
  - Local: $a(s_{i-1}) = W_a s_{i-1}$

General View of Dot-Product Attention

- Three elements

  Query : decoder state
  Key : encoder state
  Value : encoder state

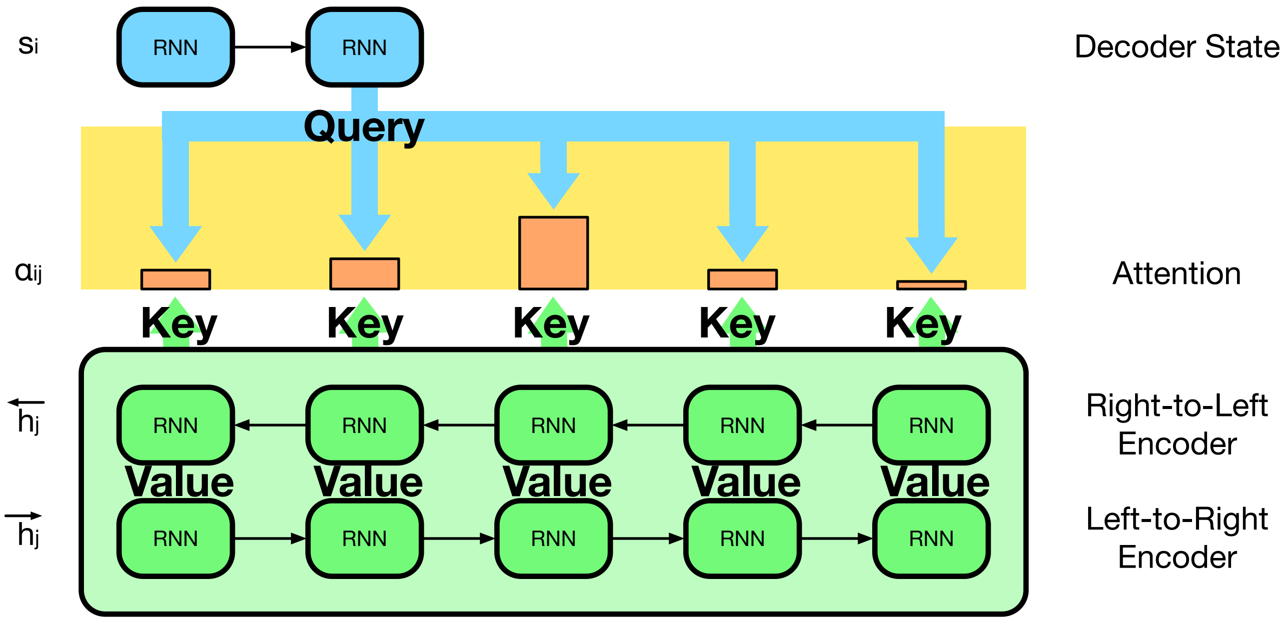
- Intuition
  - given a query (the decoder state)
  - we check how well it matches keys in the database (the encoder states)
  - and then use the matching score to scale the retrieved value (also the encoder state)

- Computation

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

General View of Dot-Product Attention

$$\text{Attention}(Q, K, V)$$



- Query: encoder state, Key and Value: decoder state

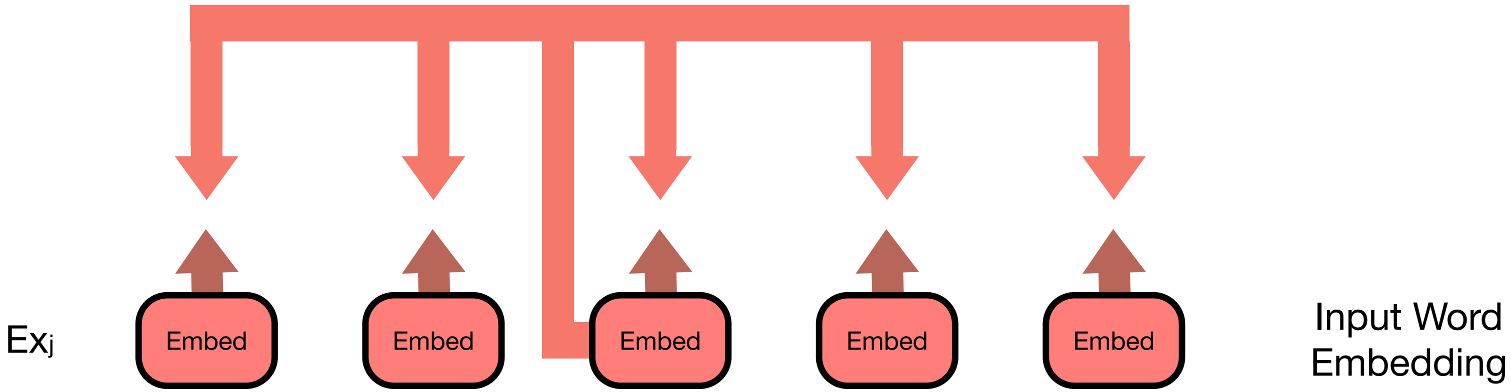
$$\text{Attention}(S, H, H)$$

Self Attention

- Finally, a very different take at attention
- Motivation so far: need for alignment between input words and output words
- Now: refine representation of input words in the encoder
  - representation of an input word mostly depends on itself
  - but also informed by the surrounding context
  - previously: recurrent neural networks (considers left or right context)
  - now: attention mechanism
- Self attention:

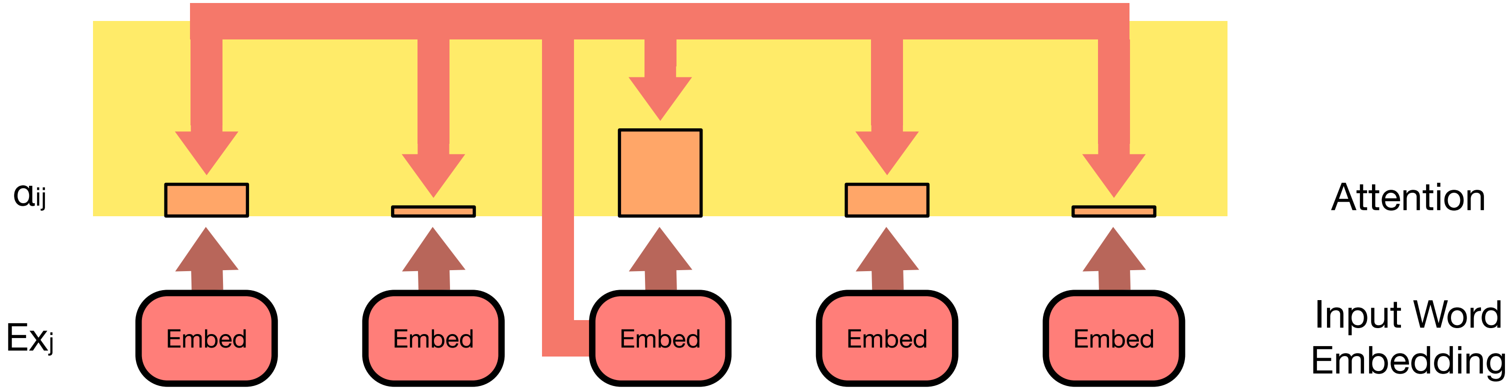
  Which of the surrounding words is most relevant to refine representation?

Self Attention



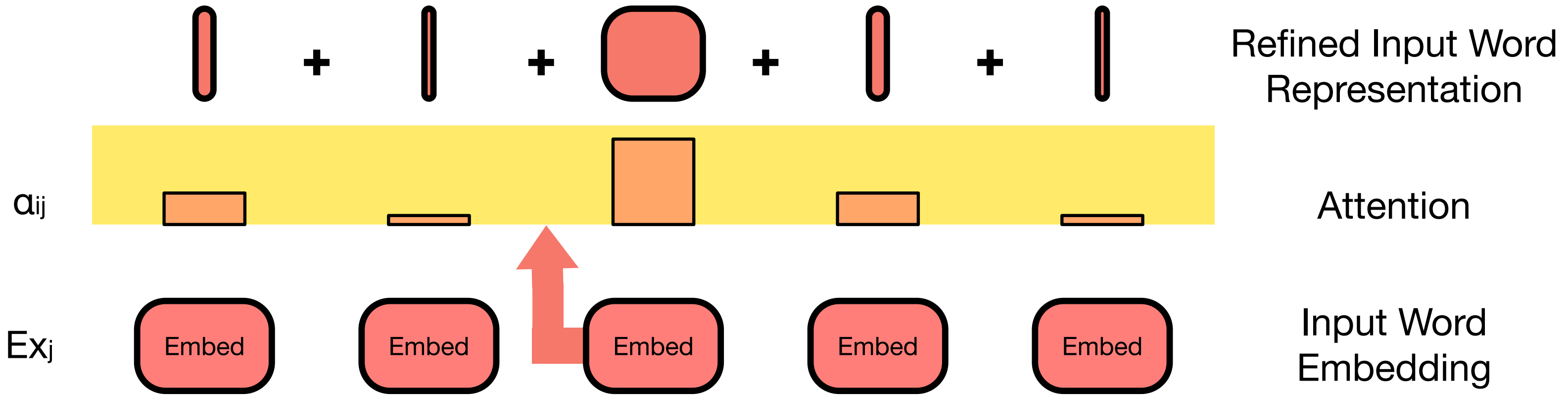
- Given: input word embeddings
- Task: consider how each should be refined in view of others
- Needed: how much attention to pay to others

Self Attention



- Computation of attention weights as before
  - Key: word embedding (or generally: encoder state for word H)
  - Query: word embedding (or generally: encoder state for word H)
- Again, multiple with weight matrices: Q=HW^Q and K=HW^K
- Attention weights: QK^T

Self Attention



- Full self attention

$$\text{self-attention}(H) = \text{Attention}(HW^Q, HW^K, H)$$

- Resulting vector uses weighted context words

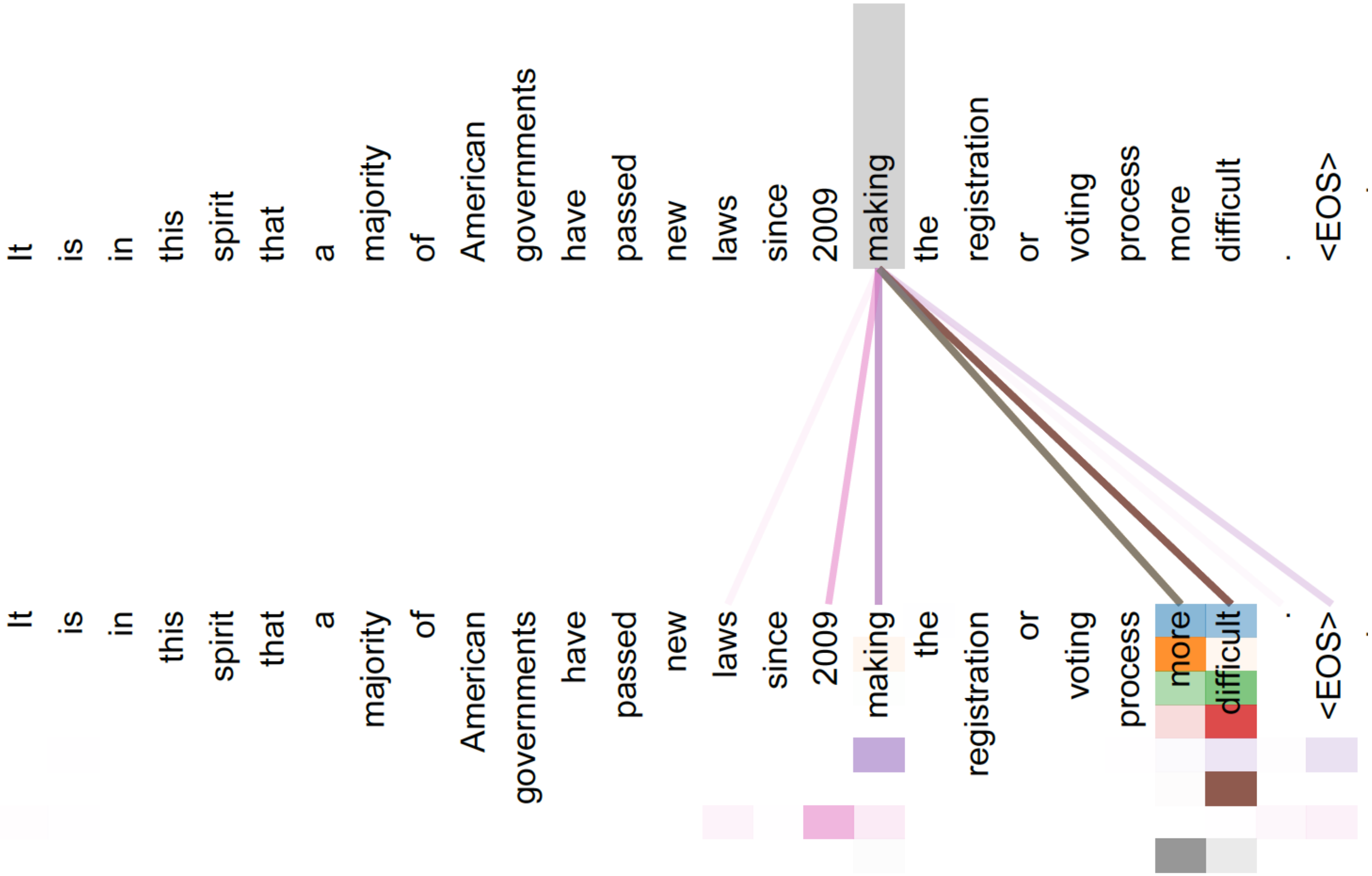
Multi-Head Attention

- Add redundancy
  - say, 16 attention weights
  - each based on its own parameters W_i^Q, W_i^K, W_i^V ▮
- Formally:

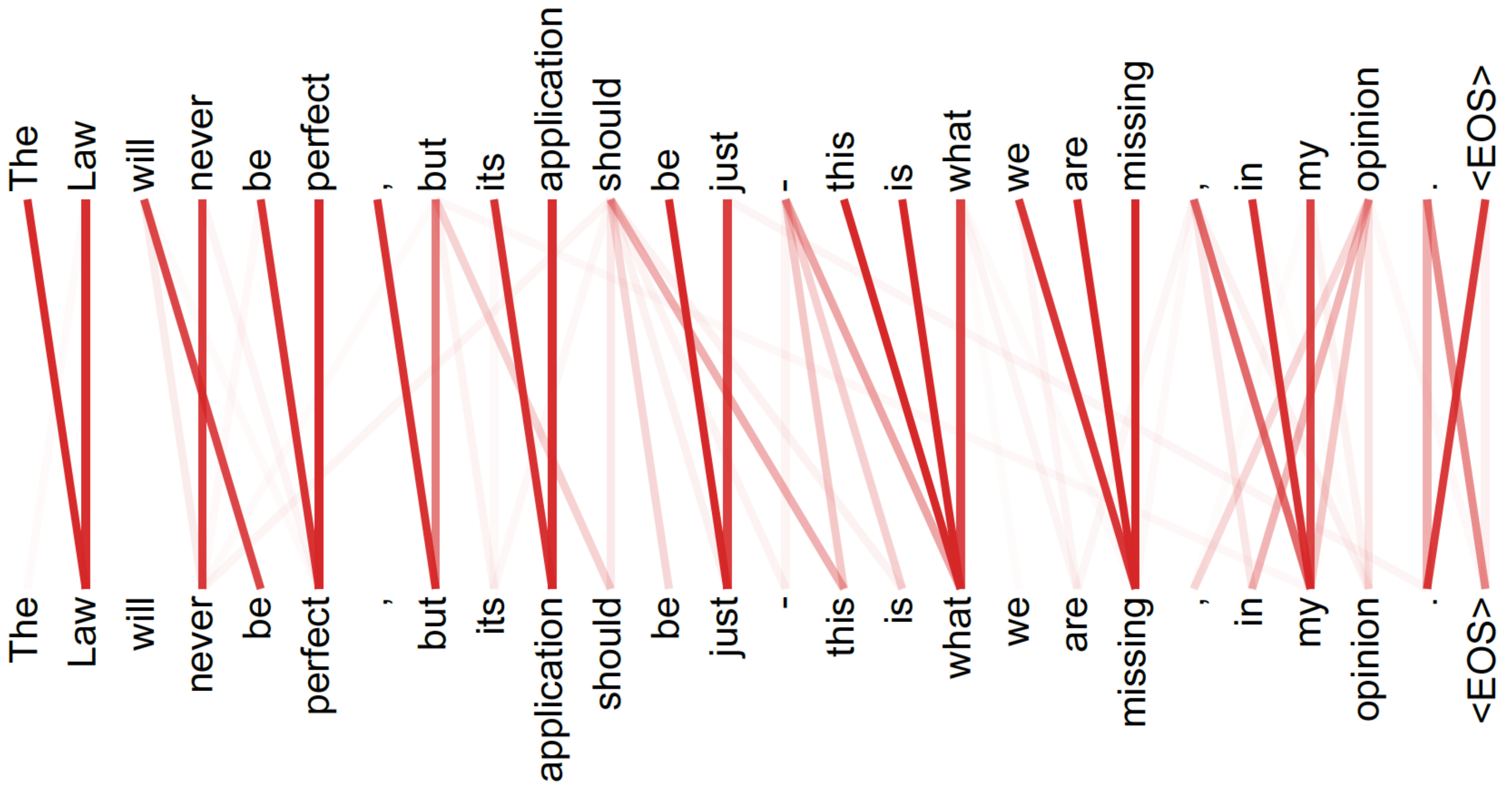
$$\begin{aligned}\text{head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \\ \text{MultiHead}(Q,K,V) &= \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O\end{aligned}$$

- Multi-head attention is a form of ensembling

Multi-Head Attention



Multi-Head Attention



"Many of the attention heads exhibit behaviour that seems related to the structure of the sentence."

transformer

Self Attention: Transformer

- Self-attention in encoder
  - refine word representation based on relevant context words
  - relevance determined by self attention

- Self-attention in decoder
  - refine output word predictions based on relevant previous output words
  - relevance determined by self attention

- Also regular attention to encoder states in decoder

- Currently most successful model

  (maybe only with self attention in decoder, but regular recurrent decoder)

Self Attention Layer

- Given: input word representations h_j, packed into a matrix $H = (h_1, ..., h_j)$
- Self attention

$$\text{self-attention}(H) = \text{MultiHead}(H, H, H)$$

- Shortcut connection

$$\text{self-attention}(h_j) + h_j$$

- Layer normalization

$$\hat{h}_j = \text{layer-normalization}(\text{self-attention}(h_j) + h_j)$$

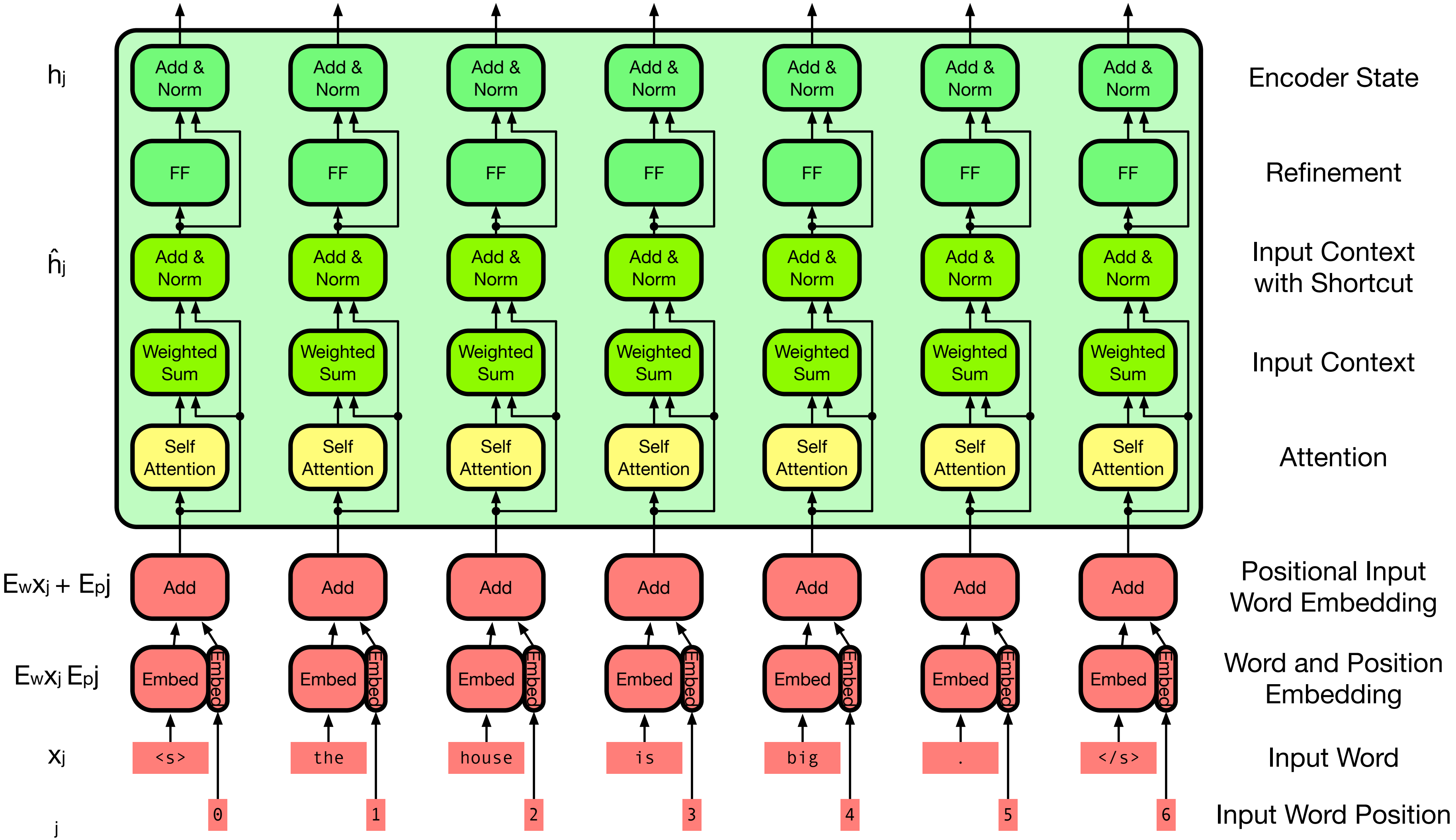
- Feed-forward step with ReLU activation function

$$\text{relu}(W\hat{h}_j + b)$$

- Again, shortcut connection and layer normalization

$$\text{layer-normalization}(\text{relu}(W\hat{h}_j + b) + \hat{h}_j)$$

Encoder



Sequence of self-attention layers

Self-Attention in the Decoder

- Same idea as in the encoder
- Output words are initially encoded by word embeddings $s_i = Ey_i$.
- Self attention is computed over previous output words
  - association of a word s_i is limited to words s_k $(k \leq i)$
  - resulting representation $\tilde{s}_i$

$$\text{self-attention}(\tilde{S}) = \text{MultiHead}(\tilde{S}, \tilde{S}, \tilde{S})$$

Attention in the Decoder

- Original intuition of attention mechanism: focus on relevant input words
- Computed with dot product $\tilde{S}H^T$
- Compute attention between the decoder states $\tilde{S}$ and the final encoder states H

$$\text{attention}(\tilde{S}, H) = \text{MultiHead}(\tilde{S}, H, H)$$

- Note: attention mechanism formally mirrors self-attention

Full Decoder

- Self-attention

$$\text{self-attention}(\tilde{S}) = \text{MultiHead}(\tilde{S}, \tilde{S}, \tilde{S})$$

  - shortcut connections
  - layer normalization

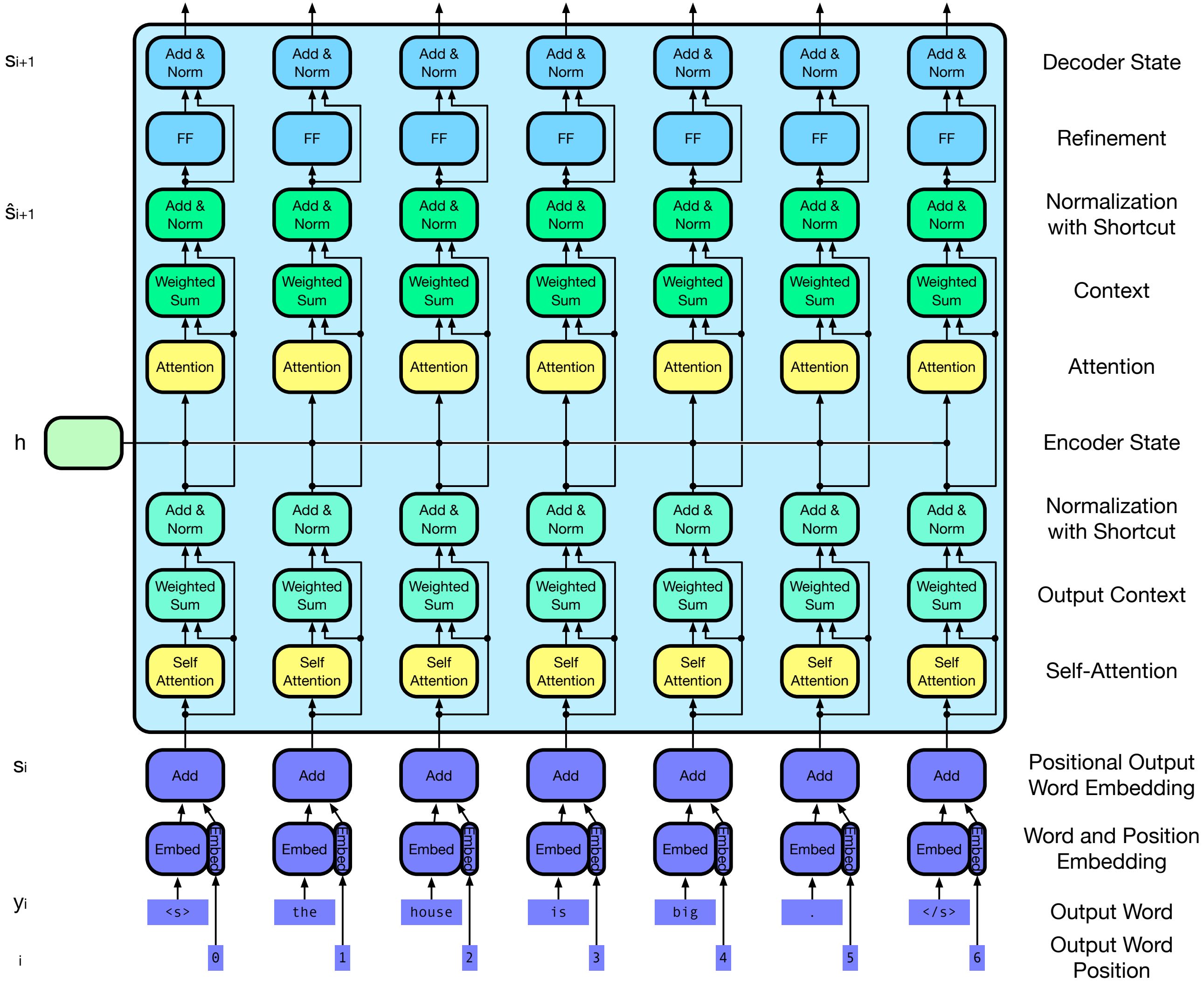
- Attention

$$\text{attention}(\tilde{S}, H) = \text{softmaxMultiHead}(\tilde{S}, H, H)$$

  - shortcut connections
  - layer normalization
  - feed-forward layer

- Multiple stacked layers

Decoder



Decoder computes attention-based representations of the output in several layers, initialized with the embeddings of the previous output words

Multiple Layers

- Stack several transformer layers (say, $D = 6$)
- Encoder
  - Start with input word embedding

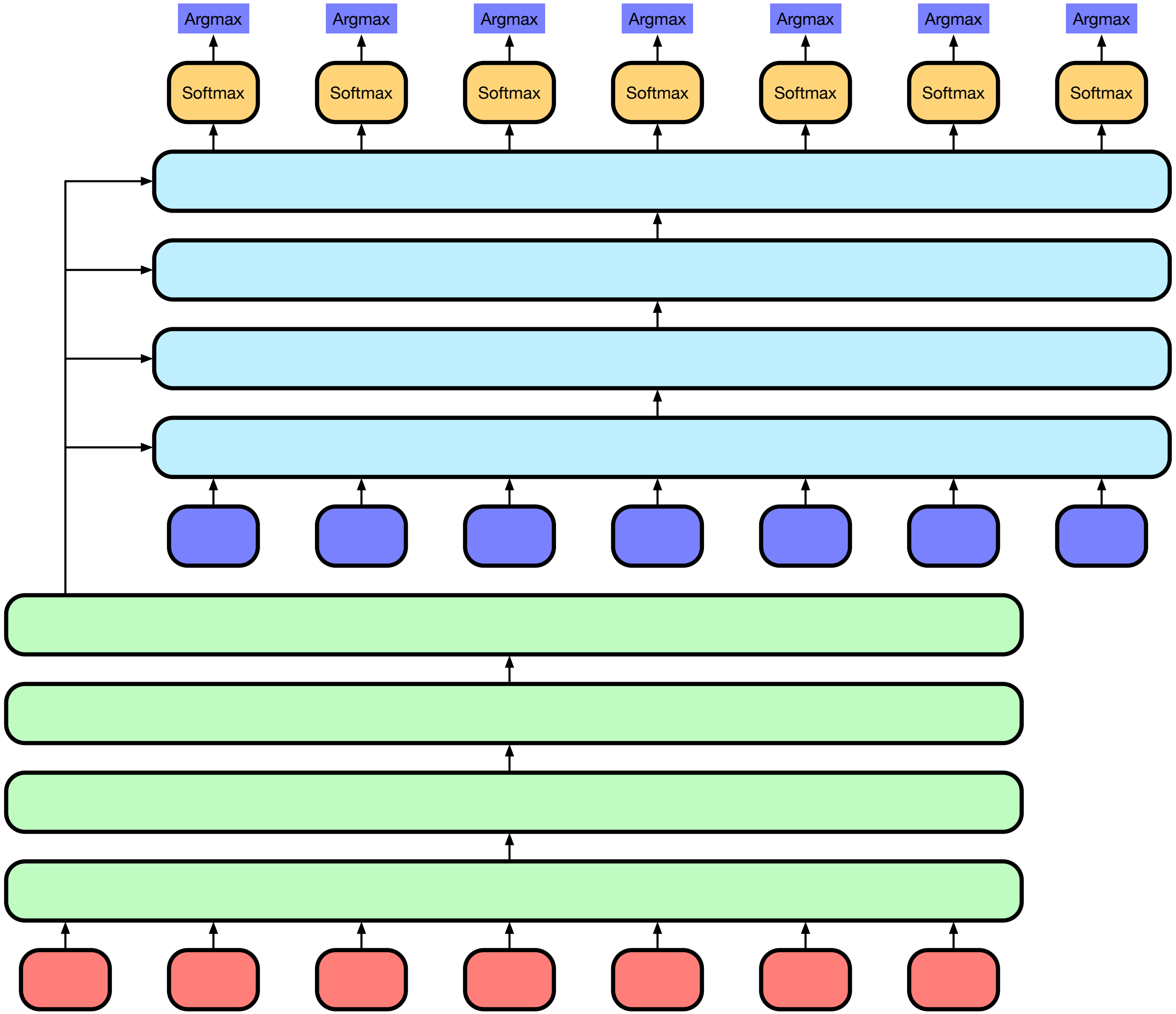
$$h_{0,j} = Ex_j$$

  - Stacked layers

$$h_{d,j} = \text{self-attention-layer}(h_{d-1,j})$$

- Same for decoder

Multiple Layers in Encoder and Decoder



Output Word

Output Word Prediction

Decoder Layer

Decoder Layer

Decoder Layer

Decoder Layer

Output Word Embedding

Encoder Layer

Encoder Layer

Encoder Layer

Encoder Layer

Input Word

Learning Rate

- Gradient computation gives direction of change

- Scaled by learning rate

- Weight updates

- Simplest form: fixed value

- Annealing
  - start with larger value (big changes at beginning)
  - reduce over time (minor adjustments to refine model)

Ensuring Randomness

- Typical theoretical assumption

  independent and identically distributed

  training examples

- Approximate this ideal
  - avoid undue structure in the training data
  - avoid undue structure in initial weight setting

- ML approach: Maximum entropy training
  - Fit properties of training data
  - Otherwise, model should be as random as possible (i.e., has maximum entropy)

Shuffling the Training Data

- Typical training data in machine translation
  - different types of corpora
    - European Parliament Proceedings
    - collection of movie subtitles
  - temporal structure in each corpus
  - similar sentences next too each other (e.g., same story / debate)

- Online updating: last examples matter more

- Convergence criterion: no improvement recently

  → stretch of hard examples following easy examples: prematurely stopped

⇒ randomly shuffle the training data

(maybe each epoch)

Weight Initialization

- Initialize weights to random values
- Values are chosen from a uniform distribution
- Ideal weights lead to node values in transition area for activation function

For Example: Sigmoid

- Input values in range $[-1;1]$

⇒ Output values in range [0.269;0.731]

- Magic formula (n size of the previous layer)

$$\Big[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\Big]$$

- Magic formula for hidden layers

$$\Big[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\Big]$$

  - n_j is the size of the previous layer
  - n_{j+1} size of next layer

Problem: Overconfident Models

- Predictions of the neural machine translation models are surprisingly confident

- Often almost all the probability mass is assigned to a single word
  (word prediction probabilities of over 99%)

- Problem for decoding and training
  - decoding: sensible alternatives get low scores, bad for beam search
  - training: overfitting is more likely

- Solution: label smoothing

- Jargon notice
  - in classification tasks, we predict a *label*
  - jargon term for any output

  → here, we smooth the word predictions

Label Smoothing during Decoding

- Common strategy to combat peaked distributions: smooth them

- Recall
  - prediction layer produces numbers for each word
  - converted into probabilities using the softmax

$$p(y_i) = \frac{\exp s_i}{\sum_j \exp s_j}$$

- Softmax calculation can be smoothed with so-called **temperature** T

$$p(y_i) = \frac{\exp s_i/T}{\sum_j \exp s_j/T}$$

- Higher temperature → distribution smoother
  (i.e., less probability is given to most likely choice)

Label Smoothing during Training

- Root of problem: training
- Training object: assign all probability mass to single correct word ▮
- Label smoothing
  - truth gives some probability mass to other words (say, 10% of it)
  - uniformly distributed over all words
  - relative to unigram word probabilities
    (relative counts of each word in the target side of the training data)

adjusting the learning rate

Adjusting the Learning Rate

- Gradient descent training: weight update follows the gradient downhill

- Actual gradients have fairly large values, scale with a learning rate (low number, e.g., $\mu = 0.001$)

- Change the learning rate over time
  - starting with larger updates
  - refining weights with smaller updates
  - adjust for other reasons

- Learning rate schedule

Momentum Term

- Consider case where weight value far from optimum

- Most training examples push the weight value in the same direction

- Small updates take long to accumulate

- Solution: momentum term m_t
  - accumulate weight updates at each time step t
  - some decay rate for sum (e.g., 0.9)
  - combine momentum term m_{t-1} with weight update value Δw_t

$$m_t = 0.9 m_{t-1} + \Delta w_t$$

$$w_t = w_{t-1} - \mu \, m_t$$

Adapting Learning Rate per Parameter

- Common strategy: reduce the learning rate μ over time
- Initially parameters are far away from optimum → change a lot
- Later nuanced refinements needed → change little

- Now: different learning rate for each parameter

Adagrad

- Different parameters at different stages of training
  → different learning rate for each parameter

- Adagrad
  - record gradients for each parameter
  - accumulate their square values over time
  - use this sum to reduce learning rate

- Update formula
  - gradient $g_t = \frac{dE_t}{dw}$ of error E with respect to weight w
  - divide the learning rate μ by accumulated sum

$$\Delta w_t = \frac{\mu}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}} g_t$$

- Big changes in the parameter value (corresponding to big gradients g_t)
  → reduction of the learning rate of the weight parameter

Adam: Elements

- Combine idea of momentum term and reduce parameter update by accumulated change

- Momentum term idea (e.g., $\beta_1 = 0.9$)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

- Accumulated gradients (decay with $\beta_2 = 0.999$)

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Adam: Technical Correction

- Initially, values for m_t and v_t are close to initial value of 0

- Adjustment

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- With $t \rightarrow \infty$ this correction goes away

$$\lim_{t \rightarrow \infty} \frac{1}{1 - \beta^t} \rightarrow 1$$

Adam

- Given
  - learning rate μ
  - momentum $\hat{m}_t$
  - accumulated change $\hat{v}_t$

- Weight update per Adam (e.g., $\epsilon = 10^{-8}$)

$$\Delta w_t = \frac{\mu}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Batched Gradient Updates

- Accumulate all weight updates for all the training example → update (converges slowly)

- Process each training example → update (stochastic gradient descent) (quicker convergence, but last training disproportionately higher impact)

- Process data in batches
  - compute all their gradients for individual word predictions errors
  - use sum over each batch to update parameters

  → better parallelization on GPUs

- Process data on multiple compute cores
  - batch processing may take different amount of time
  - asynchronous training: apply updates when they arrive
  - mismatch between original weights and updates may not matter much

avoiding local optima

Avoiding Local Optima

- One of hardest problem for designing neural network architectures and optimization methods
- Ensure that model converges to at least to a set of parameter values that give results close to this optimum on unseen test data.
- There is no real solution to this problem.
- It requires experimentation and analysis that is more craft than science.
- Still, this section presents a number of methods that generally help avoiding getting stuck in local optima.

Overfitting and Underfitting

- Neural machine translation models
  - 100s of millions of parameters
  - 100s of millions of training examples (individual word predictions)

- No hard rules for relationship between these two numbers

- Too many parameters and too few training examples → overfitting

- Too few parameters and many training examples → underfitting

Regularization

- Motivation: prefer as few parameters as possible

- Strategy: set un-needed paramters a value of 0

- Method
  - adjust training objective
  - add cost for any non-zero parameter
  - typically done with L2 norm

- Practical impact
  - derivative of L2 norm is value of parameter
  - if not signal from training: reduce value of parameter
  - alsp called weight decay

- Not common in deep learning, but other methods understood as regularization

Curriculum Learning

- Human learning
  - learn simple concepts first
  - learn more complex material later
- Early epochs: only easy training examples
  - only short sentences
  - create artificial data by extracting smaller segments (similar to phrase pair extraction in statistical machine translation)
  - Later epochs: all training data
- Not easy to callibrate

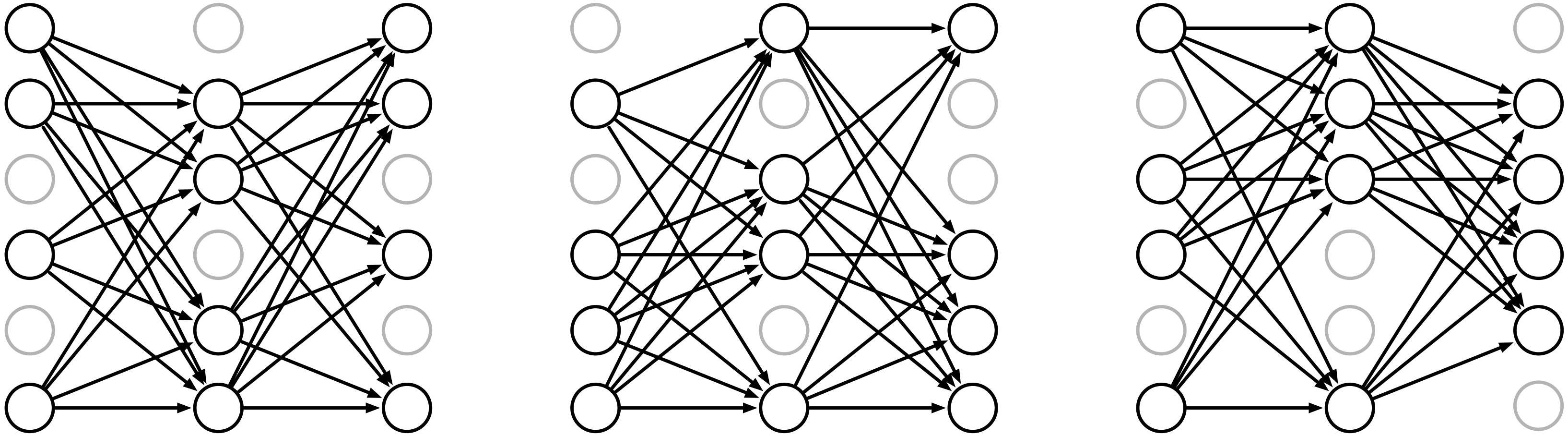
Dropout

- Training may get stuck in local optima
  - some properties of task have been learned
  - discovery of other properties would take it too far out of its comfort zone.

- Machine translation example
  - model learned the language model aspects
  - but cannot figure out role of input sentence

- Drop out: for each batch, eliminate some nodes

Dropout



- Dropout
  - For each batch, different random set of nodes is removed
  - Their values are set to 0 and their weights are not updated
  - 10%, 20% or even 50% of all the nodes

- Why does this work?
  - robustness: redundant nodes play similar nodes
  - ensemble learning: different subnetworks are different models

Gradient Clipping

- Exploding gradients: gradients become too large during backward pass

⇒ Limit total value of gradients for a layer to threshold (τ)

- Use of L2 norm of gradient values g

$$L2(g) = \sqrt{\sum_j g_j^2}$$

- Adjust each gradient value g_i for each element i in the vector

$$g'_i = g_i \times \frac{\tau}{\max(\tau, L2(g))}$$

Layer Normalization

- During inference, average node values may become too large or too small
- Has also impact on training (gradients are multiplied with node values)

⇒ Normalize node values

- During training, learn bias layer

Layer Normalization: Math

- Feed-forward layer h^l, weights W, computed sum s^l, activation function

$$s^l = W\ h^{l-1}$$

$$h^l = \text{sigmoid}(h^l)$$

- Compute mean μ^l and variance σ^l of sum vector s^l

$$\mu^l = \frac{1}{H} \sum_{i-1}^{H} s_i^l$$

$$\sigma^l = \sqrt{\frac{1}{H} \sum_{i-1}^{H} (s_i^l - \mu^l)^2}$$

Layer Normalization: Math

- Normalize s^l

$$\hat{s^l} = \frac{1}{\sigma^l}(s^l - \mu^l)$$

- Learnable bias vectors g and b

$$\hat{s^l} = \frac{g}{\sigma^l}(s^l - \mu^l) + b$$

Shortcuts and Highways

- Deep learning: many layers of processing

⇒ Error propagation has to travel farther

- All parameters in processing change have to be adjusted

- Instead of always passing through all layers, add connections from first to last

- Jargon alert
  - shortcuts
  - residual connections
  - skip connections

Shortcuts

- Feed-forward layer

$$y = f(x)$$

- Pass through input x

$$y = f(x) + x$$

- Note: gradient is

$$y' = f'(x) + 1$$

- Constant 1 → gradient is passed through unchanged

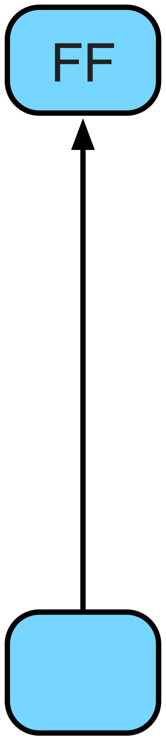
Highways

- Regulate how much information from $f(x)$ and x should impact the output y
- Gate $t(x)$ (typically computed by a feed-forward layer)

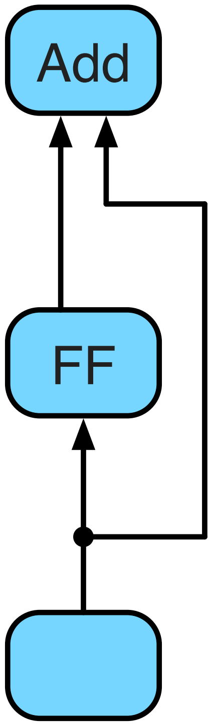
$$y = t(x)\ f(x) + (1 - t(x))\ x$$

Shortcuts and Highways

Basic Layer



Skip Connection



Highway Network