Natural Language Processing with Deep Learning

CS224N/Ling284

Lecture 13:
Transformer Networks and
Convolutional Neural Networks

Richard Socher

Announcements

- Projects Project advice OH. Come every week from now on.

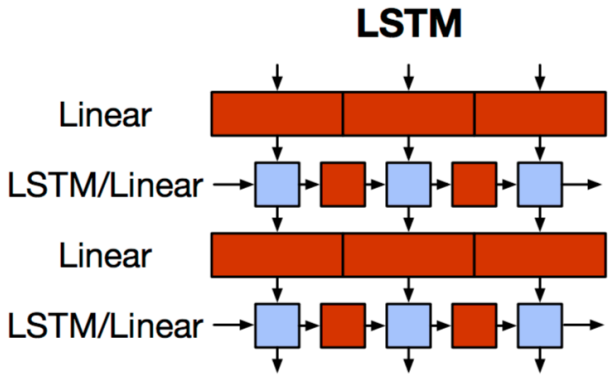
Overview of today

- Attention is all you need? – Transformer Networks

- CNN Variant 1: Simple single layer
- Application: Sentence classification
- Some practical details and tricks for CNNs

Problems with RNNs = Motivation for Transformers

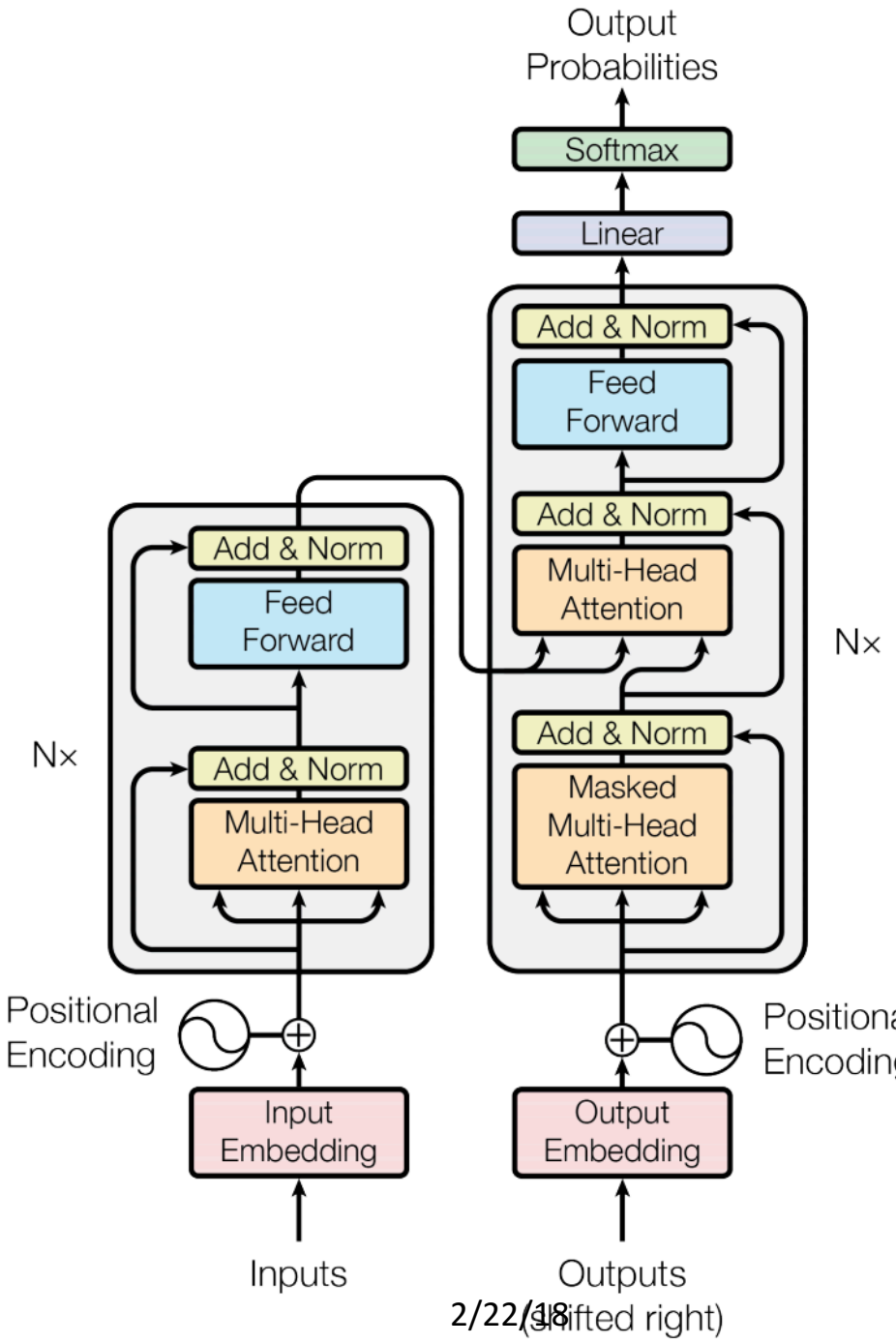
- Sequential computation prevents parallelization



- Despite GRUs and LSTMs, RNNs still need attention mechanism to deal with long range dependencies – path length for co-dependent computation between states grows with sequence
- But if attention gives us access to any state… maybe we don't need the RNN? 🤔

Transformer Overview

- Sequence-to-sequence
- Encoder-Decoder
- Task: machine translation with parallel corpus
- Predict each translated word
- Final cost/error function is standard cross-entropy error on top of a softmax classifier



This and related figures from paper:
https://arxiv.org/pdf/1706.03762.pdf

Transformer Paper

- Attention Is All You Need [2017]
- by Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin
- Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention and started the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first Transformer models and has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention and the parameter-free position representation and became the other person involved in nearly every detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase and tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial codebase, and efficient inference and visualizations. Lukasz and Aidan spent countless long days designing various parts of and implementing tensor2tensor, replacing our earlier codebase, greatly improving results and massively accelerating our research.

Transformer Basics

- Let's define the basic building blocks of transformer networks first: new attention layers!

Dot-Product Attention (Extending our previous def.)

- Inputs: a query q and a set of key-value (k-v) pairs to an output
- Query, keys, values, and output are all vectors

- Output is weighted sum of values, where
- Weight of each value is computed by an inner product of query and corresponding key
- Queries and keys have same dimensionality d_k value have d_v

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

Dot-Product Attention – Matrix notation

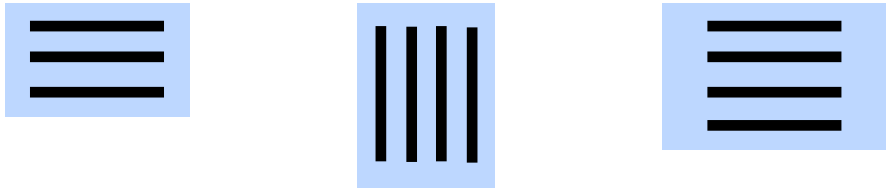
- When we have multiple queries q, we stack them in a matrix Q:

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

- Becomes: $A(Q, K, V) = softmax(QK^T)V$

$[|Q| \text{ x } d_k] \text{ x } [d_k \text{ x } |K|] \text{ x } [|K| \text{ x } d_v]$

softmax row-wise



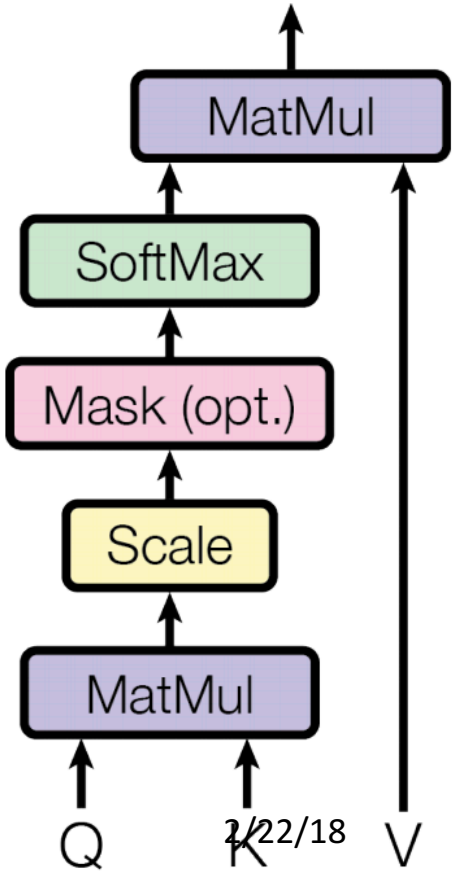
$= [|Q| \text{ x } d_v]$

Scaled Dot-Product Attention

- Problem: As d_k gets large, the variance of $q^T k$ increases → some values inside the softmax get large → the softmax gets very peaked --> hence its gradient gets smaller.

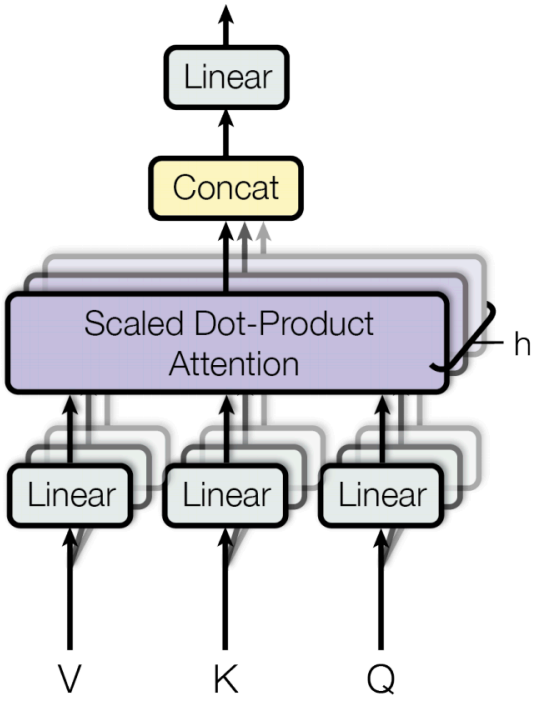
- Solution: Scale by length of query/key vectors:

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



Self-attention and Multi-head attention

- The input word vectors could be the queries, keys and values
- In other words: the word vectors themselves select each other
- Word vector stack = Q = K = V
- Problem: Only one way for words to interact with one-another
- Solution: Multi-head attention
- First map Q, K, V into h many lower dimensional spaces via W matrices
- Then apply attention, then concatenate outputs and pipe through linear layer

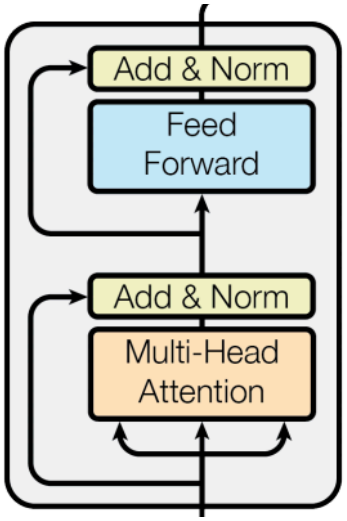


$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Complete transformer block

- Each block has two “sublayers”

1. Multihead attention
2. 2 layer feed-forward Nnet (with relu)



Each of these two steps also has:

Residual (short-circuit) connection and LayerNorm:

LayerNorm(x + Sublayer(x))

Layernorm changes input to have mean 0 and variance 1,
per layer and per training point (and adds two more parameters)

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2} \qquad h_i = f\left(\frac{g_i}{\sigma_i}\left(a_i - \mu_i\right) + b_i\right)$$

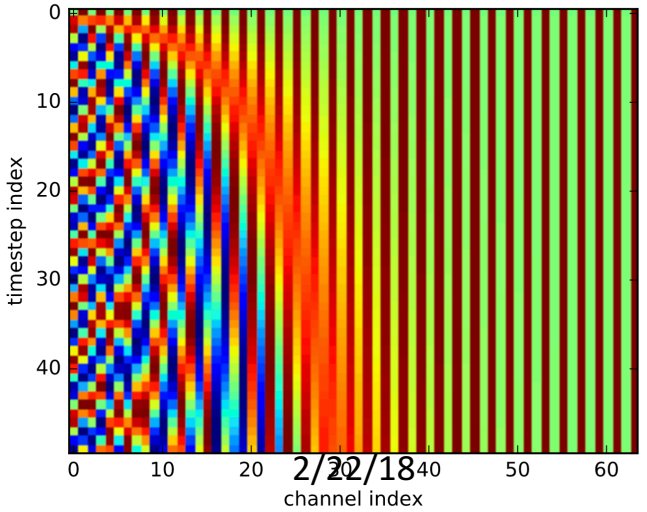
Layer Normalization by Ba, Kiros and Hinton, https://arxiv.org/pdf/1607.06450.pdf

Encoder Input

- Actual word representations are byte-pair encodings (see last lecture)

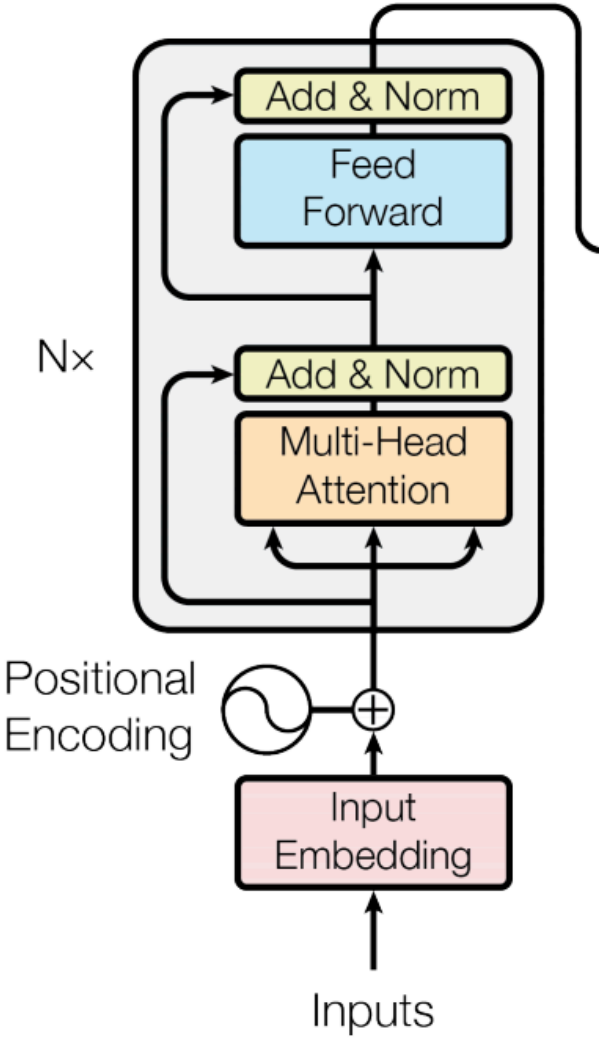
- Also added is a positional encoding so same words at different locations have different overall representations:

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



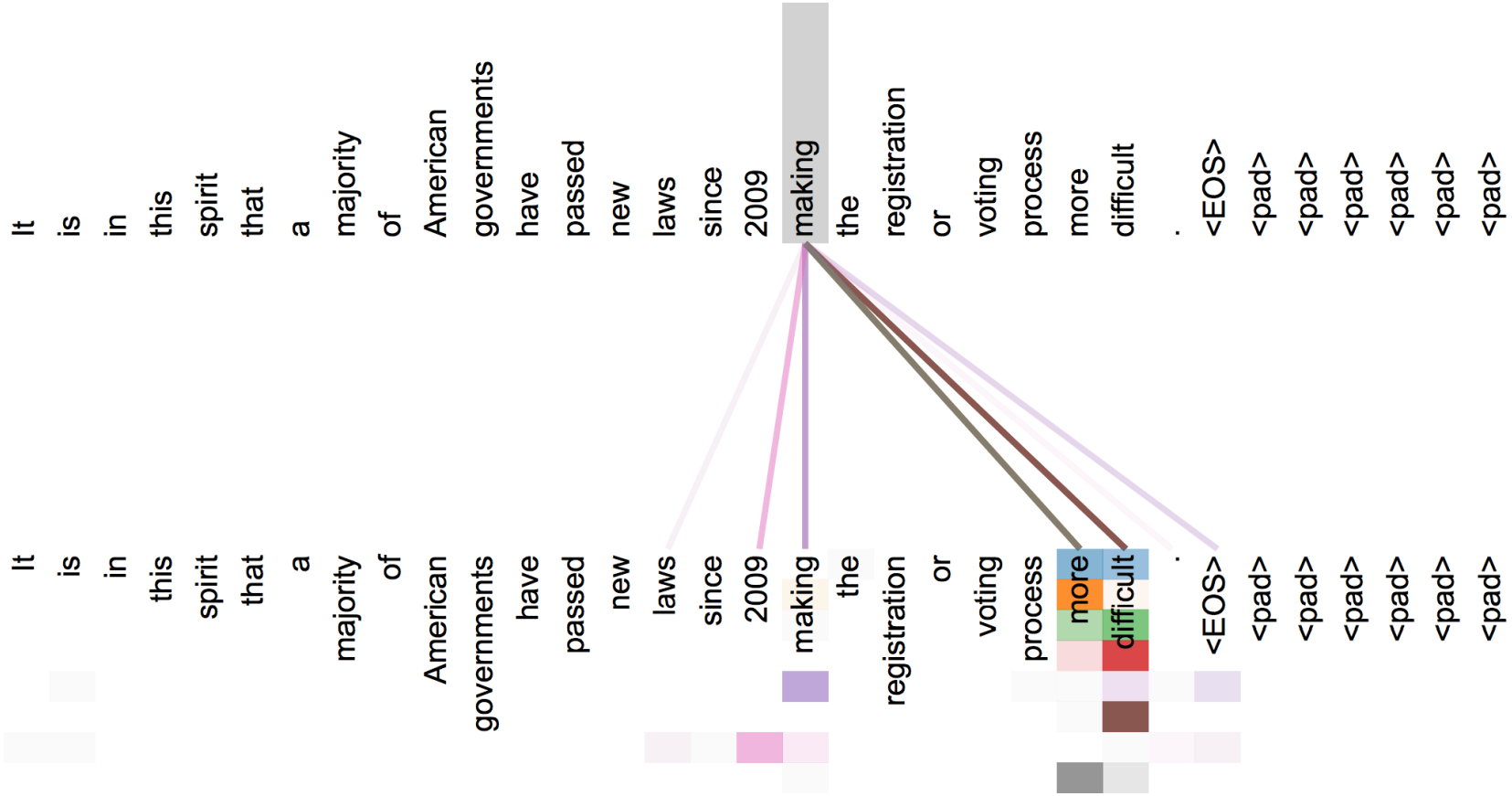
Complete Encoder

- For encoder, at each block, we use the same Q, K and V from the previous layer
- Blocks are repeated 6 times

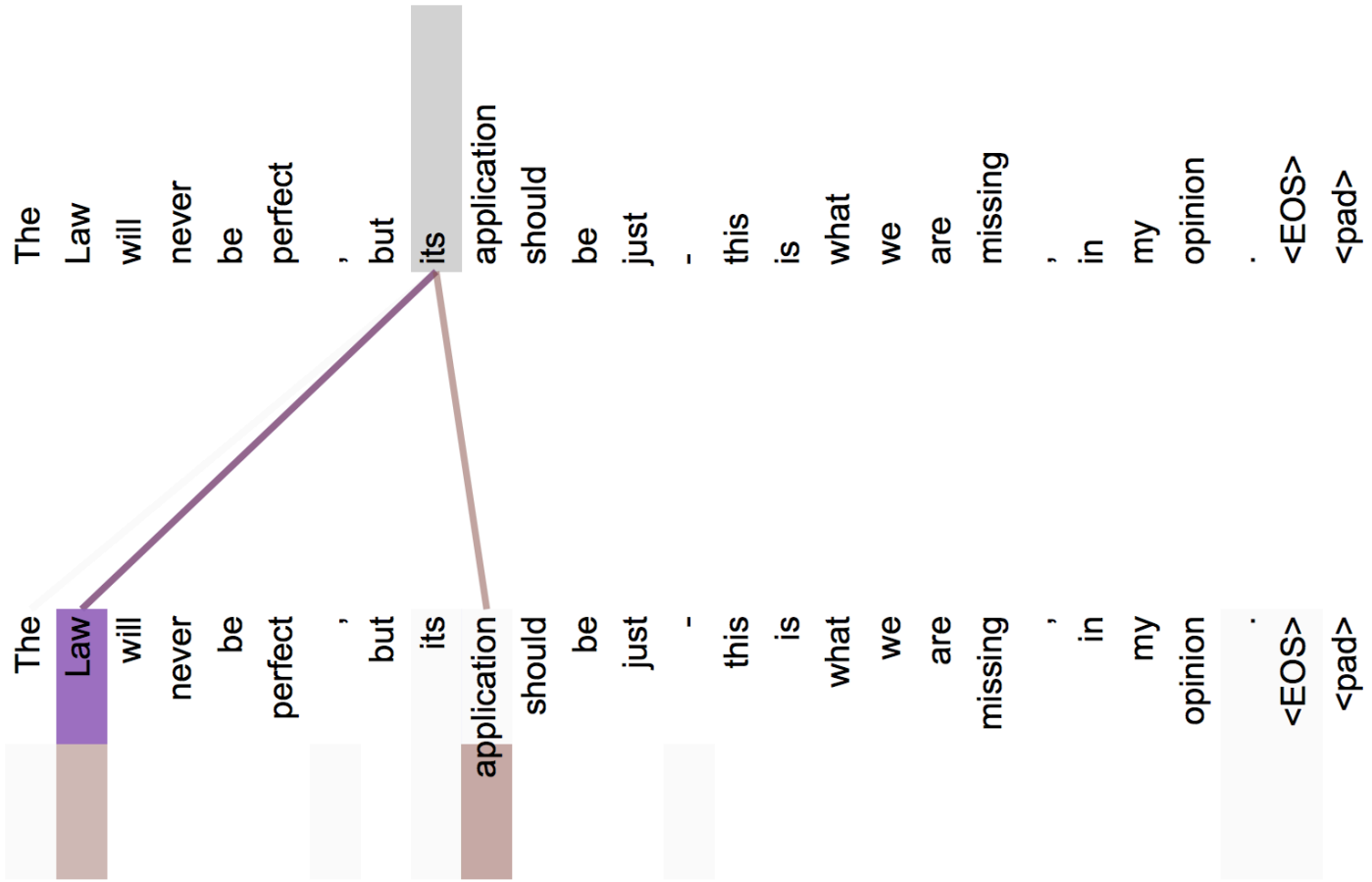


Attention visualization in layer 5

- Words start to pay attention to other words in sensible ways



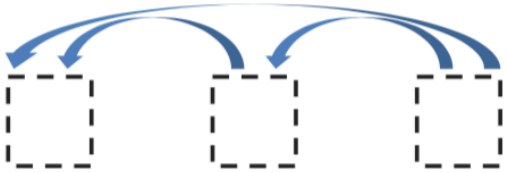
Attention visualization: Implicit anaphora resolution



In 5th layer. Isolated attentions from just the word 'its' for attention heads 5 and 6. Note that the attentions are very sharp for this word.

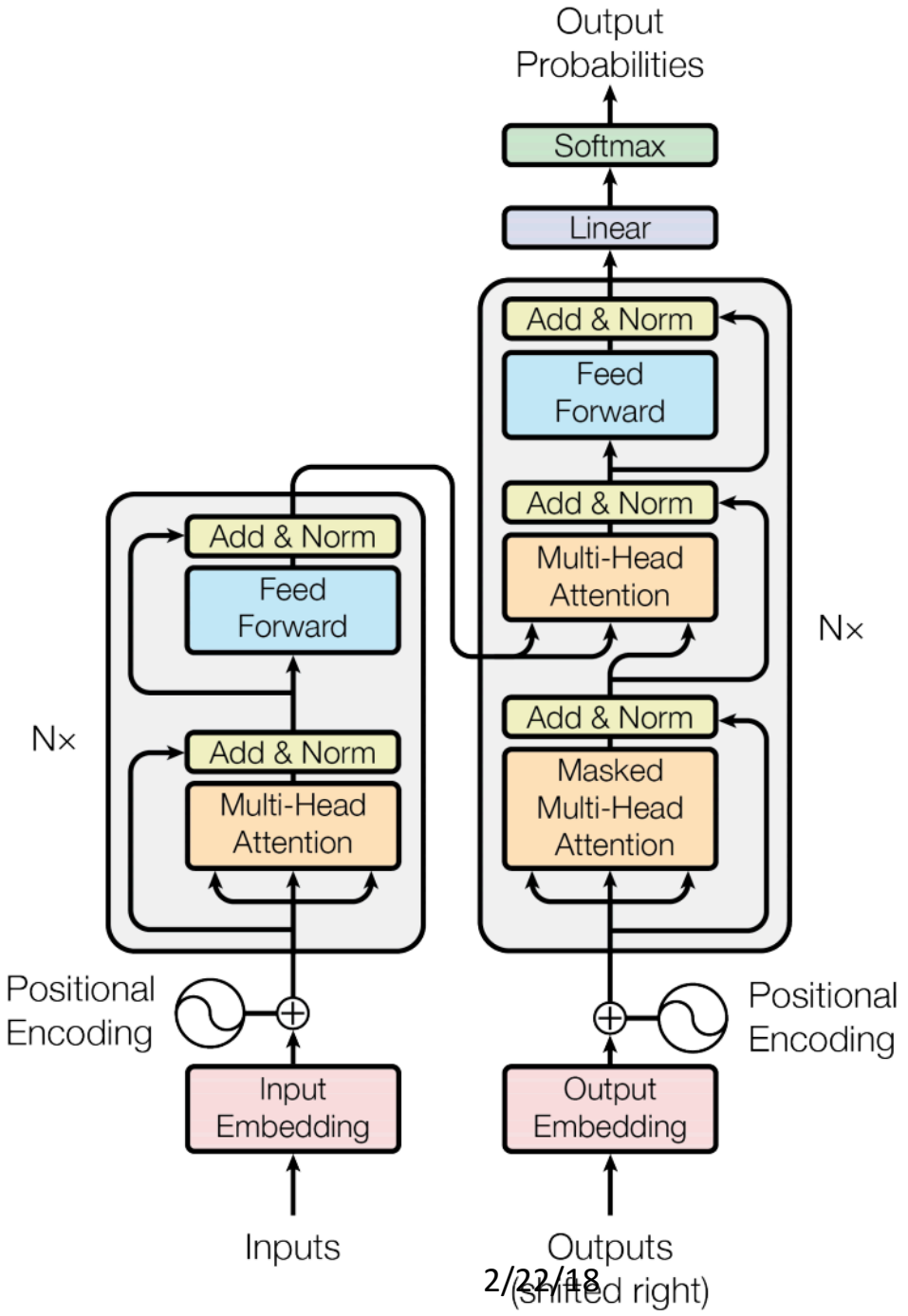
Transformer Decoder

- 2 sublayer changes in decoder
- Masked decoder self-attention on previously generated outputs:



- Encoder-Decoder Attention, where queries come from previous decoder layer and keys and values come from output of encoder

- Blocks repeated 6 times also



Tips and tricks of the Transformer

Details in paper and later lectures:

- Byte-pair encodings
- Checkpoint averaging
- ADAM optimizer with learning rate changes
- Dropout during training at every layer just before adding residual
- Label smoothing
- Auto-regressive decoding with beam search and length penalties

- → Overall, they are hard to optimize and unlike LSTMs don't usually work out of the box and don't play well yet with other building blocks on tasks.

Experimental Results for MT

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Experimental Results for Parsing

Parser	Training	WSJ 23 F1
Vinyals & Kaiser el al. (2014) [37]	WSJ only, discriminative	88.3
Petrov et al. (2006) [29]	WSJ only, discriminative	90.4
Zhu et al. (2013) [40]	WSJ only, discriminative	90.4
Dyer et al. (2016) [8]	WSJ only, discriminative	91.7
Transformer (4 layers)	WSJ only, discriminative	91.3
Zhu et al. (2013) [40]	semi-supervised	91.3
Huang & Harper (2009) [14]	semi-supervised	91.3
McClosky et al. (2006) [26]	semi-supervised	92.1
Vinyals & Kaiser el al. (2014) [37]	semi-supervised	92.1
Transformer (4 layers)	semi-supervised	92.7
Luong et al. (2015) [23]	multi-task	93.0
Dyer et al. (2016) [8]	generative	93.3

CNNs

Convolutional Neural Networks (CNNs)

- Main CNN idea: What if we compute multiple vectors for every possible phrase in parallel?

- Regardless of whether phrase is grammatical
- Example: “the country of my birth” computes vectors for:
  - the country, country of, of my, my birth, the country of, country of my, of my birth, the country of my, country of my birth

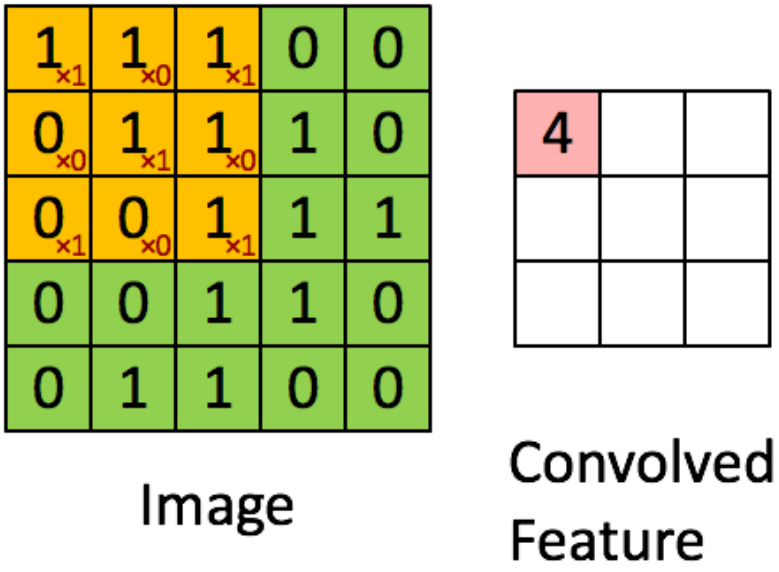
- Then group them afterwards (more soon)

- Not very linguistically or cognitively plausible but very fast!

What is convolution anyway?

- 1d discrete convolution generally: $(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m].$
- Convolution is great to extract features from images

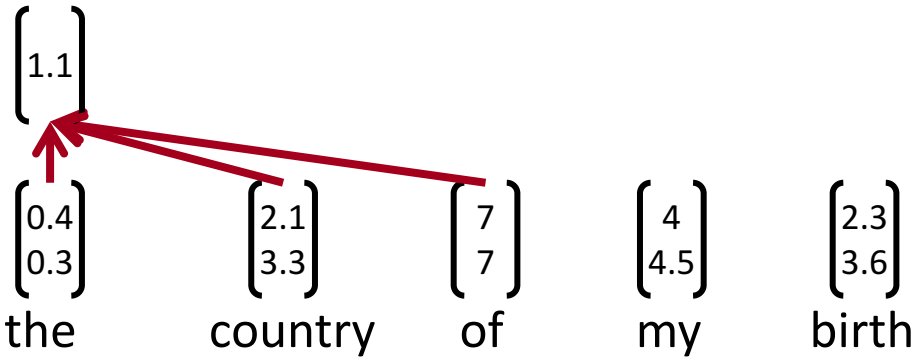
- 2d example →
- Yellow and red numbers show filter weights
- Green shows input



Stanford UFLDL wiki

Single Layer CNN

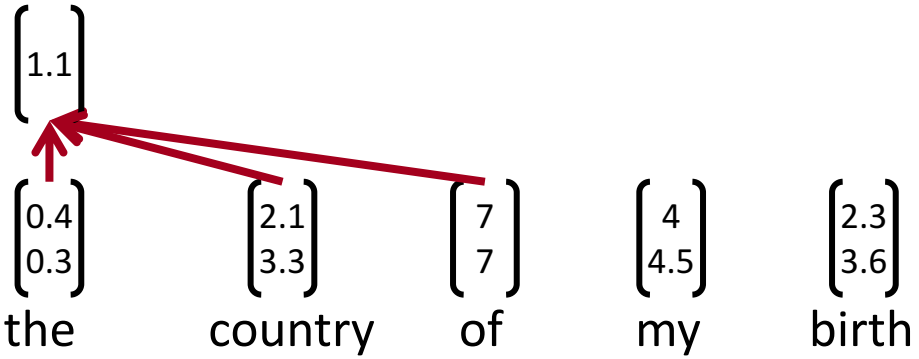
- A simple variant using one convolutional layer and **pooling**
- Based on Collobert and Weston (2011) and Kim (2014) "Convolutional Neural Networks for Sentence Classification"
- Word vectors: $\mathbf{x}_i \in \mathbb{R}^k$
- Sentence: $\mathbf{x}_{1:n} = \mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \ldots \oplus \mathbf{x}_n$ (vectors concatenated)
- Concatenation of words in range: $\mathbf{x}_{i:i+j}$
- Convolutional filter: $\mathbf{w} \in \mathbb{R}^{hk}$ (goes over window of h words)
- Could be 2 (as before) higher, e.g. 3:



Single layer CNN

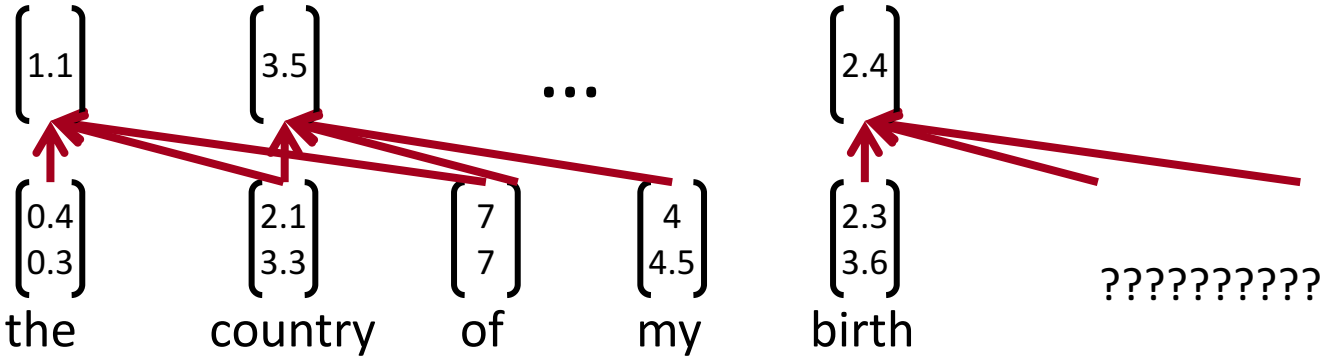
- Convolutional filter: $\mathbf{w} \in \mathbb{R}^{hk}$ (goes over window of h words)
- Note, filter is vector!
- Window size h could be 2 (as before) or higher, e.g. 3:
- To compute feature for CNN layer:

$$c_i = f(\mathbf{w}^T \mathbf{x}_{i:i+h-1} + b)$$



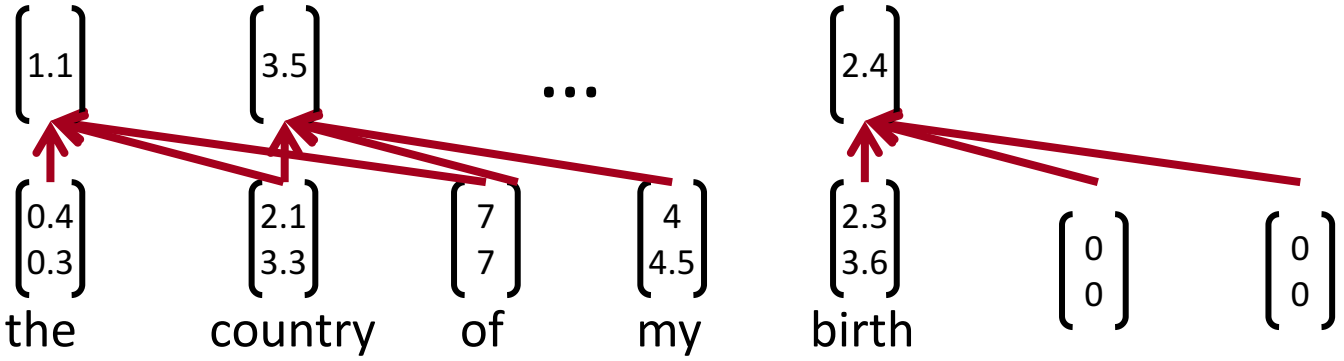
Single layer CNN

- Filter w is applied to all possible windows (concatenated vectors)
- Sentence: $\mathbf{x}_{1:n} = \mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \ldots \oplus \mathbf{x}_n$
- All possible windows of length h: $\{\mathbf{x}_{1:h}, \mathbf{x}_{2:h+1}, \ldots, \mathbf{x}_{n-h+1:n}\}$
- Result is a feature map: $\mathbf{c} = [c_1, c_2, \ldots, c_{n-h+1}] \in \mathbb{R}^{n-h+1}$



Single layer CNN

- Filter w is applied to all possible windows (concatenated vectors)
- Sentence: $\mathbf{x}_{1:n} = \mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \ldots \oplus \mathbf{x}_n$
- All possible windows of length h: $\{\mathbf{x}_{1:h}, \mathbf{x}_{2:h+1}, \ldots, \mathbf{x}_{n-h+1:n}\}$
- Result is a feature map: $\mathbf{c} = [c_1, c_2, \ldots, c_{n-h+1}] \in \mathbb{R}^{n-h+1}$



Single layer CNN: Pooling layer

- New building block: Pooling
- In particular: max-over-time pooling layer
- Idea: capture most important activation (maximum over time)

- From feature map $\mathbf{c} = [c_1, c_2, \ldots, c_{n-h+1}] \in \mathbb{R}^{n-h+1}$

- Pooled single number: $\hat{c} = \max\{\mathbf{c}\}$

- But we want more features!

Solution: Multiple filters

- Use multiple filter weights w
- Useful to have different window sizes h
- Because of max pooling $\hat{c} = \max\{\mathbf{c}\}$, length of **c** irrelevant

$$\mathbf{c} = [c_1, c_2, \ldots, c_{n-h+1}] \in \mathbb{R}^{n-h+1}$$

- So we can have some filters that look at unigrams, bigrams, tri-grams, 4-grams, etc.

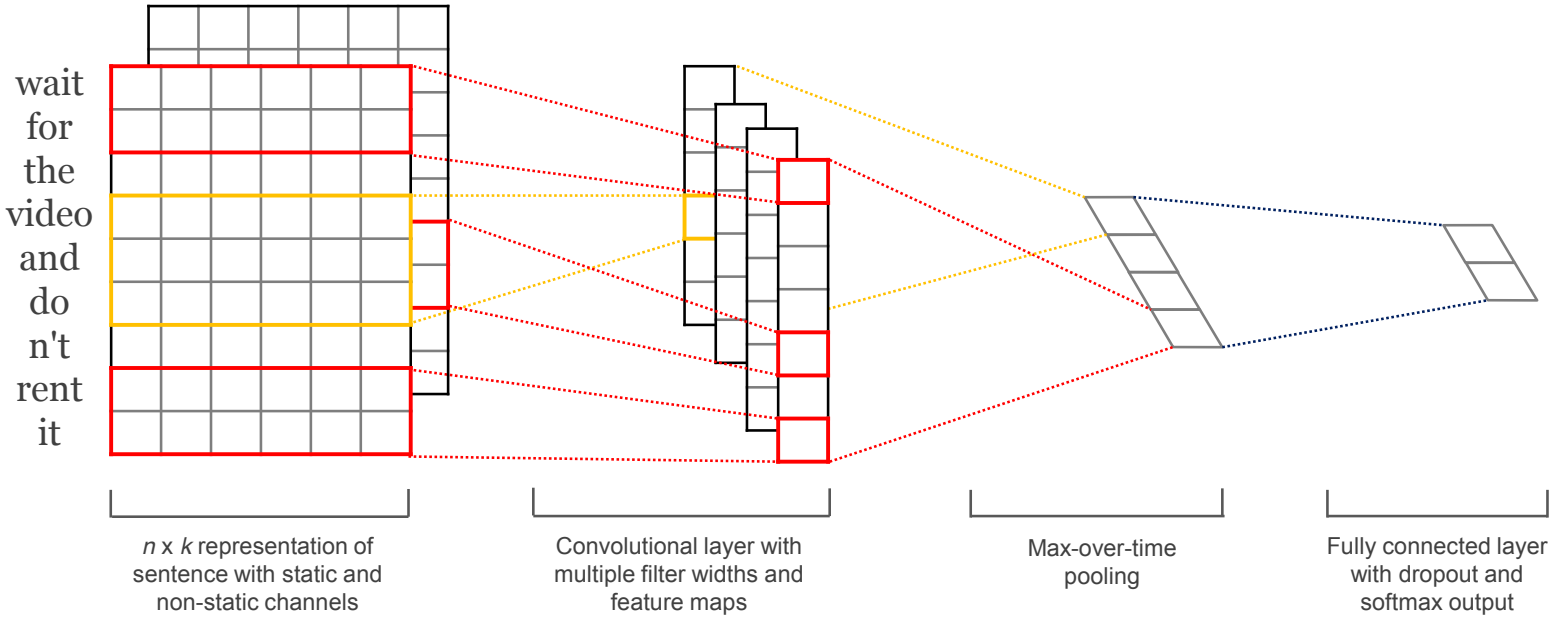
Multi-channel idea

- Initialize with pre-trained word vectors (word2vec or Glove)
- Start with two copies
- Backprop into only one set, keep other “static”
- Both channels are added to c_i before max-pooling

Classification after one CNN layer

- First one convolution, followed by one max-pooling
- To obtain final feature vector: $\mathbf{z} = [\hat{c}_1, \ldots, \hat{c}_m]$ (assuming m filters w)
- Simple final softmax layer $y = softmax\left(W^{(S)} z + b\right)$

Figure from Kim (2014)



n words (possibly zero padded) and each word vector has k dimensions

Tricks to make it work better: Dropout

- Idea: randomly mask/dropout/set to 0 some of the feature weights z
- Create masking vector r of Bernoulli random variables with probability p (a hyperparameter) of being 1

- Delete features during training:

$$y = softmax\left(W^{(S)}(r \circ z) + b\right)$$

- Reasoning: Prevents co-adaptation (overfitting to seeing specific feature constellations)

- Paper: Hinton et al. 2012: Improving neural networks by preventing co-adaptation of feature detectors

Tricks to make it work better: Dropout

$$y = softmax\left(W^{(S)}(r \circ z) + b\right)$$

- At training time, gradients are backpropagated only through those elements of z vector for which $r_i = 1$

- At test time, there is no dropout, so feature vectors z are larger.
- Hence, we scale final vector by Bernoulli probability p

$$\hat{W}^{(S)} = pW^{(S)}$$

- Kim (2014) reports **2 – 4% improved accuracy** and ability to use very large networks without overfitting

Another regularization trick

- Constrain l_2 norms of weight vectors of each class (row in softmax weight $W^{(S)}$) to fixed number s (also a hyperparameter)

- If $\|W_{c\cdot}^{(S)}\| > s$, then rescale it so that: $\|W_{c\cdot}^{(S)}\| = s$

- Not very common

All hyperparameters in Kim (2014)

- Find hyperparameters based on dev set
- Nonlinearity: reLu
- Window filter sizes h = 3,4,5
- Each filter size has 100 feature maps
- Dropout p = 0.5
- L2 constraint s for rows of softmax s = 3
- Mini batch size for SGD training: 50
- Word vectors: pre-trained with word2vec, k = 300

- During training, keep checking performance on dev set and pick highest accuracy weights for final evaluation

Experiments

Model	MR	SST-1	SST-2	Subj	TREC	CR	MPQA
CNN-rand	76.1	45.0	82.7	89.6	91.2	79.8	83.4
CNN-static	81.0	45.5	86.8	93.0	92.8	84.7	**89.6**
CNN-non-static	**81.5**	48.0	87.2	93.4	93.6	84.3	89.5
CNN-multichannel	81.1	47.4	**88.1**	93.2	92.2	**85.0**	89.4
RAE (Socher et al., 2011)	77.7	43.2	82.4	–	–	–	86.4
MV-RNN (Socher et al., 2012)	79.0	44.4	82.9	–	–	–	–
RNTN (Socher et al., 2013)	–	45.7	85.4	–	–	–	–
DCNN (Kalchbrenner et al., 2014)	–	48.5	86.8	–	93.0	–	–
Paragraph-Vec (Le and Mikolov, 2014)	–	**48.7**	87.8	–	–	–	–
CCAE (Hermann and Blunsom, 2013)	77.8	–	–	–	–	–	87.2
Sent-Parser (Dong et al., 2014)	79.5	–	–	–	–	–	86.3
NBSVM (Wang and Manning, 2012)	79.4	–	–	93.2	–	81.8	86.3
MNB (Wang and Manning, 2012)	79.0	–	–	**93.6**	–	80.0	86.3
G-Dropout (Wang and Manning, 2013)	79.0	–	–	93.4	–	82.1	86.1
F-Dropout (Wang and Manning, 2013)	79.1	–	–	**93.6**	–	81.9	86.3
Tree-CRF (Nakagawa et al., 2010)	77.3	–	–	–	–	81.4	86.1
CRF-PR (Yang and Cardie, 2014)	–	–	–	–	–	82.7	–
SVM_S (Silva et al., 2011)	–	–	–	–	**95.0**	–	–

Problem with comparison?

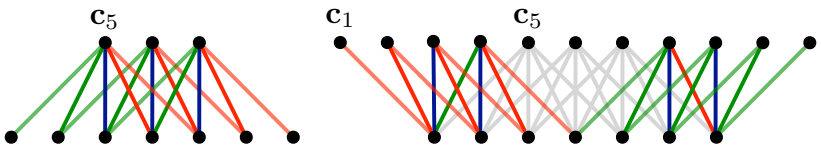
- Dropout gives 2 – 4 % accuracy improvement
- Several ‘’baselines’’ didn’t use dropout

- Still remarkable results and simple architecture!

- Difference to window and RNN architectures we described in previous lectures: pooling, many filters and dropout
- Some of these ideas can be used in RNNs too and have since been improved → more in later lectures

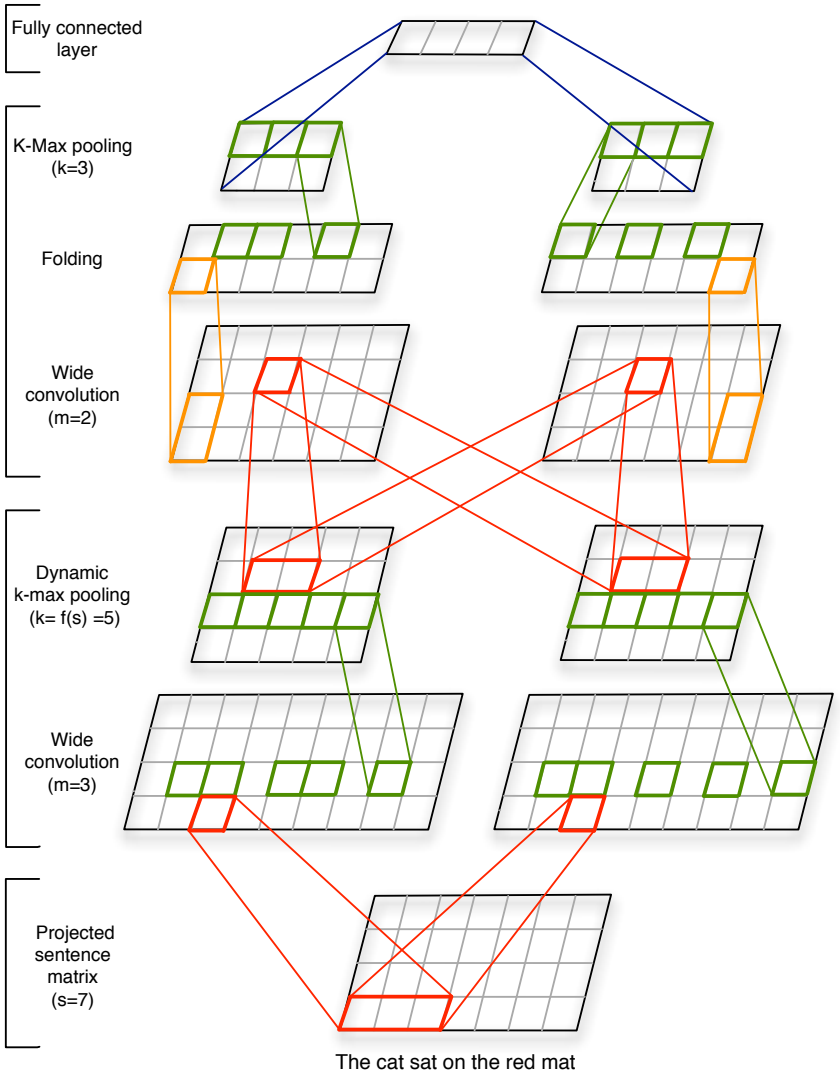
CNN alternatives

- Narrow vs wide convolution



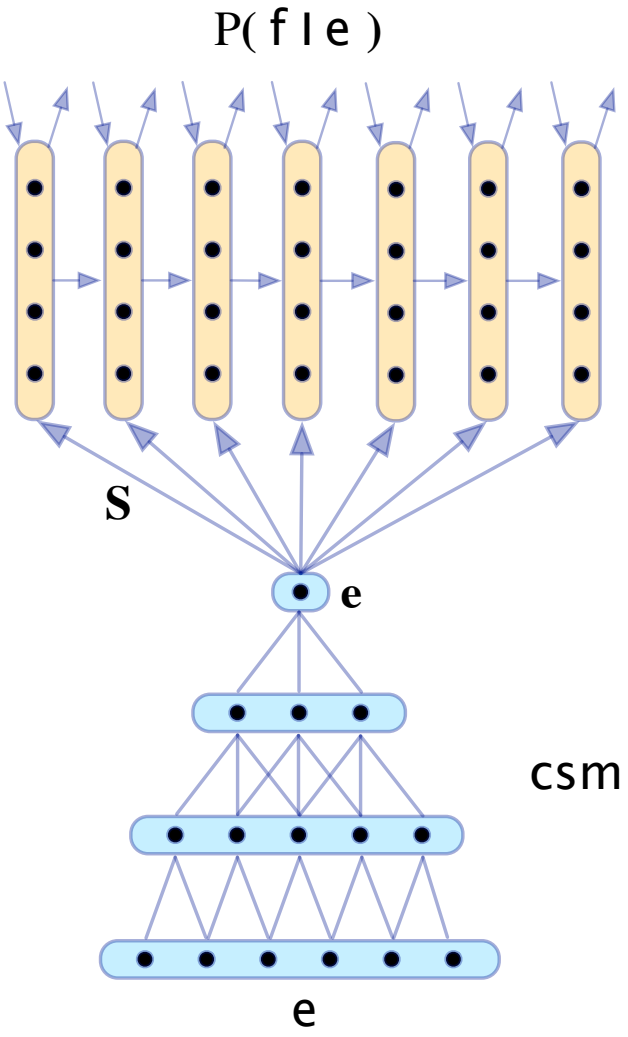
- Complex pooling schemes (over sequences) and deeper convolutional layers

- Kalchbrenner et al. (2014)



CNN application: Translation

- One of the first successful neural machine translation efforts
- Uses CNN for encoding and RNN for decoding
- Kalchbrenner and Blunsom (2013)
  "Recurrent Continuous Translation Models"
- Many more recent models we may cover in future lectures



Next Lectures:

- Coreference resolution with Kevin
- Recursive Neural Networks
- Then model comparisons and tuning tricks