CS546: Machine Learning in NLP (Spring 2020)
http://courses.engr.illinois.edu/cs546/

Lecture 9: Transformers, ELMO

Julia Hockenmaier
juliahmr@illinois.edu
3324 Siebel Center

Office hours: Monday, 11am—12:30pm

Project proposals

Prepare a one minute presentation: 1 to 2 pages.
— what are you planning to do?
— why is this interesting?
— what's your data, evaluation metric?
— what software can you build on?

Email me a PPT *and* PDF version of your slides by 10am on Jan 28.

Be in class to give your presentation!

Paper presentations

First set this Friday

You will receive an email from me with your group's paper assignments
— everybody needs to choose one paper (or one section of a longer paper)
— first come, first serve
— please arrange among your group to bring in a computer to present on (you should use a single slide deck/computer, if possible)
— email me slides

Today's class

Context-Dependent Embeddings: ELMO
Transformers

ELMo

Deep contextualized word representations
Peters et al., NAACL 2018

see also https://allenai.github.io/allennlp-docs/tutorials/how_to/elmo/

Embeddings from Language Models

Replace static embeddings (lexicon lookup) with context-dependent embeddings (produced by a deep neural language model)

=> Each token's representation is a function of the entire input sentence, computed by a deep (multi-layer) bidirectional language model

=> Return for each token a (task-dependent) linear combination of its representation across layers.

=> Different layers capture different information

ELMo architecture

— Train a multi-layer bidirectional language model with character convolutions on raw text

— Each layer of this language model network computes a vector representation for each token.

— Freeze the parameters of the language model.

— For each task: train task-dependent softmax weights to combine the layer-wise representations into a single vector for each token *jointly* with a task-specific model that uses those vectors

ELMo's Bidirectional language models

The forward LM is a deep LSTM that goes over the sequence from start to end to predict token t_k based on the prefix $t_1 \ldots t_{k-1}$:

$$p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s)$$

Parameters: token embeddings Θ_x LSTM $\overrightarrow{\Theta}_{LSTM}$ softmax Θ_s

The backward LM is a deep LSTM that goes over the sequence from end to start to predict token t_k based on the suffix $t_{k+1} \ldots t_N$:

$$p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s)$$

Train these LMs jointly, with the same parameters for the token representations and the softmax layer (but not for the LSTMs)

$$\sum_{k=1}^{N} \left(\log p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \right)$$

ELMo’s token representations

The input token representations are purely character-based: a character CNN, followed by linear projection to reduce dimensionality

“2048 character n-gram convolutional filters with two highway layers, followed by a linear projection to 512 dimensions”

Advantage over using fixed embeddings:
no UNK tokens, any word can be represented

ELMo's token representations

Given a token representation $\mathbf{x}_k$, each layer j of the LSTM language models computes a vector representation $\mathbf{h}_{k,j}$ for every token k.

With L layers, ELMo represents each token as

$$
\begin{aligned}
R_k &= \{\mathbf{x}_k^{LM}, \overrightarrow{\mathbf{h}}_{k,j}^{LM}, \overleftarrow{\mathbf{h}}_{k,j}^{LM} \mid j = 1, \ldots, L\} \\
&= \{\mathbf{h}_{k,j}^{LM} \mid j = 0, \ldots, L\},
\end{aligned}
$$

where $\mathbf{h}_{k,j}^{LM} = [\overrightarrow{\mathbf{h}}_{k,j}^{LM}; \overleftarrow{\mathbf{h}}_{k,j}^{LM}]$ and $\mathbf{h}_{k,0}^{LM} = \mathbf{x}_k$

ELMo learns softmax weights s_j^{task} to collapse these vectors into a single vector and a task-specific scalar γ^{task}:

$$
\mathbf{ELMo}_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \mathbf{h}_{k,j}^{LM}.
$$

How do you use ELMo?

ELMo embeddings can be used as (additional) input to any neural model

— ELMo can be tuned with dropout and L2-regularization (so that all layer weights stay close to each other)

— It often helps to fine-tune the biLMs (train them further) on task-specific raw text

In general: concatenate $\mathbf{ELMo}_k^{task}$ with other embeddings $\mathbf{x}_k$ for token input

If the output layer of the task network operates over token representations, ELMO embeddings can also (additionally) be added there.

Results

ELMo gave improvements on a variety of tasks:
— question answering (SQuAD)
— entailment/natural language inference (SNLI)
— semantic role labeling (SRL)
— coreference resolution (Coref)
— named entity recognition (NER)
— sentiment analysis (SST-5)

TASK	PREVIOUS SOTA		OUR BASELINE	ELMO + BASELINE	INCREASE (ABSOLUTE/ RELATIVE)
SQuAD	Liu et al. (2017)	84.4	81.1	85.8	4.7 / 24.9%
SNLI	Chen et al. (2017)	88.6	88.0	88.7 ± 0.17	0.7 / 5.8%
SRL	He et al. (2017)	81.7	81.4	84.6	3.2 / 17.2%
Coref	Lee et al. (2017)	67.2	67.2	70.4	3.2 / 9.8%
NER	Peters et al. (2017)	91.93 ± 0.19	90.15	92.22 ± 0.10	2.06 / 21%
SST-5	McCann et al. (2017)	53.7	51.4	54.7 ± 0.5	3.3 / 6.8%

Using ELMo at input vs output

Task	Input Only	Input & Output	Output Only
SQuAD	85.1	**85.6**	84.8
SNLI	88.9	**89.5**	88.7
SRL	**84.7**	84.3	80.9

Table 3: Development set performance for SQuAD, SNLI and SRL when including ELMo at different locations in the supervised model.

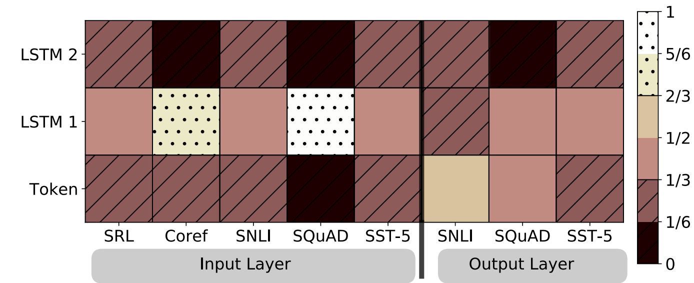


Figure 2: Visualization of softmax normalized biLM layer weights across tasks and ELMo locations. Normalized weights less then $1/3$ are hatched with horizontal lines and those greater then $2/3$ are speckled.

The supervised models for question-answering, entailment and SRL all use sequence architectures.
— We can concatenate ELMo to the input and/or the output of that network (with different layer weights)
—> Input always helps, Input+output often helps
—> Layer weights differ for each task

Transformers

Vashwani et al. *Attention is all you need,* NIPS 2017

Transformers

Sequence transduction model based on attention (no convolutions or recurrence)
— easier to parallelize than recurrent nets
— faster to train than recurrent nets
— captures more long-range dependencies than CNNs with fewer parameters

Transformers use stacked self-attention and pointwise, fully-connected layers for the encoder and decoder

Transformer Architecture

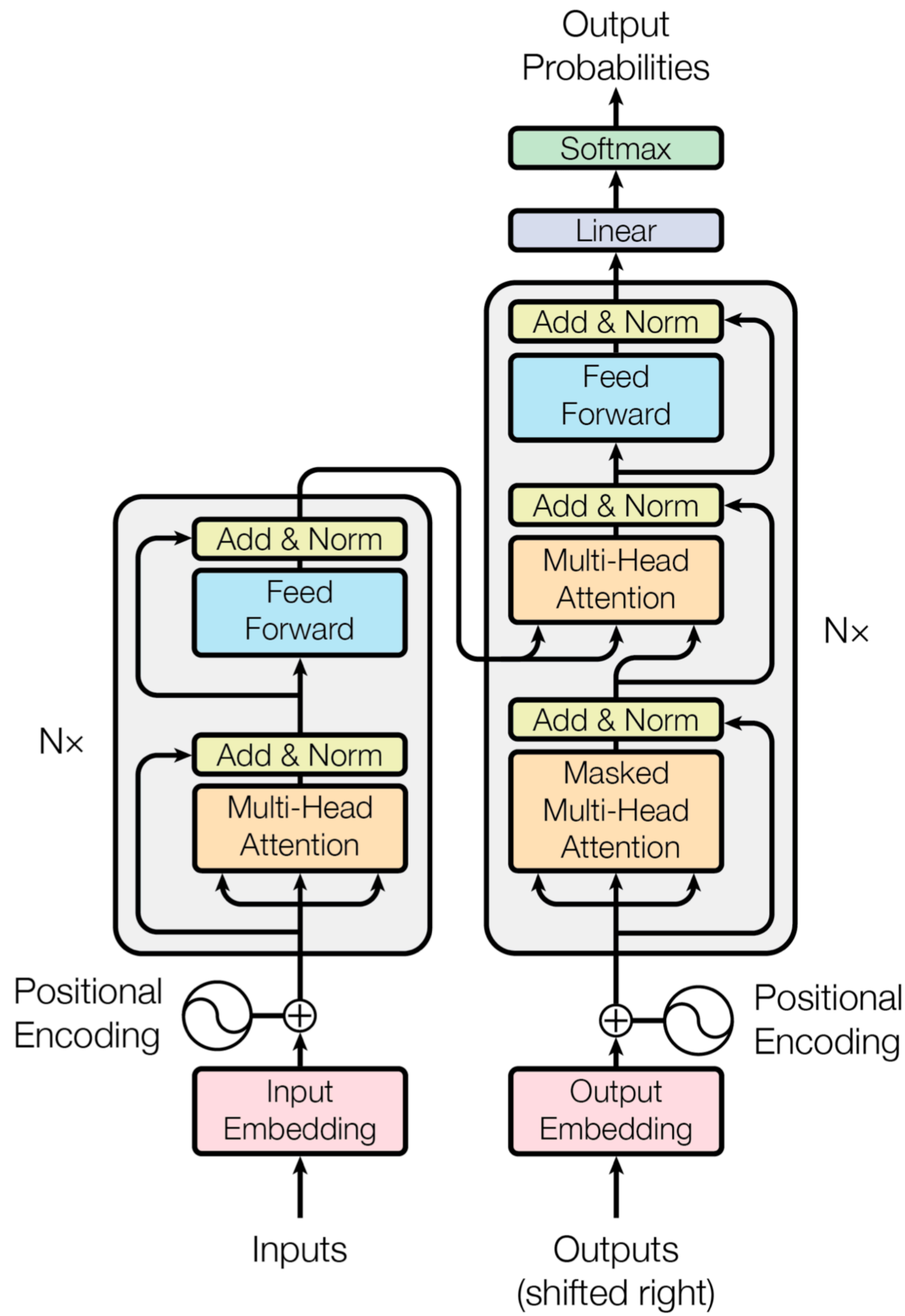


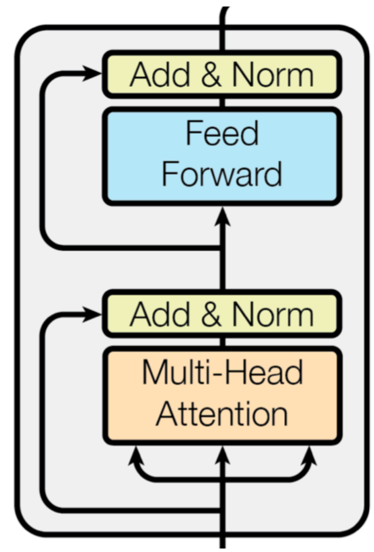
Figure 1: The Transformer - model architecture.

Encoder

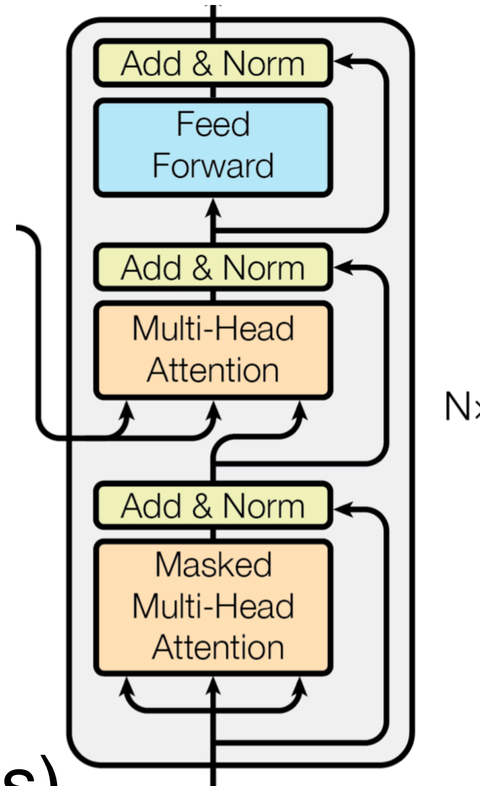
A stack of N=6 identical layers
All layers and sublayers are 512-dimensional

Each layer consists of two sublayers
— one multi-headed self attention layer
— one position-wise fully connected layer

Each sublayer has a residual connection
and is normalized:
LayerNorm(x + Sublayer(x))



Decoder



A stack of N=6 identical layers
All layers and sublayers are 512-d

Each layer consists of **three** sublayers
— one multi-headed self attention layer
over decoder output (ignoring future tokens)
— one multi-headed attention layer
over encoder output
— one position-wise fully connected layer

Each sublayer has a residual connection
and is normalized:
LayerNorm(x + Sublayer(x))

Self-attention w/ queries, keys, values

Let's add learnable parameters ($k \times k$ weight matrices),
and turn each vector $\mathbf{x}^{(i)}$ into three versions:

- **Query** vector $\mathbf{q}^{(i)} = \mathbf{W}_q \mathbf{x}^{(i)}$
- **Key** vector: $\mathbf{k}^{(i)} = \mathbf{W}_k \mathbf{x}^{(i)}$
- **Value** vector: $\mathbf{v}^{(i)} = \mathbf{W}_v \mathbf{x}^{(i)}$

The **attention weight of the *j*-th position** to compute the **new output for the *i*-th position** depends on the **query of i** and the **key of j (scaled):**

$$w_j^{(i)} = \frac{\exp(\mathbf{q}^{(i)}\mathbf{k}^{(j)})/\sqrt{k}}{\sum_j (\exp(\mathbf{q}^{(i)}\mathbf{k}^{(j)})/\sqrt{k})}$$

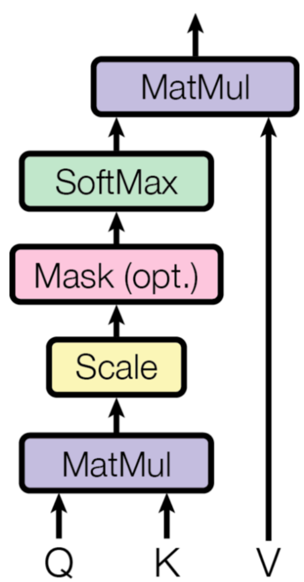
The **new output vector for the i-th position** depends on the **attention weights** and **value** vectors of all **input positions j**:

$$\mathbf{y}^{(i)} = \sum_{j=1..T} w_j^{(i)} \mathbf{v}^{(j)}$$

Scaled Dot-Product Attention

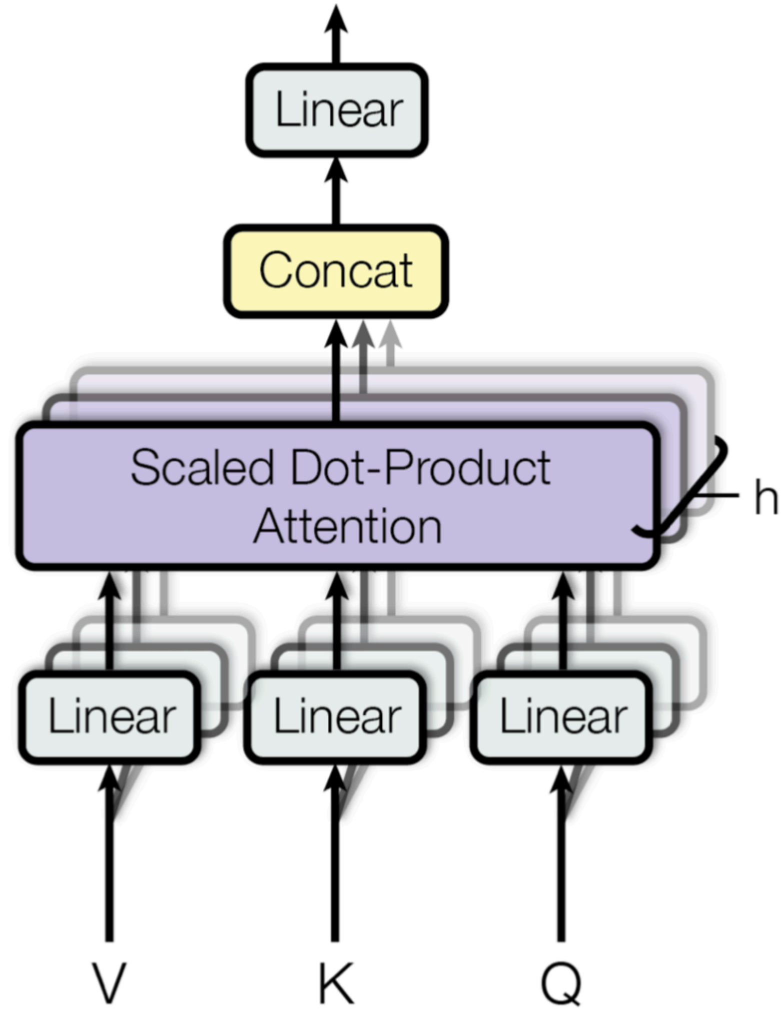
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Scaled Dot-Product Attention



Multi-Head attention

- Learn *h* different linear projections of Q,K,V
- Compute attention separately on each of these *h* versions
- Concatenate and project the resultant vectors to a lower dimensionality.
- Each attention head can use low dimensionality



$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Position-wise feedforward nets

We train a feedforward net for each layer that only reads in input for its token
(two linear transformations with ReLU in between)

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Input and output: 512 dimensions
Internal layer: 2048 dimensions

Parameters differ from layer to layer
(but are shared across positions)
(cf. 1x1 convolutions)

Positional Encoding

How does this model capture sequence order?

Positional embeddings have the same dimensionality as word embeddings (512) and are added in.

Fixed representations: each dimension is a sinuoid (a sine or cosine function with a different frequency)

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