Lecture 27: Intro to Large Language Models

Julia Hockenmaier
juliahmr@illinois.edu
Today’s class

Recap: Using RNNs for various NLP tasks

From static to contextual embeddings: ELMO

Recap: Transformers

Subword tokenizations

Early Large Language Models (GPT, BERT)
Recap:
Using RNNs for different NLP tasks
RNNs for language generation

AKA “autoregressive generation”

In a hole?

Sampled Word

Softmax

Embedding

Input Word

Figure 9.7 Autoregressive generation with an RNN-based neural language model.

Task is part-of-speech tagging, discussed in detail in Chapter 8. In an RNN approach to POS tagging, inputs are word embeddings and the outputs are tag probabilities generated by a softmax layer over the tagset, as illustrated in Fig. 9.8.

In this figure, the inputs at each time step are pre-trained word embeddings corresponding to the input tokens. The RNN block is an abstraction that represents an unrolled simple recurrent network consisting of an input layer, hidden layer, and output layer at each time step, as well as the shared \( U \), \( V \) and \( W \) weight matrices that comprise the network. The outputs of the network at each time step represent the distribution over the POS tagset generated by a softmax layer.

To generate a tag sequence for a given input, we can run forward inference over the input sequence and select the most likely tag from the softmax at each step. Since we’re using a softmax layer to generate the probability distribution over the output ...

Figure 9.8 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.
An RNN for Machine Translation

Now, consider an ingenious extension of this idea from the world of machine translation (MT), the task of automatically translating sentences from one language into another. The primary resources used to train modern translation systems are known as parallel texts, or bitexts. These are large text collections consisting of pairs of sentences from different languages that are translations of one another. Traditionally in MT, the text being translated is referred to as the source and the translation output is called the target.

To extend language models and autoregressive generation to machine translation, we'll first add an end-of-sentence marker at the end of each bitext's source sentence and then simply concatenate the corresponding target to it. These concatenated source-target pairs can now serve as training data for a combined language model. Training proceeds as with any RNN-based language model. The network is trained autoregressively to predict the next word in a set of sequences comprised of the concatenated source-target bitexts, as shown in Fig. 10.2.

To translate a source text using the trained model, we run it through the network performing forward inference to generate hidden states until we get to the end of the source. Then we begin autoregressive generation, asking for a word in the context of the hidden layer from the end of the source input as well as the end-of-sentence marker. Subsequent words are conditioned on the previous hidden state and the embedding for the last word generated.

Early efforts using this clever approach demonstrated surprisingly good results on standard datasets and led to a series of innovations that were the basis for networks discussed in the remainder of this chapter. Chapter 11 provides an in-depth discussion of the fundamental issues in translation as well as the current state-of-the-art approaches to MT. Here, we'll focus on the powerful models that arose from these early efforts.
RNNs for sequence classification

If we just want to assign **one label** to the entire sequence, we don’t need to produce output at each time step, so we can use a simpler architecture.

We can use the hidden state of the last word in the sequence as input to a feedforward net:
Basic RNNs for sequence labeling

Sequence labeling (e.g. POS tagging):
Assign one label to each element in the sequence.

RNN Architecture:
Each time step has a distribution over output classes

Extension: add a CRF layer to capture dependencies among labels of adjacent tokens.
ELMo
Embeddings from Language Models

Replace static embeddings (lexicon lookup) with context-dependent embeddings (produced by a 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

Peters et al., NAACL 2018
ELMo

Pre-training:
— 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 language model parameters.

Fine-tuning (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 input 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 bidirectional language models

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

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

Parameters: token embeddings $\Theta_x$, LSTM $\Theta_{LSTM}$, softmax $\Theta_s$

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

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

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

$$\sum_{k=1}^{N} \left( \log p(t_k \mid t_1, \ldots, t_{k-1}; \Theta_x, \Theta_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, \ldots, t_N; \Theta_x, \Theta_{LSTM}, \Theta_s) \right)$$
ELMo’s output token representations

Given an input token representation $x_k$, each layer $j$ of the LSTM language models computes a vector representation $h_{k,j}$ for every token $k$.

With $L$ layers, ELMo represents each token as $L$ vectors $h_{k,l}^{LM}$

$$R_k = \{ x_k^{LM}, h_{k,j}^{LM,LM}, \overrightarrow{h}_{k,j}^{LM} | j = 1, \ldots, L \}$$

$$= \{ h_{k,j}^{LM} | j = 0, \ldots, L \},$$

where $h_{k,j}^{LM} = [h_{k,j}^{LM}; \overrightarrow{h}_{k,j}^{LM}]$ and $h_{k,0}^{LM} = x_k$

ELMo learns softmax weights $s_j^{task}$ and a task-specific scalar $\gamma^{task}$ to collapse these $L$ vectors into a single task-specific token vector:

$$ELMo_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} h_{k,j}^{LM}.$$
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)

<table>
<thead>
<tr>
<th>TASK</th>
<th>PREVIOUS SOTA</th>
<th>OUR BASELINE</th>
<th>ELMo + BASELINE</th>
<th>INCREASE (ABSOLUTE/RELATIVE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQuAD</td>
<td>Liu et al. (2017)</td>
<td>84.4</td>
<td>81.1</td>
<td>4.7 / 24.9%</td>
</tr>
<tr>
<td>SNLI</td>
<td>Chen et al. (2017)</td>
<td>88.6</td>
<td>88.0</td>
<td>0.7 / 5.8%</td>
</tr>
<tr>
<td>SRL</td>
<td>He et al. (2017)</td>
<td>81.7</td>
<td>81.4</td>
<td>3.2 / 17.2%</td>
</tr>
<tr>
<td>Coref</td>
<td>Lee et al. (2017)</td>
<td>67.2</td>
<td>67.2</td>
<td>3.2 / 9.8%</td>
</tr>
<tr>
<td>NER</td>
<td>Peters et al. (2017)</td>
<td>91.93 ± 0.19</td>
<td>90.15</td>
<td>2.06 / 21%</td>
</tr>
<tr>
<td>SST-5</td>
<td>McCann et al. (2017)</td>
<td>53.7</td>
<td>51.4</td>
<td>3.3 / 6.8%</td>
</tr>
</tbody>
</table>
ELMo:

ELMo showed that **contextual embeddings** are very useful: it outperformed other models on many tasks.

- ELMo embeddings could also be concatenated with other token-specific features, depending on the task.

ELMo requires training a task-specific softmax and scalar to predict how best to combine each layer.

- Not all layers were equally useful for each task.
Recap: Seq2seq, Transformers
Encoder-Decoder (seq2seq) model

The **decoder** is a language model that generates an output sequence **conditioned on the input** sequence.

- **Vanilla RNN**: condition on the **last** hidden state
- **Attention**: condition on **all** hidden states
Transformers use Self-Attention

**Attention so far** (in seq2seq architectures):
In the **decoder** (which has access to the complete input sequence), compute attention weights over **encoder** positions that depend on each **decoder** position

**Self-attention:**
If the **encoder** has access to the complete input sequence, we can also compute attention weights over **encoder** positions that depend on each **encoder** position

    **self-attention:**

For each **decoder** position \( t \)...

    ...Compute an attention weight for each **encoder** position \( s \)
    ...Renormalize these weights (that depend on \( t \)) w/ softmax to get a new weighted avg. of the input sequence vectors
Transformer Architecture

Non-Recurrent Encoder-Decoder architecture

— No hidden states
— Context information captured via attention and positional encodings
— Consists of stacks of layers with various sublayers

Vaswani et al, NIPS 2017
Encoder Vaswani et al, NIPS 2017

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

Each layer consists of **two sublayers**
— one **multi-head self attention** layer
— one **position-wise feed forward** layer

Each sublayer is followed by an **“Add & Norm”** layer:
... a **residual connection** $x + \text{Sublayer}(x)$
  (the input $x$ is added to the output of the sublayer)
... followed by a **normalization step**
  (using the mean and standard deviation of its activations)
  $\text{LayerNorm}(x + \text{Sublayer}(x))$
Decoder  Vaswani et al, NIPS 2017

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

Each layer consists of three sublayers
— one **masked multi-head self attention layer**
  over **decoder** output
  (masked, i.e. ignoring future tokens)
— one **multi-headed attention layer**
  over **encoder** output
— one **position-wise feed forward layer**

Each sublayer has a residual connection
and is normalized: $\text{LayerNorm}(x + \text{Sublayer}(x))$
Subword Tokenization
BPE Tokenization (Sennrich et al, ACL 2016)

BytePair Encoding (Gage 1994): a compression algorithm that iteratively replaces the most common pair of adjacent bytes with a single, unused byte

BPE tokenization: introduce new tokens by merging the most common adjacent pairs of tokens

Start with all characters, plus a special end-of-word character
Introduce new token by merging the most common pair of adjacent tokens.
(Assumption: each individual token will still occur in a different context, so we will also keep both tokens in the vocabulary)

Machine translation: train one tokenizer across both languages (better generalization for related languages)
Wordpiece tokenization  (Wu et al, 2016)


Segment words into subtokens (with special word boundary symbols to recover original tokenization)

**Input:** Jet makers feud over seat width with big orders at stake
**Output:** _Jet_makers_feud_over_seat_width_with_big_orders_at_stake

**Training of Wordpiece:**
- Specify desired number of tokens, D
- Add word boundary token (at beginning of words)
- Optimization task: greedily merge adjacent characters to improve log-likelihood of data until the vocabulary has size D.
Subword Regularization (Kudo, ACL 2018)

Observation: Subword tokenization can be ambiguous. Can this be harnessed?

Approach: Train a (translation) model with (multiple) subword segmentations that are sampled from a character-based unigram language model.

Training the unigram model:
Start with an overly large seed vocabulary $V$ (all possible single-character tokens and many multi-character tokens)
Randomly sample a segmentation from the unigram model
Decide which multi-character words to remove from $V$ based on how the likelihood decreases by removing them
Stop when the vocabulary is small enough.
Generative Pre-Training (Radford et al, 2018)

Auto-regressive 12-layer transformer decoder
- Each token only conditioned on preceding context
- BPE tokenization (|V| = 40K), 768 hidden size, 12 attention heads

Pre-trained on raw text as a language model
- (Maximize the probability of predicting the next word)

Fine-tuned on labeled data (and language modeling)
- Include new start, delimiter and end tokens,
- plus linear layer added to last layer of end token output.
**BERT** (Devlin et al, NAACL 2019)

**Fully bidirectional transformer encoder**
- $\text{BERT}_{\text{base}}$: 12 layers, hidden size=768, 12 att’n heads (110M parameters)
- $\text{BERT}_{\text{large}}$: 24 layers, hidden size=1024, 16 attention heads (340M parameters)

**Input:** sum of token, positional, segment embeddings
- **Segment embeddings** (A and B): is this token part of sentence A (before SEP) or sentence B (after SEP)?

[CLS] and [SEP] tokens: added during pre-training

**Pre-training tasks:**
- Masked language modeling
- Next sentence prediction
## BERT Input

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[CLS]</td>
<td>my</td>
<td>dog</td>
<td>is</td>
<td>cute</td>
<td>[SEP]</td>
</tr>
<tr>
<td></td>
<td>he</td>
<td>likes</td>
<td>play</td>
<td># #ing</td>
<td>[SEP]</td>
</tr>
</tbody>
</table>

### Token Embeddings

- $E_{[CLS]}$
- $E_{my}$
- $E_{dog}$
- $E_{is}$
- $E_{cute}$
- $E_{[SEP]}$
- $E_{he}$
- $E_{likes}$
- $E_{play}$
- $E_{# #ing}$
- $E_{[SEP]}$

### Segment Embeddings

- $E_A$
- $E_A$
- $E_A$
- $E_A$
- $E_A$
- $E_B$
- $E_B$
- $E_B$
- $E_B$
- $E_B$

### Position Embeddings

- $E_0$
- $E_1$
- $E_2$
- $E_3$
- $E_4$
- $E_5$
- $E_6$
- $E_7$
- $E_8$
- $E_9$
- $E_{10}$
Pre-training tasks

BERT is jointly pre-trained on two tasks:

**Next-sentence prediction:** [based on CLS token]
Does sentence B follow sentence A in a real document?

**Masked language modeling:**
15% of tokens are randomly chosen as **masking tokens**
10% of the time, a masking token remains unchanged
10% of the time, a masking token is replaced by a random token
80% of the time, a masking token is replaced by `[MASK]`, and the output layer has to predict the original token
Using BERT for classification

Sentence Pair Classification

Single Sentence Classification

Add a **softmax classifier** on final layer of **[CLS]** token
Using BERT for Question-Answering

**Input:** [CLS] question [SEP] answer passage [SEP]

Learn to predict a **START** and an **END token** on answer tokens

Represent START and END as H-dimensional vectors \( S, E \)

Find the most likely start and end tokens in the answer by computing a softmax over the dot product of all token embeddings \( T_i \) and \( S \) (or \( E \))

\[
P(T_i \text{ is start}) = \frac{\exp(T_i \cdot S)}{\sum_j \exp(T_j \cdot S)}
\]
Using BERT for Sequence Labeling

Add a **softmax classifier** to the tokens in the sequence
Fine-tuning BERT

To use BERT on any task, it needs to be fine-tuned:

— Add any new parts to the model  
  (e.g. classifier layers)  
  This will add **new parameters** (initialized randomly)

— Retrain the entire model (update all parameters)
More compact BERT models  
(Turc et al., 2019)

Pre-training and fine-tuning works well on much smaller BERT variants  
https://arxiv.org/abs/1908.08962

Additional improvements through knowledge distillation:

- **Pre-train** a compact model (‘student’) in the standard way
- Train/Fine-tune a large model (‘teacher’) on the target task
- **Knowledge distillation** step:
  - Train the student on noisy task predictions made by teacher
  - Fine-tune student on actual task data

Students can have more layers (but smaller embeddings) than models trained in the standard way
BERT Variants
RoBERTA  (Liu et al. 2019)

Investigates **better pre-training** for BERT
Found that BERT was undertrained.
Optimizes hyperparameter choice.
Evaluates next-sentence prediction task
RoBERTA outperforms BERT on several tasks.

**Pre-training improvements:**
- **Dynamic masking:** randomly change which tokens in a sentence get masked (BERT: same tokens in each epoch)
- **Much larger batch sizes** (2K sentences instead of 256)
- Use byte-level BPE, not character level BPE
BART  (Lewis et al., ACL 2020)

Combines **bidirectional encoder** (like BERT) with **auto-regressive** (unidirectional) **decoder** (like GPT)

Used for **classification, generation, translation**

Uses final token of decoder sequence for classification tasks.

Pre-training: corrupts (encoder) input with **masking, deletion, rotation, permutation, infilling.**

Decoder needs to recover original input.
SentenceBERT (Reimers & Gurevych, EMNLP 2019)

For tasks that require scoring of **sentence pairs**
(e.g. semantic textual similarity, or entailment recognition)

Motivation: BERT treats sequence pairs as one (long) sequence, but cross-attention across $O(2n)$ words is very slow.

SentenceBERT Solution: **Siamese network**

Run BERT over each sentence independently

Compute **one vector** ($u$ and $v$) **for each sentence** by (mean or max) pooling over word embeddings or by using CLS token

**Classification tasks:**
concatenate $u$, $v$, and $u-v$, use as input to softmax

**Similarity tasks:**
use the cosine similarity of $u$ and $v$ as similarity score

**Training:** start with BERT, fine-tune Siamese model on task-specific data