midterm review

CS 585, Fall 2019
Introduction to Natural Language Processing
http://people.cs.umass.edu/~miyyer/cs585/

Mohit Iyyer
College of Information and Computer Sciences
University of Massachusetts Amherst
questions from last time...

• grading of HW2 / milestone1 in progress
• midterm!!!!
text classification

- input: some text $\mathbf{x}$ (e.g., sentence, document)
- output: a label $\mathbf{y}$ (from a finite label set)
- goal: learn a mapping function $f$ from $\mathbf{x}$ to $\mathbf{y}$

fyi: basically every NLP problem reduces to learning a mapping function with various definitions of $\mathbf{x}$ and $\mathbf{y}$!
$f$ can be hand-designed rules

- if “won $10,000,000” in $x$, $y = \text{spam}$
- if “CS585 Fall 2019” in $x$, $y = \text{not spam}$

what are the drawbacks of this method?
$f$ can be learned from data

- given **training data** (already-labeled $x,y$ pairs)
  learn $f$ by maximizing the likelihood of the training data
- this is known as **supervised learning**
naive Bayes

• represents input text as a bag of words
• assumption: each word is independent of all other words
• given labeled data, we can use naive Bayes to estimate probabilities for unlabeled data
• **goal:** infer probability distribution that generated the labeled data for each label
class conditional probabilities

Bayes rule (ex: $x =$ sentence, $y =$ label in \{pos, neg\})

$$p(y \mid x) = \frac{p(y) \cdot P(x \mid y)}{p(x)}$$

Our predicted label is the one with the highest posterior probability, i.e.,

$$\hat{y} = \arg \max_{y \in Y} p(y) \cdot P(x \mid y)$$
n-gram LMs
goal: assign probability to a piece of text

- why would we ever want to do this?

- translation:
  - $P(i \text{ flew to the movies}) \lll P(i \text{ went to the movies})$

- speech recognition:
  - $P(i \text{ saw a van}) \llllll P(\text{eyes awe of an})$
Probabilistic Language Modeling

• Goal: compute the probability of a sentence or sequence of words:
  \[ P(W) = P(w_1, w_2, w_3, w_4, w_5, \ldots w_n) \]

• Related task: probability of an upcoming word:
  \[ P(w_5 | w_1, w_2, w_3, w_4) \]

• A model that computes either of these:
  \[ P(W) \text{ or } P(w_n | w_1, w_2, \ldots w_{n-1}) \]
  is called a language model or LM
Markov Assumption

\[ P(w_1w_2\ldots w_n) \approx \prod_{i} P(w_i \mid w_{i-k} \ldots w_{i-1}) \]

• In other words, we approximate each component in the product

\[ P(w_i \mid w_1w_2\ldots w_{i-1}) \approx P(w_i \mid w_{i-k} \ldots w_{i-1}) \]
The Maximum Likelihood Estimate (MLE)
- relative frequency based on the empirical counts on a training set

\[ P(w_i \mid w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})} \]

\[ P(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})} \quad (c \text{ — count}) \]
Perplexity

The best language model is one that best predicts an unseen test set

• Gives the highest P(sentence)

Perplexity is the inverse probability of the test set, normalized by the number of words:

\[ PP(W) = \frac{1}{N} \left( P(w_1w_2...w_N) \right)^{-\frac{1}{N}} \]

Chain rule:

\[ PP(W) = \sqrt[\sqrt{\frac{1}{N}}]{\frac{1}{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1...w_{i-1})}}} \]

For bigrams:

\[ PP(W) = \sqrt[\sqrt{\frac{1}{N}}]{\frac{1}{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}} \]

Minimizing perplexity is the same as maximizing probability
The intuition of smoothing (from Dan Klein)

- When we have sparse statistics:
  \[ P(w \mid \text{denied the}) \]
  - 3 allegations
  - 2 reports
  - 1 claims
  - 1 request
  - 7 total

- Steal probability mass to generalize better
  \[ P(w \mid \text{denied the}) \]
  - 2.5 allegations
  - 1.5 reports
  - 0.5 claims
  - 0.5 request
  - 2 other
  - 7 total
Add-1 estimation is a blunt instrument

- So add-1 isn’t used for N-grams:
  - We’ll see better methods
- But add-1 is used to smooth other NLP models
  - For text classification
  - In domains where the number of zeros isn’t so huge.
Absolute discounting: just subtract a little from each count

- Suppose we wanted to subtract a little from a count of 4 to save probability mass for the zeros
- How much to subtract?
- Church and Gale (1991)’s clever idea
- Divide up 22 million words of AP Newswire
  - Training and held-out set
  - for each bigram in the training set
  - see the actual count in the held-out set!

<table>
<thead>
<tr>
<th>Bigram count in training</th>
<th>Bigram count in heldout set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.0000270</td>
</tr>
<tr>
<td>1</td>
<td>0.448</td>
</tr>
<tr>
<td>2</td>
<td>1.25</td>
</tr>
<tr>
<td>3</td>
<td>2.24</td>
</tr>
<tr>
<td>4</td>
<td>3.23</td>
</tr>
<tr>
<td>5</td>
<td>4.21</td>
</tr>
<tr>
<td>6</td>
<td>5.23</td>
</tr>
<tr>
<td>7</td>
<td>6.21</td>
</tr>
<tr>
<td>8</td>
<td>7.21</td>
</tr>
<tr>
<td>9</td>
<td>8.26</td>
</tr>
</tbody>
</table>
log-linear LMs (and more generally, logistic regression)
The General Problem

• We have some **input domain** $\mathcal{X}$

• Have a finite **label set** $\mathcal{Y}$

• Aim is to provide a **conditional probability** $P(y \mid x)$ for any $x, y$ where $x \in \mathcal{X}, y \in \mathcal{Y}$
Language Modeling

• $x$ is a “history” $w_1, w_2, \ldots, w_{i-1}$, e.g.,

Third, the notion “grammatical in English” cannot be identified in any way with the notion “high order of statistical approximation to English”. It is fair to assume that neither sentence (1) nor (2) (nor indeed any part of these sentences) has ever occurred in an English discourse. Hence, in any statistical

• $y$ is an “outcome” $w_i$
Feature Vector Representations

• Aim is to provide a conditional probability $P(y \mid x)$ for “decision” $y$ given “history” $x$

A feature is some function $\phi(x)$; in LMs $\phi(context)$. Features are often binary indicators; i.e. $\phi(x) \in \{0,1\}$

If you have $m$ features, you can form a feature vector $x \in \mathbb{R}^m$

what could be some useful indicator features for language modeling?
given features $x$, how do we predict the next word $y$?

$$s = Wx + b$$

- **score vector** $s \in \mathbb{R}^{V}$
- **features** $x \in \mathbb{R}^{m}$
- **weight matrix** $W \in \mathbb{R}^{V \times m}$

Each row of $W$ contains weights for a (word $y$, $x$) pair.
how do we obtain probabilities?

\[
s = Wx + b
\]

score vector \( s \in \mathbb{R}^{|V|} \)

weight matrix \( W \in \mathbb{R}^{|V| \times m} \)

features \( x \in \mathbb{R}^m \)

\[
p_i = \frac{e^{s_i}}{\sum_j e^{s_j}}; p = \text{softmax}(s)
\]
“Log-linear”?

\[ p(y|x, W) = \frac{e^{W_{y,x}}}{\sum_{y' \in V} e^{W_{y',x}}} \]

\[ \log p(y|x, W) = W_{y,x} - \log \sum_{y' \in V} e^{W_{y',x}} \]

linear in weights and features…

… except for this!

known as log-sum-exp,

very important for these models

\[ \log p(y|x, W) \propto W_{y,x} \]

why is this true?
what do we have left?

• how do we find the optimal values of $W$ and $b$ for our language modeling problem?

• gradient descent! this involves computing:

  1. a *loss function*, which tells us how good the current values of $W$ and $b$ are on our training data

  2. the partial derivatives $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$
first, an aside: what is the bias $b$?

• Let’s say we have a feature that is always set to 1 regardless of what the input text is.
• This is clearly not an informative feature. However, let’s say it was the only one I had…

first, how many weights do I need to learn for this feature?

okay… what is the best set of weights for it?
Training with softmax and cross-entropy error

• For each training example \{x,y\}, our objective is to maximize the probability of the correct class \ y

• Hence, we minimize the negative log probability of that class:

\[
L = - \log p(y \mid x, W) = - \log \left( \frac{e^{W_y x}}{\sum_{y' \in V} e^{W_{y'} x}} \right)
\]
Background: Why “Cross entropy” error

• Assuming a ground truth (or gold or target) probability distribution that is 1 at the right class and 0 everywhere else: \( p = [0,...,0,1,0,...0] \) and our computed probability is \( q \), then the cross entropy is:

\[
H(p, q) = - \sum_{w \in V} p(w) \log q(w)
\]

• Because of one-hot \( p \), the only term left is the negative log probability of the true class
let’s say I also have the derivatives

\[
\frac{\partial L}{\partial W}, \quad \frac{\partial L}{\partial b}
\]

• the partial derivatives tell us how the loss changes given a change in the corresponding parameter

• we can thus take steps in the negative direction of the gradient to minimize the loss function
word embeddings
why do neural networks work better?

• multiple layer and nonlinearities allow feature combinations that a linear model can’t get
  • e.g., XOR function

• the learned representations of words and contexts are tuned to the prediction problem
  • unlike one-hot vectors
why use vectors to encode meaning?

- computing the similarity between two words (or phrases, or documents) is extremely useful for many NLP tasks

- Q: how **tall** is Mount Everest?
  A: The official **height** of Mount Everest is 29029 ft
one-hot vectors

- we’ve already seen these before in bag-of-words models (e.g., naive Bayes)!
- represent each word as a vector of zeros with a single 1 identifying the index of the word

<table>
<thead>
<tr>
<th>vocabulary</th>
<th>movie = &lt;0, 0, 0, 0, 1, 0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>film = &lt;0, 0, 0, 0, 0, 1&gt;</td>
</tr>
<tr>
<td>hate</td>
<td></td>
</tr>
<tr>
<td>love</td>
<td></td>
</tr>
<tr>
<td>the</td>
<td></td>
</tr>
<tr>
<td>movie</td>
<td></td>
</tr>
<tr>
<td>film</td>
<td></td>
</tr>
</tbody>
</table>

what are the issues of representing a word this way?
all words are equally (dis)similar!

movie = <0, 0, 0, 0, 1, 0>
film = <0, 0, 0, 0, 0, 1>
dot product is zero!
these vectors are orthogonal

how can we compute a vector representation such that the dot product correlates with word similarity?
Word2vec

- Instead of counting how often each word $w$ occurs near "apricot"
- Train a classifier on a binary prediction task:
  - Is $w$ likely to show up near "apricot"?
- We don’t actually care about this task
  - But we'll take the learned classifier weights as the word embeddings
Setup

Let's represent words as vectors of some length (say 300), randomly initialized.

So we start with $300 \times V$ random parameters.

Over the entire training set, we’d like to adjust those word vectors such that we

- Maximize the similarity of the target word, context word pairs $(t,c)$ drawn from the positive data.
- Minimize the similarity of the $(t,c)$ pairs drawn from the negative data.
Skip-Gram Training Data

Training sentence:

... lemon, a tablespoon of apricot jam a pinch ...

\[
c_1 \quad c_2 \quad \text{target} \quad c_3 \quad c_4
\]

Asssume context words are those in +/- 2 word window
Skip-Gram Goal

Given a tuple \((t,c)\) = target, context

- \((\text{apricot}, \text{jam})\)
- \((\text{apricot}, \text{aardvark})\)

Return probability that \(c\) is a real context word:

\[
P(+ | t, c) \\
\]
\[
P(− | t, c) = 1 - P(+ | t, c) \\
\]

\[
\]

\[
\]

\[
\]
How to compute $p(+ | t, c)$?

Intuition:
- Words are likely to appear near similar words
- Model similarity with dot-product!
- $\text{Similarity}(t, c) \propto t \cdot c$

Problem:
- Dot product is not a probability!
- (Neither is cosine)
Turning dot product into a probability

The sigmoid lies between 0 and 1:

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

The classifier

Let's start by thinking about the classification task, and then turn to how to train. Imagine a sentence like the following, with a target word `apricot` and assume we're using a window of ±2 context words:

... lemon, a [tablespoon of apricot jam, a] pinch ...

c1 c2 t c3 c4

Our goal is to train a classifier such that, given a tuple \((t, c)\) of a target word \(t\) paired with a candidate context word (for example \((\text{apricot}, \text{jam})\), or perhaps \((\text{apricot}, \text{aardvark})\)) it will return the probability that \(c\) is a real context word (true for \(\text{jam}\), false for \(\text{aardvark}\)):

\[ P(+ | t, c) \] (6.15)

The probability that word \(c\) is not a real context word for \(t\) is just 1 minus Eq. 6.15:

\[ P(- | t, c) = 1 - P(+) | t, c) \] (6.16)

How does the classifier compute the probability \(P(+ | t, c)\)? The intuition of the skip-gram model is to base this probability on similarity: a word is likely to occur near the target if its embedding is similar to the target embedding. How can we compute similarity between embeddings? Recall that two vectors are similar if they have a high dot product (cosine, the most popular similarity metric, is just a normalized dot product). In other words:

\[ \text{Similarity}(t, c) \approx t \cdot c \] (6.17)

Of course, the dot product \(t \cdot c\) is not a probability, it's just a number ranging from 0 to 1. (Recall, for that matter, that cosine isn't a probability either). To turn the dot product into a probability, we'll use the logistic or sigmoid function \(s(x)\), the fundamental core of logistic regression:

\[ s(x) = \frac{1}{1 + e^{-x}} \]

The probability that word \(c\) is a real context word for target word \(t\) is thus computed as:

\[ P(+ | t, c) = \frac{1}{1 + e^{t \cdot c}} \] (6.19)

The sigmoid function just returns a number between 0 and 1, so to make it a probability we'll need to make sure that the total probability of the two possible events (\(c\) being a context word, and \(c\) not being a context word) sum to 1.

The probability that word \(c\) is not a real context word for \(t\) is thus:

\[ P(- | t, c) = \frac{e^{t \cdot c}}{1 + e^{t \cdot c}} \]

\[ = \frac{1}{1 + e^{-t \cdot c}} \] (6.20)
Turning dot product into a probability

\[
P(+) | t, c) = \frac{1}{1 + e^{-t \cdot c}}
\]

\[
P(- | t, c) = 1 - P(+) | t, c)
\]

\[
= \frac{e^{-t \cdot c}}{1 + e^{-t \cdot c}}
\]
Learning the classifier

Iterative process.

We’ll start with 0 or random weights.

Then adjust the word weights to:
- make the positive pairs more likely
- and the negative pairs less likely

over the entire training set.

Guess what algorithm we’ll use to make this happen?
neural LMs
A fixed-window neural Language Model

output distribution
\[ \hat{y} = \text{softmax}(W_2h + b_2) \]

hidden layer
\[ h = f(W_1c + b_1) \]

concatenated word embeddings
\[ c = [c_1; c_2; c_3; c_4] \]
A RNN Language Model

output distribution

\[ \hat{y} = \text{softmax}(W_2 h^{(t)} + b_2) \]

hidden states

\[ h^{(t)} = f(W_h h^{(t-1)} + W_e c_t + b_1) \]

\( h^{(0)} \) is initial hidden state!

word embeddings

\( c_1, c_2, c_3, c_4 \)

\[ \hat{y}^{(4)} = P(x^{(5)}|\text{the students opened their}) \]
why is this good?

RNN Advantages:
- Can process any length input
- Model size doesn’t increase for longer input
- Computation for step $t$ can (in theory) use information from many steps back
- Weights are shared across timesteps → representations are shared

RNN Disadvantages:
- Recurrent computation is slow
- In practice, difficult to access information from many steps back

$$\hat{y}^{(4)} = P(x^{(5)}|\text{the students opened their})$$
Training a RNN Language Model

• Get a **big corpus of text** which is a sequence of words \( x^{(1)}, \ldots, x^{(T)} \)

• Feed into RNN-LM; compute output distribution \( \hat{y}^{(t)} \) for **every step** \( t \).
  • i.e. predict probability dist of **every word**, given words so far

• **Loss function** on step \( t \) is usual cross-entropy between our predicted probability distribution \( \hat{y}^{(t)} \), and the true next word \( y^{(t)} = x^{(t+1)} \):

\[
J^{(t)}(\theta) = CE(y^{(t)}, \hat{y}^{(t)}) = -\sum_{j=1}^{|V|} y_j^{(t)} \log \hat{y}_j^{(t)}
\]

• Average this to get **overall loss** for entire training set:

\[
J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta)
\]
Training a RNN Language Model

= negative log prob of “students”

Corpus

Loss

47
Training a RNN Language Model

= negative log prob
of “opened”

Corpus

the
C_1

students
C_2

opened
C_3

their
C_4

exams
Training a RNN Language Model

Let $J(\theta)$ be the negative log probability of a word sequence.

Corpus: the students opened their exams ...

$J(1)(\theta)$, $J(2)(\theta)$, $J(3)(\theta)$, $J(4)(\theta)$
Training a RNN Language Model

\[
\begin{align*}
\text{Loss} & \rightarrow J^{(1)}(\theta) \quad J^{(2)}(\theta) \quad J^{(3)}(\theta) \quad J^{(4)}(\theta) \\
\hat{y}^{(1)} & \uparrow W_2 \quad \hat{y}^{(2)} \uparrow W_2 \quad \hat{y}^{(3)} \uparrow W_2 \quad \hat{y}^{(4)} \uparrow W_2 \\
h^{(0)} & \rightarrow W_h \quad h^{(1)} \rightarrow W_h \quad h^{(2)} \rightarrow W_h \quad h^{(3)} \rightarrow W_h \quad h^{(4)} \rightarrow W_h \\
W_e & \rightarrow \text{the } C_1 \quad \text{students } C_2 \quad \text{opened } C_3 \quad \text{their } C_4 \quad \text{exams } ... 
\end{align*}
\]

= negative log prob of "exams"
Training a RNN Language Model

\[
J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta)
\]

Corpus: the, students, opened, their, exams

\[h^{(0)} \rightarrow W_h \rightarrow W_e \rightarrow \hat{y}^{(1)}\]
\[h^{(1)} \rightarrow W_h \rightarrow W_e \rightarrow \hat{y}^{(2)}\]
\[h^{(2)} \rightarrow W_h \rightarrow W_e \rightarrow \hat{y}^{(3)}\]
\[h^{(3)} \rightarrow W_h \rightarrow W_e \rightarrow \hat{y}^{(4)}\]
\[\ldots \rightarrow W_h \rightarrow W_e \rightarrow \hat{y}^{(T)}\]
Training a RNN Language Model

• However: Computing loss and gradients across entire corpus is too expensive!

• **Recall:** Stochastic Gradient Descent allows us to compute loss and gradients for small chunk of data, and update.

• \(\rightarrow\) In practice, consider \(x^{(1)}, \ldots, x^{(T)}\) as a sentence

\[
J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta)
\]

• Compute loss \(J(\theta)\) for a sentence (actually usually a batch of sentences), compute gradients and update weights. Repeat.
okay... enough with the unconditional LMs. let’s switch to conditional LMs!

we’ll start with *machine translation*
today: neural MT

• we’ll use French (\(f\)) to English (\(e\)) as a running example

• **goal**: given French sentence \(f\) with tokens \(f_1, f_2, \ldots, f_n\) produce English translation \(e\) with tokens \(e_1, e_2, \ldots, e_m\)

  \[
  \text{is } n \text{ always equal to } m? \\
  \]

• **real goal**: compute \(\arg\max_e p(e | f)\)
today: neural MT

- let’s use an NN to directly model \( p(e | f) \)

\[
p(e | f) = p(e_1, e_2, \ldots, e_m | f)
\]

\[
= p(e_1 | f) \cdot p(e_2 | e_1, f) \cdot p(e_3 | e_2, e_1, f) \cdot \ldots
\]

\[
= \prod_{i=1}^{m} p(e_i | e_1, \ldots, e_{i-1}, f)
\]

how does this formulation relate to the language models we discussed previously?
Neural Machine Translation (NMT)
The sequence-to-sequence model

Encoding of the source sentence.
Provides initial hidden state for Decoder RNN.

Encoder RNN produces an encoding of the source sentence.
Neural Machine Translation (NMT)

The sequence-to-sequence model

Encoding of the source sentence. Provides initial hidden state for Decoder RNN.

Decoder RNN is a Language Model that generates target sentence conditioned on encoding.

Encoder RNN produces an encoding of the source sentence.
Sequence-to-sequence: the bottleneck problem

Encoding of the source sentence. This needs to capture all information about the source sentence. Information bottleneck!

Target sentence (output)

The poor don’t have any money <END>

Source sentence (input)

les pauvres sont démunis

<START> the poor don’t have any money <END>
The solution: **attention**

- **Attention mechanisms** (Bahdanau et al., 2015) allow the decoder to focus on a particular part of the source sequence at each time step
  - Conceptually similar to **word alignments**
How does it work?

• in general, we have a single *query* vector and multiple *key* vectors. We want to score each query-key pair

in machine translation with RNNs, what are the queries and keys?
**Sequence-to-sequence with attention**

Encoder RNN

Source sentence (input)

les pauvres sont démunis

Decoder RNN

Attention scores

Query 1: decoder, first time step

dot product with keys (encoder hidden states)
Sequence-to-sequence with attention

On this decoder timestep, we’re mostly focusing on the first encoder hidden state ("les")

Take softmax to turn the scores into a probability distribution

Source sentence (input)

les pauvres sont démunis
Sequence-to-sequence with attention

Use the attention distribution to take a **weighted sum** of the encoder hidden states.

The attention output mostly contains information the hidden states that received high attention.

Encoder RNN

Source sentence (input)

Attention scores

Attention distribution

Attention output

les pauvres sont démunis

<START>
Sequence-to-sequence with attention

Encoder RNN

Source sentence (input)  

les pauvres sont démunis  

Decoder RNN

Attention distribution

Attention scores

Attention output

the  

\( \hat{y}_1 \)

Concatenate attention output with decoder hidden state, then use to compute \( \hat{y}_1 \) as before

\( \# \)
Sequence-to-sequence with attention

Encoder RNN

Source sentence (input)

les pauvres sont démunis

Decoder RNN

Attention scores

Attention distribution

Attention output

poor

\( \hat{y}_2 \)

decoder, second time step
Attention is great

- Attention significantly improves NMT performance
  - It’s very useful to allow decoder to focus on certain parts of the source
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with vanishing gradient problem
  - Provides shortcut to faraway states
- Attention provides some interpretability
  - By inspecting attention distribution, we can see what the decoder was focusing on
  - We get alignment for free!
  - This is cool because we never explicitly trained an alignment system
  - The network just learned alignment by itself
decoding

• given that we trained a seq2seq model, how do we find the most probable English sentence?

• more concretely, how do we find

\[
\arg \max \prod_{i=1}^{m} p(e_i | e_1, \ldots, e_{i-1}, f)
\]

• can we enumerate all possible English sentences e?
can we just do attention and get rid of recurrence?
Self-attention as an encoder!

(core component of Transformer)

<table>
<thead>
<tr>
<th></th>
<th>this</th>
<th>is</th>
<th>an</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>this</td>
<td>dark gray</td>
<td>light gray</td>
<td>white</td>
<td>white</td>
</tr>
<tr>
<td>is</td>
<td>light gray</td>
<td>dark gray</td>
<td>white</td>
<td>white</td>
</tr>
<tr>
<td>an</td>
<td>white</td>
<td>white</td>
<td>dark gray</td>
<td>white</td>
</tr>
<tr>
<td>example</td>
<td>white</td>
<td>light gray</td>
<td>white</td>
<td>deep purple</td>
</tr>
</tbody>
</table>

Vaswani et al., 2017

figure: Graham Neubig
Self-attention

[Vaswani et al. 2017]
Self-attention

[Vaswani et al. 2017]
Self-attention

[Vaswani et al. 2017]
Self-attention

\[ \text{[Vaswani et al. 2017]} \]
Self-attention

[Vaswani et al. 2017]
Self-attention

[Vaswani et al. 2017]
Self-attention

[Vaswani et al. 2017]

\[ M \]

\[ Q \]

\[ K \]

\[ V \]

Layer \( p \)

Nobel committee awards Strickland who advanced optics

options advanced who Strickland awards committee Nobel
Self-attention

\[ M \]

\[ A \]

\[ Q \]

\[ K \]

\[ V \]

Layer \( p \)

Nobel committee awards Strickland who advanced optics

[optics advanced who Strickland awards committee Nobel]

[Vaswani et al. 2017]
Multi-head self-attention

[Vaswani et al. 2017]
Multi-head self-attention

[Vaswani et al. 2017]
Multi-head self-attention

\[ M_H \]
\[ M_f \]

Awards committee Nobel

Vaswani et al. 2017
Multi-head self-attention

- Nobel committee awards Strickland who advanced optics
- [Vaswani et al. 2017]
Multi-head self-attention

Layer J

Multi-head self-attention + feed forward

Layer p

Multi-head self-attention + feed forward

Layer 1

Multi-head self-attention + feed forward

Nobel committee awards Strickland who advanced optics
Positional encoding

EMBEDDING WITH TIME SIGNAL

POSITIONAL ENCODING

EMBEDDINGS

INPUT

Je

suis

étudiant
Last major missing piece:

- Decoder self-attention masking
Byte pair encoding (BPE)

- Deal with rare words / large vocabulary by instead using subword tokenization

<table>
<thead>
<tr>
<th>system</th>
<th>sentence</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>health research institutes</td>
</tr>
<tr>
<td>reference</td>
<td>Gesundheitsforschungsinstitute</td>
</tr>
<tr>
<td>WDict</td>
<td>Forschungsinstitute</td>
</tr>
<tr>
<td>C2-50k</td>
<td>Forschungs</td>
</tr>
<tr>
<td>BPE-60k</td>
<td>Gesundheits</td>
</tr>
<tr>
<td>BPE-J90k</td>
<td>Gesundheits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>system</th>
<th>sentence</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>asinine situation</td>
</tr>
<tr>
<td>reference</td>
<td>dumme Situation</td>
</tr>
<tr>
<td>WDict</td>
<td>asinine situation → UNK → asinine</td>
</tr>
<tr>
<td>C2-50k</td>
<td>as</td>
</tr>
<tr>
<td>BPE-60k</td>
<td>as</td>
</tr>
<tr>
<td>BPE-J90K</td>
<td>as</td>
</tr>
</tbody>
</table>

Sennrich et al., ACL 2016
transfer learning
What is transfer learning?

• **In our context:** take a network trained on a task for which it is easy to generate labels, and adapt it to a different task for which it is harder.

• **In computer vision:** train a CNN on ImageNet, transfer its representations to every other CV task

• **In NLP:** train a really big language model on billions of words, transfer to every NLP task!
History of Contextual Representations

- **ELMo: Deep Contextual Word Embeddings, AI2 & University of Washington, 2017**

**Train Separate Left-to-Right and Right-to-Left LMs**

**Apply as “Pre-trained Embeddings”**

Existing Model Architecture
Unidirectional context
Build representation incrementally

Bidirectional context
Words can “see themselves”
**Solution:** Mask out $k\%$ of the input words, and then predict the masked words

- We always use $k = 15\%$

What are the pros and cons of increasing $k$?
• Use 30,000 WordPiece vocabulary on input.
• Each token is sum of three embeddings
• Single sequence is much more efficient.
Empirical advantages of Transformer vs. LSTM:

1. Self-attention == no locality bias
   - Long-distance context has “equal opportunity”

2. Single multiplication per layer == efficiency on TPU
   - Effective batch size is number of words, not sequences
Fine-Tuning Procedure
HMMs / sequence modeling
These are all **log-linear models**

Naive Bayes

HMMs

Logistic Regression

Linear-chain CRFs

Fig. 2.3 Diagram of the relationship between naive Bayes, logistic regression, HMMs, linear-chain CRFs, generative models, and general CRFs.

One perspective for gaining insight into the difference between generative and discriminative modeling is due to Minka [80]. Suppose we have a generative model $p_g$ with parameters $\theta$. By definition, this takes the form

$$p_g(y, x; \theta) = p_g(y; \theta)p_g(x|y; \theta).$$

(2.10)

But we could also rewrite $p_g$ using Bayes rule as

$$p_g(y, x; \theta) = p_g(x; \theta)p_g(y|x; \theta),$$

(2.11)

where $p_g(x; \theta)$ and $p_g(y|x; \theta)$ are computed by inference, i.e.,

$$p_g(x; \theta) = P_y p_g(y, x; \theta)$$

and

$$p_g(y|x; \theta) = p_g(y, x; \theta)/p_g(x; \theta).$$

Now, compare this generative model to a discriminative model over the same family of joint distributions. To do this, we define a prior $p(x)$ over inputs, such that $p(x)$ could have arisen from $p_g$ with some parameter setting. That is,

$$p(x) = p(x; \theta_0) = P_y p_g(y, x|\theta_0).$$

We combine this with a conditional distribution $p_c(y|x; \theta)$ that could also have arisen from $p_g$, that is,

$$p_c(y|x; \theta) = p_g(y, x; \theta)/p_g(x; \theta).$$

Then the resulting distribution is

$$p_c(y, x) = p_c(x; \theta_0)p_c(y|x; \theta).$$

(2.12)

By comparing (2.11) with (2.12), it can be seen that the conditional approach has more freedom to fit the data, because it does not require
HMM Recapitulation

**HMM Definition**

Assume $K$ parts of speech, a lexicon size of $V$, a series of observations \( \{x_1, \ldots, x_N\} \), and a series of unobserved states \( \{z_1, \ldots, z_N\} \).

- $\pi$ A distribution over start states (vector of length $K$):
  \[ \pi_i = p(z_1 = i) \]

- $\theta$ Transition matrix (matrix of size $K$ by $K$):
  \[ \theta_{i,j} = p(z_n = j | z_{n-1} = i) \]

- $\beta$ An emission matrix (matrix of size $K$ by $V$):
  \[ \beta_{j,w} = p(x_n = w | z_n = j) \]

Two problems: How do we move from data to a model? (Estimation)
How do we move from a model and unlabeled data to labeled data? (Inference)
Training Sentences

\[
x = \text{tokens} \quad \begin{array}{cccc}
\text{x} & \text{here} & \text{come} & \text{old} & \text{flattop}
\end{array}
\]

\[
z = \text{POS tags} \quad \begin{array}{cccc}
\text{z} & \text{MOD} & \text{V} & \text{MOD} & \text{N}
\end{array}
\]

\[
a \quad \text{a crowd of people stopped and stared}
\]

\[
\text{DET} & \text{N} & \text{PREP} & \text{N} & \text{V} & \text{CONJ} & \text{V}
\]

\[
gotta \quad \text{gotta get you into my life}
\]

\[
\text{V} & \text{V} & \text{PRO} & \text{PREP} & \text{PRO} & \text{V}
\]

\[
\text{and} \quad \text{and I love her}
\]

\[
\text{CONJ} & \text{PRO} & \text{V} & \text{PRO}
\]
Training Sentences

here come old flattop
MOD V MOD N

a crowd of people stopped and stared
DET N PREP N V CONJ V

gotta get you into my life
V V PRO PREP PRO N

and I love her
CONJ PRO V PRO
**HMM Estimation**

**Training Sentences**

Here come old flattop

MOD V MOD N

A crowd of people stopped and stared

DET N PREP N V CONJ V

Gotta get you into my life

V V PRO PREP PRO N

And I love her

CONJ PRO V PRO
Viterbi Algorithm

- Given an unobserved sequence of length $L$, \( \{x_1, \ldots, x_L\} \), we want to find a sequence \( \{z_1 \ldots z_L\} \) with the highest probability.
- It’s impossible to compute $K^L$ possibilities.
- So, we use dynamic programming to compute most likely tags for each token subsequence from 0 to $t$ that ends in state $k$.
- Memoization: fill a table of solutions of sub-problems
- Solve larger problems by composing sub-solutions
- Base case:
  \[
  \delta_1(k) = \pi_k \beta_{k,x_i}
  \]  
- Recursion:
  \[
  \delta_n(k) = \max_j (\delta_{n-1}(j) \theta_{j,k}) \beta_{k,x_n}
  \]