Carnegie Mellon University
Language
Technologies
Institute

11-411/11-611 Natural Language Processing

Neural Language Models

David R. Mortensen

February 21, 2023

Language Technologies Institute

Learning Objectives

At the end of this lecture, you should be able to do the following things:

- Generalize the notion of language models beyond the ngram models that we have discussed thus far
- State four advantages of neural LMs over ngram LMs
- State one disadvantage of NLMs (runtime efficiency)
- Apply your knowledge of neural nets (feedforward neural nets, RNNs, LSTMs) to language modeling architectures
- Explain how a feedforward neural language model is applied and trained
- Explain the relationship between embeddings and LMs
- Explain the advantages of RNNs over feedforward neural nets for language modeling
- Explain the advantage of LSTMs over RNNs
- Describe masked language models

Units—Building Blocks of Neural Networks

Introducing Feedforward Language Models

We Start with Feedforward LMs because They Are Simple

- To start, we will look at language models built on feedforward neural networks
- In practice, it is not common to use such LMs
- Instead, RNNs or the Transformer are used as the basis for most neural language models
- However, feedforward neural nets are easy to understand and will allow us to discuss the basic concepts involved

Defining Feedforward LMs

Input A representation at time t of some number of previous words $(w_{t-1}, w_{t-2}, \ldots)$

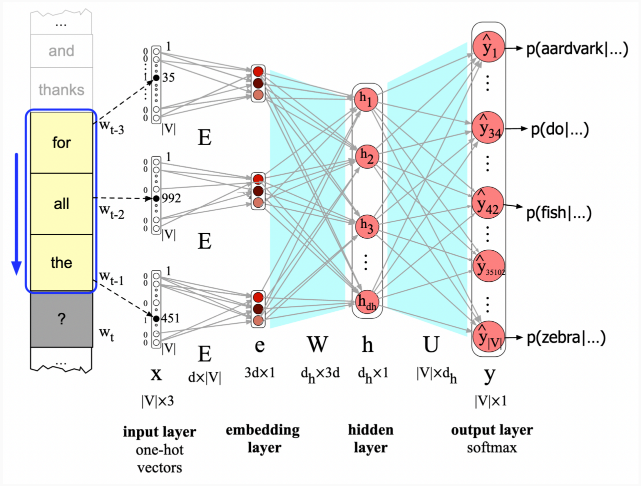
Output A probability distribution over possible next words

Such a language model is like an n-gram language model in that it approximates the probability of a word given the **entire** prior context $P(w_t|w_1 : t-1)$ by approximating based on the N previous words:

$$P(w_t|w_1, \ldots, w_{t-1}) \approx P(w_t|w_t - N + 1, \ldots, w_{t-1})$$

The Markov Assumption

Calculating a Probability Distribution over w_t Using a Simple Feedforward LM



1. Take N context words at each timestep, converting each to a d-dimensional EMBEDDING and concatenating them together (yielding a $1 \times Nd$ unit imput layer
2. Multiply by W Multiply these units by a weight matrix W
3. Apply activation function element-wise to produce a hidden layer h
4. Multiply by U, another weight matrix
5. Apply softmax, predicting at each node i the probability that the new word w_i will be the vocabulary word V_i

Why Neural LMs work better than N-gram LMs

Training data:

We've seen: I have to make sure that the cat gets fed.

Never seen: dog gets fed

Test data:

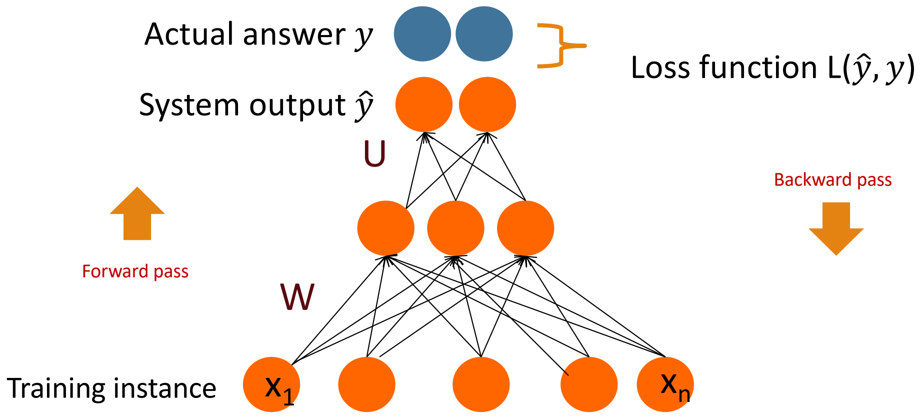
I forgot to make sure that the dog gets ___

N-gram LM can't predict "fed"!

Neural LM can use similarity of "cat" and "dog" embeddings to generalize and predict “fed” after dog

Training Neural Models

Intuition: training a 2-layer Network



Remember stochastic gradient descent from the logistic regression lecture—find gradient and optimize

The Intuition Behind Training a 2-Layer Network

For every training tuple (x, y)

1. Run **forward** computation to find the estimate $\hat{y}$
2. Run **backward** computation to update weights
    - For every output node
        - Compute the loss L between true y and estimated $\hat{y}$
        - For every weight w from the hidden layer to the output layer: update the weights
    - For every hidden node
        - Assess how much blame it deserves for the current answer
        - From every weight w from the input layer to the hidden layer
        - Update the weight

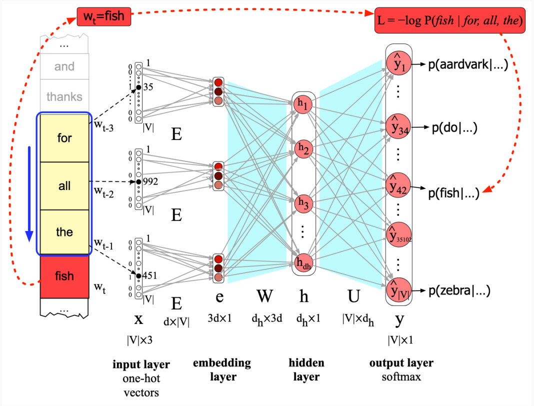
Computing the gradient requires finding the derivative of the loss with respect to each weight in every layer of the network.

Error backpropagation through computation graphs.

Training (Feedforward) Neural Language Models

Training a Feedforward LM: Forward Pass

1. **Select three embeddings from E** take words w_{t-3}, w_{t-2}, and w_{t-1}, create three ONE-HOT VECTORS for them, and multiply each by embedding matrix E, yielding the PROJECTION LAYER or EMBEDDING LAYER. Concatenate the three embeddings.
2. **Multiply by W** and add b (the bias term). Pass the result through the ReLU (or other) activation function to get hidden layer h
3. **Multiply by U**
4. **Apply softmax** resulting in a layer where each node i in the output layer estimates the probability $P(w_t = i | w_{t-1}, w_{t-2}, w_{t-3})$



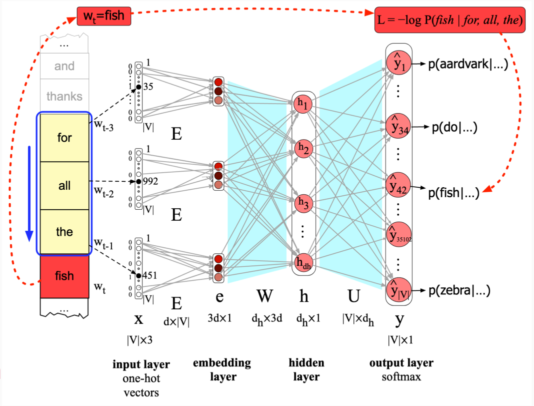
Training a Feedforward LM: Backward Pass

1. Optional: freeze embedding layer (appropriate for some tasks)
2. Define loss (e.g., **cross-entropy loss**)

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_c$$

   (in our case, the negative log of the output probability corresponding to the correct class, c)

3. Update parameters $\theta = \mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{b}$ via **gradient descent**, using **error backpropagation** on the **computation graph** to compute the gradient



More Detail on the Backward Pass

Concatenate all sentences in the training set (starting with random weights) and move through this text iteratively, predicting each word w_t. Use cross entropy (that is, negative log likelihood) at each w_t:

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i \tag{1}$$

where i is the correct class. $\hat{y}_i$ means the probability that the model assigns the correct next word w_t:

$$L_{CE} = -\log p(w_t | w_{t-1}, ..., w_{t-n+1}) \tag{2}$$

So the parameter update for the stochastic gradient descent for this loss from set s to $s+1$ is:

$$\theta^{s+1} = \theta^s - \eta \frac{\partial[-\log p(w_t | w_{t-1}, ..., w_{t-n+1})]}{\partial \theta} \tag{3}$$

Computing this is gradient and backpropagating through $\theta = \mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{b}$ is trivial and is left as an exercise for PyTorch.

There are Two Benefits to Training a Neural Language Model

If you train a neural language model, you get two things:

1. An algorithm that will allow you to predict the next word in a sequence
2. A set of embeddings E that can be used to represent words in other tasks (assuming that you did not freeze the embedding layer)

Language Models with Recurrent Neural Networks

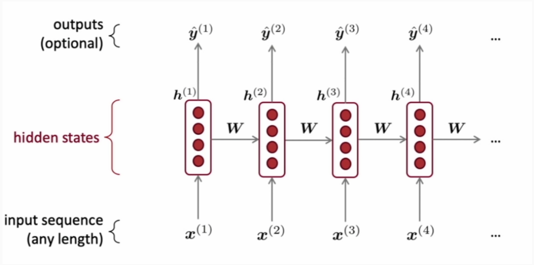
FFNNs take an input of fixed dimensions—a fixed number of features, a fixed number of tokens

The number tokens in a text—even a sentence—can be **arbitrarily large** (or short)

RNNs help us address this issue

Refresher: RNNs

- RNNs are neural networks with a recurrent structure
    - Each step takes an input, emits an output and pass internal state on to the next step
    - Hidden state multiplied by weight matrix W at each step
- Good for modeling sequences


outputs (optional)
$\hat{y}^{(1)}$ $\hat{y}^{(2)}$ $\hat{y}^{(3)}$ $\hat{y}^{(4)}$ ...
$h^{(1)}$ $h^{(2)}$ $h^{(3)}$ $h^{(4)}$
hidden states
W W W W ...
input sequence (any length)
$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$ ...

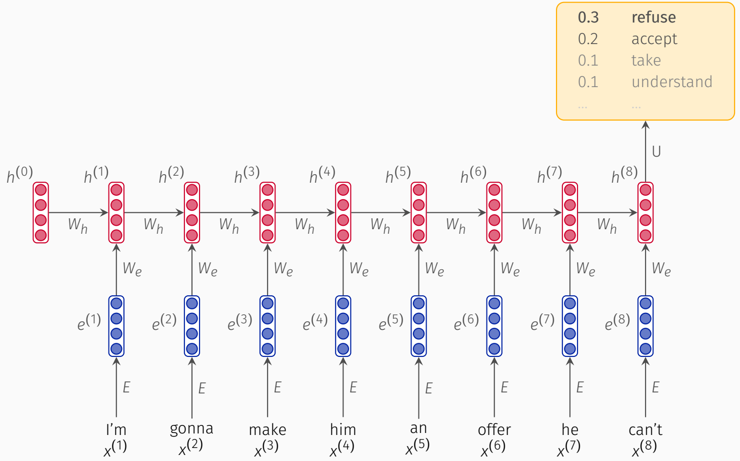
An RNN Language Model

output distribution
$\hat{y}^{(t)} = \text{softmax}(Uh^{(t)} + b_2) \in \mathbb{R}^{|V|}$

hidden states
$h^{(t)} = \sigma(W_h h^{(t-1)} + W_e e^{(t)} + b_1)$
$h^{(0)}$ is the initial hidden state

word embeddings
$e^{(t)} = Ex^{(t)}$

one-hot vectors
$x^{(t)} \in \mathbb{R}^{|V|}$


0.3 refuse
0.2 accept
0.1 take
0.1 understand
… …
U
$h^{(0)}$ $h^{(1)}$ $h^{(2)}$ $h^{(3)}$ $h^{(4)}$ $h^{(5)}$ $h^{(6)}$ $h^{(7)}$ $h^{(8)}$
W_h
W_e
$e^{(1)}$ $e^{(2)}$ $e^{(3)}$ $e^{(4)}$ $e^{(5)}$ $e^{(6)}$ $e^{(7)}$ $e^{(8)}$
E
I'm $x^{(1)}$
gonna $x^{(2)}$
make $x^{(3)}$
him $x^{(4)}$
an $x^{(5)}$
offer $x^{(6)}$
he $x^{(7)}$
can't $x^{(8)}$

Training an RNN Language Model

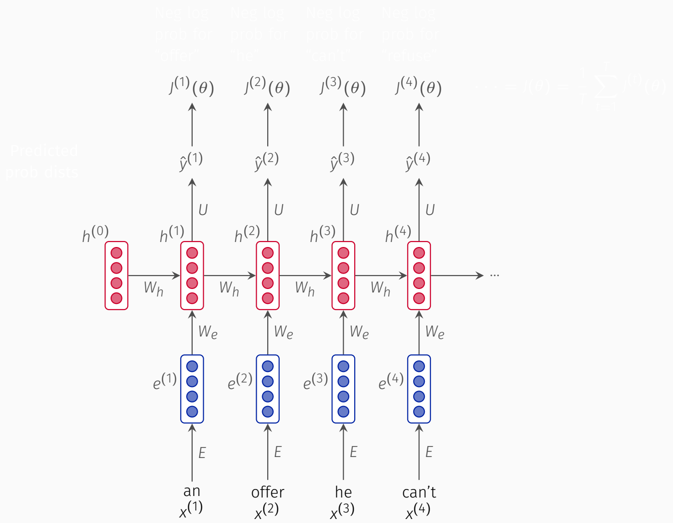
- Get a big corpus of text, which is a sequence of words $x^{(1)}, \dots, x^{(T)}$
- Feed it into the RNN-LM, computing output distribution $^{(t)}$ for every step t.
- Loss function on step t is **cross-entropy** between the predicted probability distribution $\hat{\mathbf{y}}^{(t)}$ and the true next word $y^{(t)}$ (one-hot for $x^{(t+1)}$):

$$J^{(t)}(\theta) = CE(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) = -\sum_{w \in V} \mathbf{y}_w^{(t)} \log \hat{\mathbf{y}}_w^{(t)} = -\log \hat{\mathbf{y}}_{\mathbf{x}_{t+1}}^{(t)}$$

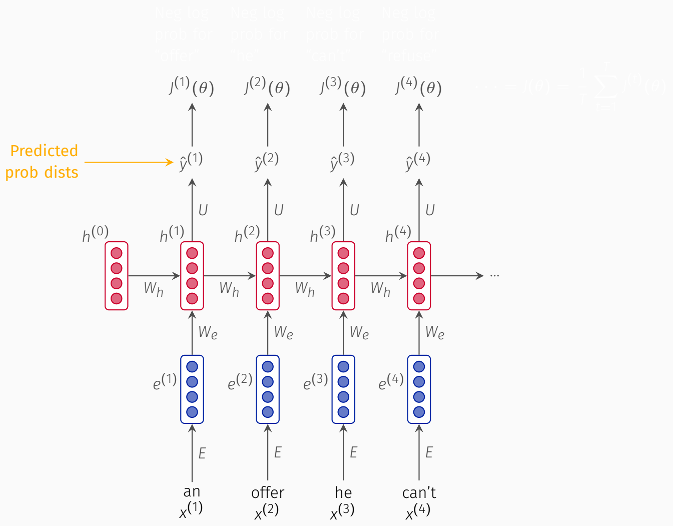
- Average this to get overall loss for the entire training set:

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\hat{\mathbf{y}}_{x_{t+1}}^{(t)}$$

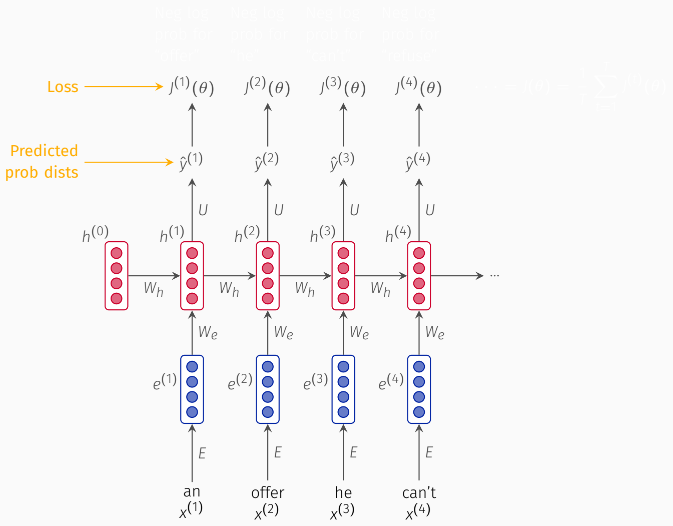
Training an RNN Language Model



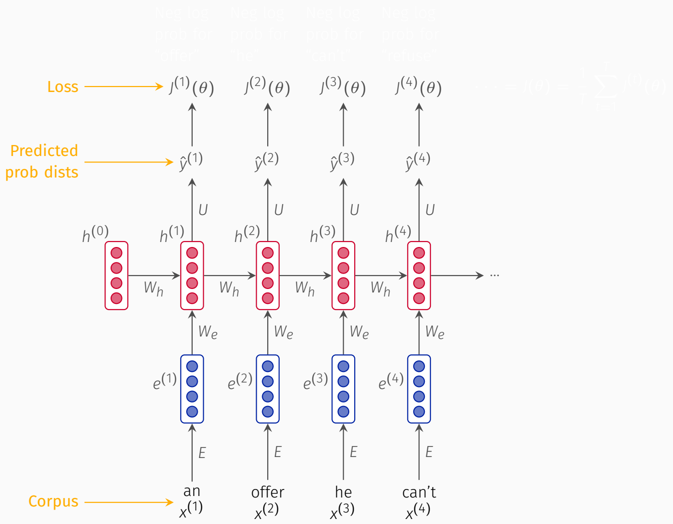
Training an RNN Language Model



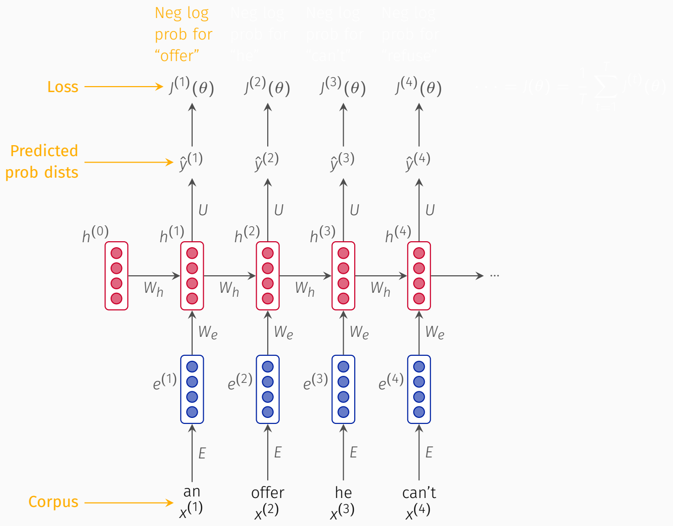
Training an RNN Language Model



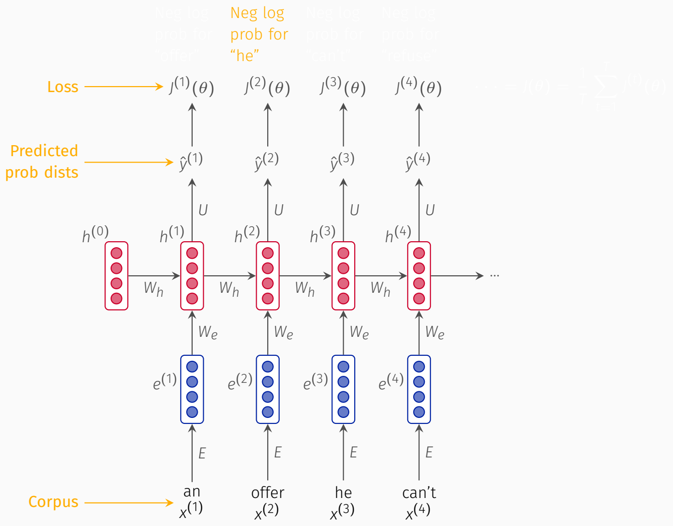
Training an RNN Language Model



Training an RNN Language Model

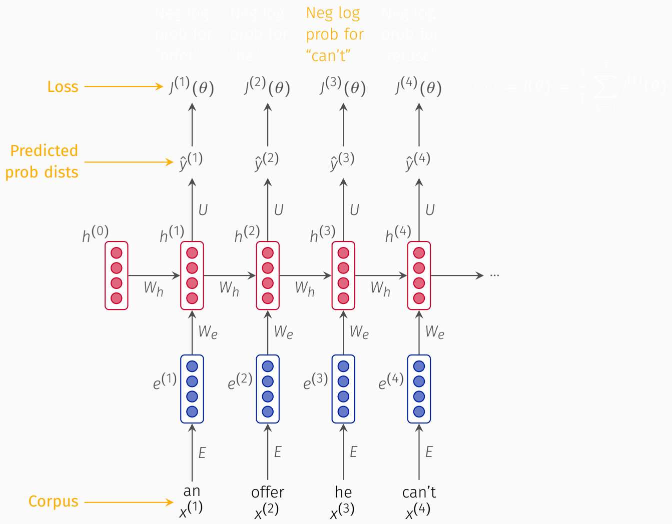


Training an RNN Language Model



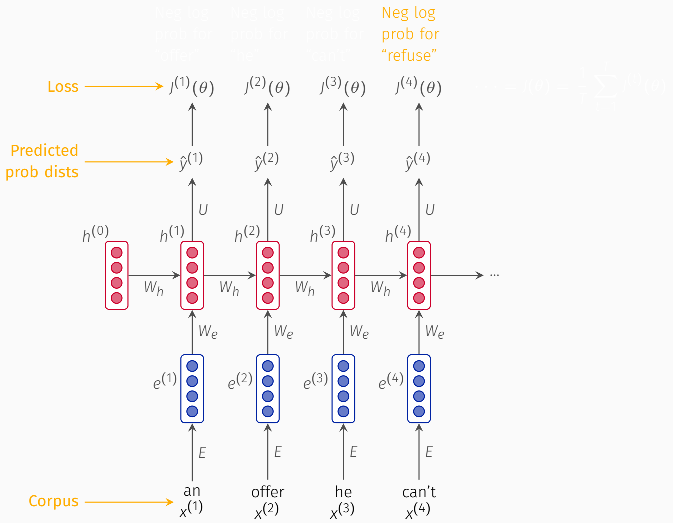
Neg log prob for "offer"
Neg log prob for "he"
Neg log prob for "can't"
Neg log prob for "refuse"
Loss
$J^{(1)}(\theta)$ $J^{(2)}(\theta)$ $J^{(3)}(\theta)$ $J^{(4)}(\theta)$
$\cdots = J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$
Predicted prob dists
$\hat{y}^{(1)}$ $\hat{y}^{(2)}$ $\hat{y}^{(3)}$ $\hat{y}^{(4)}$
U
$h^{(0)}$ $h^{(1)}$ $h^{(2)}$ $h^{(3)}$ $h^{(4)}$
W_h
...
W_e
$e^{(1)}$ $e^{(2)}$ $e^{(3)}$ $e^{(4)}$
E
Corpus
an $x^{(1)}$
offer $x^{(2)}$
he $x^{(3)}$
can't $x^{(4)}$

Training an RNN Language Model

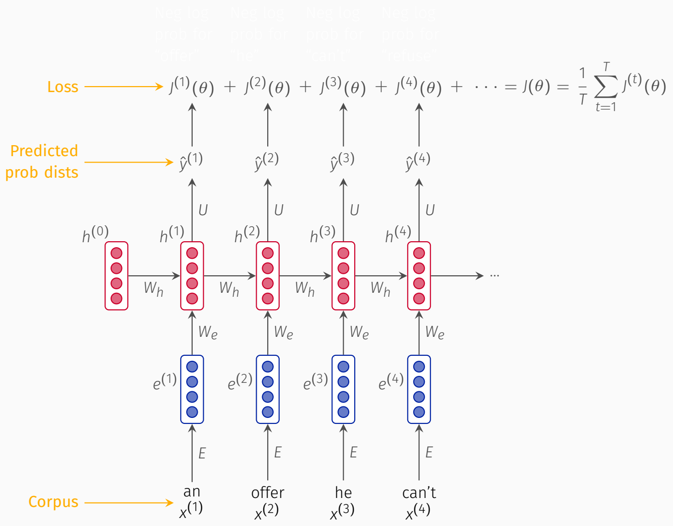


Neg log prob for "offer"
Neg log prob for "he"
Neg log prob for "can't"
Neg log prob for "refuse"
Loss
$J^{(1)}(\theta)$ $J^{(2)}(\theta)$ $J^{(3)}(\theta)$ $J^{(4)}(\theta)$ $\cdots = J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$
Predicted prob dists
$\hat{y}^{(1)}$ $\hat{y}^{(2)}$ $\hat{y}^{(3)}$ $\hat{y}^{(4)}$
U
$h^{(0)}$ $h^{(1)}$ $h^{(2)}$ $h^{(3)}$ $h^{(4)}$
W_h
...
W_e
$e^{(1)}$ $e^{(2)}$ $e^{(3)}$ $e^{(4)}$
E
Corpus
an $x^{(1)}$
offer $x^{(2)}$
he $x^{(3)}$
can't $x^{(4)}$

Training an RNN Language Model



Training an RNN Language Model



Computing Loss and Gradients in Practice

- In principle, we could compute loss and gradients across the whole corpus $(x^{(1)}, \dots, x^{(T)})$ but that would be incredibly expensive!

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

- Instead, we usually treat $x^{(1)}, \dots, x^{(T)}$ as a document, or even a sentence
- This works much better with **Stochastic Gradient Descent**, which lets us compute loss and gradients for little chunks and update as we go.
- Actually, we do this in batches: compute $J(\theta)$ for a batch of sentences; update weights; repeat.

We Will Skip the Details of Backpropogation in RNNs for Now

- The fact that training RNNs involves backpropagation over timesteps, summing as you go, means that it (the **backpropagation through time** algorithm) is a bit more complicated than backpropagation in feedforward neural networks.
- We will skip these details for now, but you will want to learn them if you are doing serious work with RNNs.

Advantages of RNN Language Models

- Input can be of an arbitrary length
- Computation can use information from many steps back (in principle)
- Longer inputs do not mean larger model sizes
- Same weights applied at every time step—**symmetry**

Disadvantages of RNN LMs

- Recurrent computation is slow
  - Computing $h^{(t)}$ requires computing $h^{(t-1)}$ which requires computing $h^{(t-2)}$
  - Cannot be parallelized
- In practice, it is difficult to access information from many steps back (cf. the VANISHING GRADIENT PROBLEM)

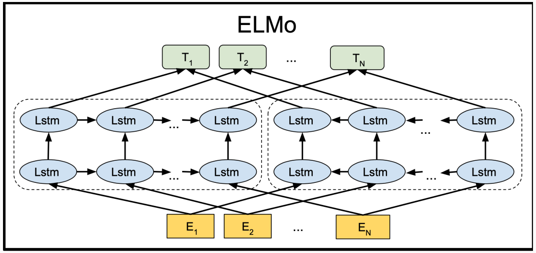
LSTMs Address (but Do not Solve) the Vanishing and Exploding Gradient Problems

- LSTMs: Long Short Term Memory
- Process data sequentially, but keep hidden state through time
- Still subject, at some level, to vanishing gradients, but to a lesser degree that traditional RNNs
- Widely used in language modeling

Prehistory of Pretrained Language Models

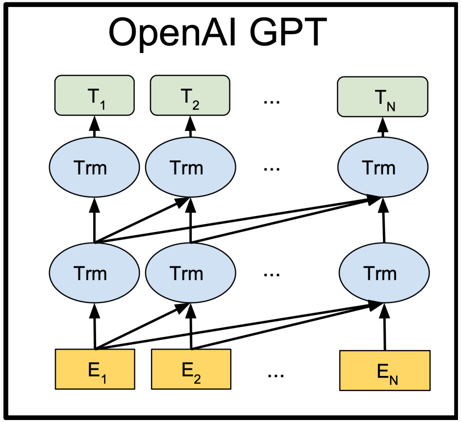
ELMo

“ELMo is a deep contextualized word representation that models both (1) complex characteristics of word use (e.g., syntax and semantics), and (2) how these uses vary across linguistic contexts (i.e., to model polysemy). These word vectors are learned functions of the internal states of a deep bidirectional language model (biLM), which is pre-trained on a large text corpus. They can be easily added to existing models and significantly improve[d] the state of the art across a broad range of challenging NLP problems, including question answering, textual entailment and sentiment analysis.”



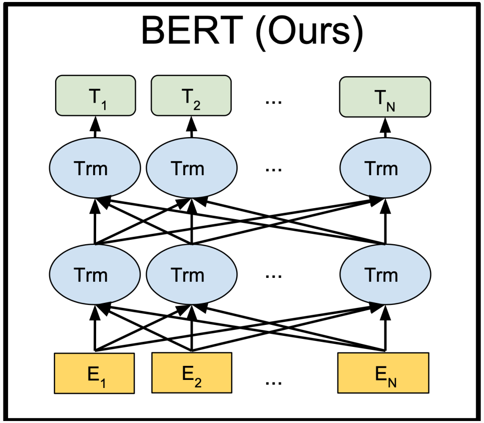
GPT (Generative Pre-Trained Transformer)

The original GPT, released in 2018, combined transformers and unsupervised pre-training (ingredients that would also be central to BERT, GPT-2, and GPT-3). A broad view of its architecture is given below:



BERT

BERT innovated beyond GPT in being bi-directional (like ELMo, only more so):



BERT also introduced a new training regimen.

Transformers and BERT

GPT and BERT Rely upon Transformers

- The ELMo architecture is based around LSTMs
- BERT and GPT incorporate a newer architectural idea: the Transformer
- Transformers have revolutionized many NLP tasks

Transformers Improved upon RNNs and CNNs

- Google introduced Transformers in 2017
- At that time, most neural NLP models were based on:
    - RNNs
    - CNNs
- These were good
- For many tasks, Transformers were better

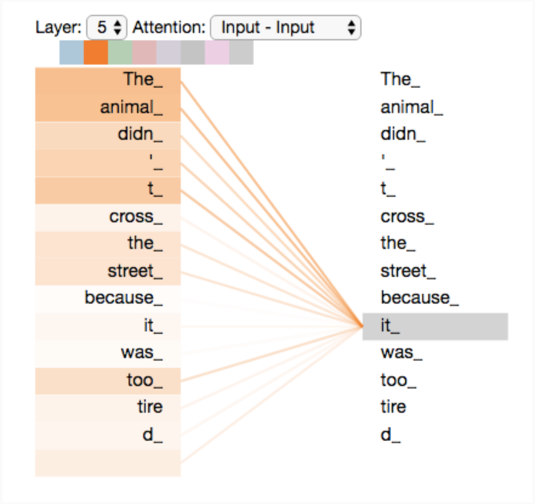
Attention: All You Need?

Attention:

- Each output element is connected to each input element
- The weightings between them are calculated dynamically, based on their connection
- Attention has long been important in NLP models for machine translation, etc.

Attention is fundamental to Transformers.

Okay, *Self-Attention* Is All You Need!



Take the sentence: “The animal didn’t cross the street because it was too tired”. What is the antecedent of *it*?

Self-attention allows the model to “attend” to all of the other positions and to process each position (including *the* and *animal*) to help it better encode the pronoun *it*.

You can compare this to the hidden state in an RNN—it conveys information about other words in the sequence to the position one is currently processing.

Transformers rely on self-attention.

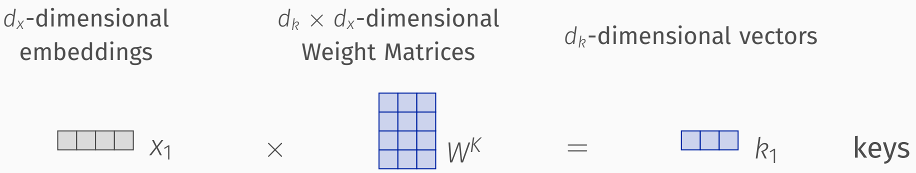
Computing Self-Attention, Step One: Compute Key, Query, and Value Vectors

d_x-dimensional embeddings

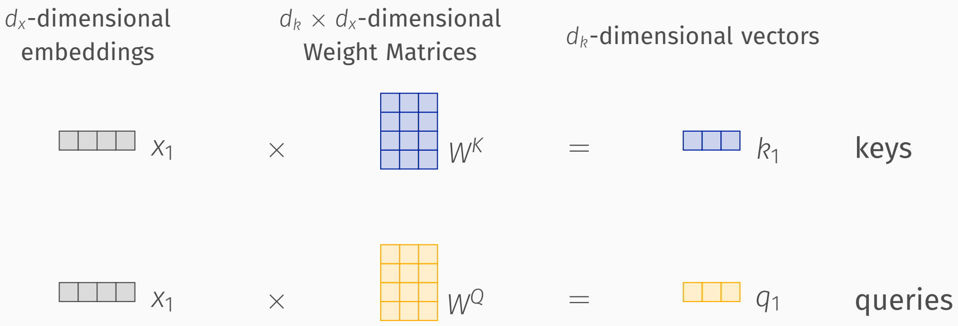
$d_k \times d_x$-dimensional Weight Matrices

d_k-dimensional vectors

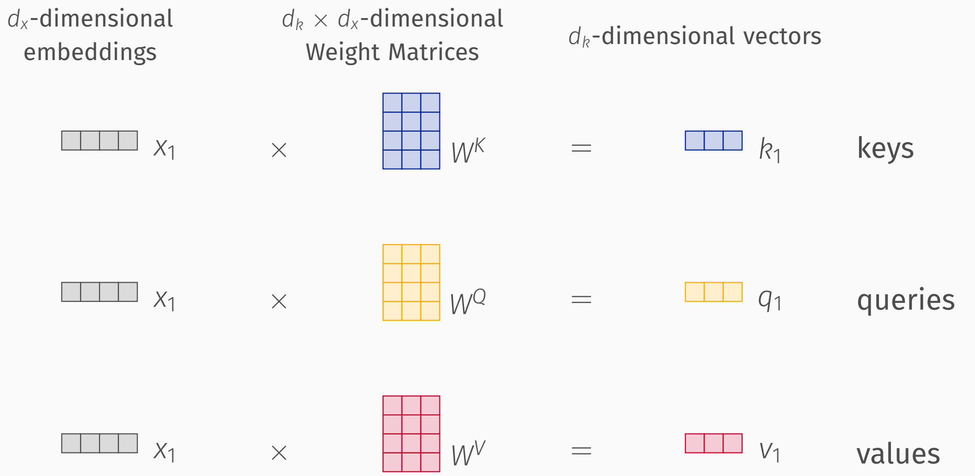
Computing Self-Attention, Step One: Compute Key, Query, and Value Vectors


d_x-dimensional embeddings
$d_k \times d_x$-dimensional Weight Matrices
d_k-dimensional vectors
x_1
$\times$
W^K
$=$
k_1
keys

Computing Self-Attention, Step One: Compute Key, Query, and Value Vectors


d_x-dimensional embeddings
$d_k \times d_x$-dimensional Weight Matrices
d_k-dimensional vectors
x_1
$\times$
W^K
$=$
k_1
keys
x_1
$\times$
W^Q
$=$
q_1
queries

Computing Self-Attention, Step One: Compute Key, Query, and Value Vectors



d_x-dimensional embeddings
$d_k \times d_x$-dimensional Weight Matrices
d_k-dimensional vectors
$x_1 \times W^K = k_1$
keys
$x_1 \times W^Q = q_1$
queries
$x_1 \times W^V = v_1$
values

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

query-key dot product

divide by $\sqrt{d_k}$

softmax

$\times$ value

sum

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

We

x_1

query-key dot product $q_1 \cdot k_1 = 13$

divide by $\sqrt{d_k}$ $\frac{13}{\sqrt{64}} = 1.63$

softmax

$\times$ value

sum

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash
	x_1	x_2
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$
softmax		
$\times$ value		
sum		

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash	our
	x_1	x_2	x_3
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$	$q_1 \cdot k_3 = 20$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$	$\frac{20}{\sqrt{64}} = 2.5$
softmax			
$\times$ value			
sum			

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash	our	cats
	x_1	x_2	x_3	x_4
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$	$q_1 \cdot k_3 = 20$	$q_1 \cdot k_4 = 12$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$	$\frac{20}{\sqrt{64}} = 2.5$	$\frac{12}{\sqrt{64}} = 1.5$
softmax				
$\times$ value				
sum				

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash	our	cats
	x_1	x_2	x_3	x_4
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$	$q_1 \cdot k_3 = 20$	$q_1 \cdot k_4 = 12$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$	$\frac{20}{\sqrt{64}} = 2.5$	$\frac{12}{\sqrt{64}} = 1.5$
softmax	0.12	0.48	0.29	0.10
$\times$ value				
sum				

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

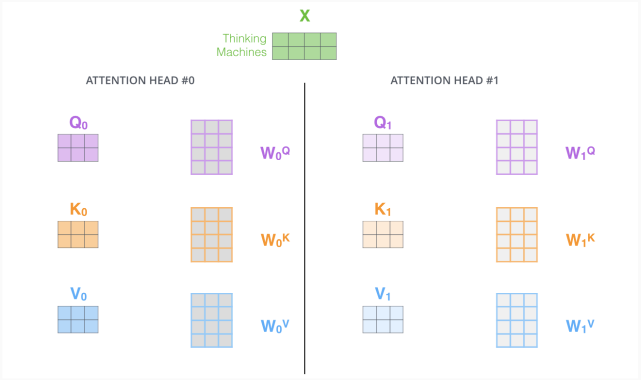
	We	wash	our	cats
	x_1	x_2	x_3	x_4
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$	$q_1 \cdot k_3 = 20$	$q_1 \cdot k_4 = 12$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$	$\frac{20}{\sqrt{64}} = 2.5$	$\frac{12}{\sqrt{64}} = 1.5$
softmax	0.12	0.48	0.29	0.10
$\times$ value	$0.12 \times v_1$	$0.48 \times v_2$	$0.29 \times v_3$	$0.10 \times v_4$
sum				

Computing Self-Attention, Step Two: Weighted Sum of Value Vectors

	We	wash	our	cats
	x_1	x_2	x_3	x_4
query-key dot product	$q_1 \cdot k_1 = 13$	$q_1 \cdot k_2 = 24$	$q_1 \cdot k_3 = 20$	$q_1 \cdot k_4 = 12$
divide by $\sqrt{d_k}$	$\frac{13}{\sqrt{64}} = 1.63$	$\frac{24}{\sqrt{64}} = 3.0$	$\frac{20}{\sqrt{64}} = 2.5$	$\frac{12}{\sqrt{64}} = 1.5$
softmax	0.12	0.48	0.29	0.10
$\times$ value	$0.12 \times v_1$	$0.48 \times v_2$	$0.29 \times v_3$	$0.10 \times v_4$
sum	z_1			

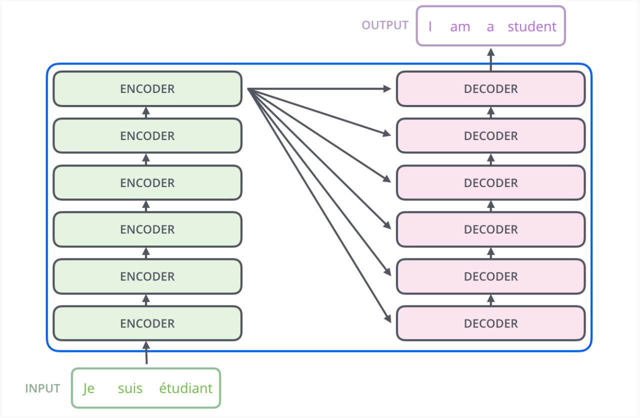
Multi-Headed Attention Expands Transformer Models' Ability to Focus on Different Positions

Maintain distinct weight matrices for each attention head—distinct representational subspaces:



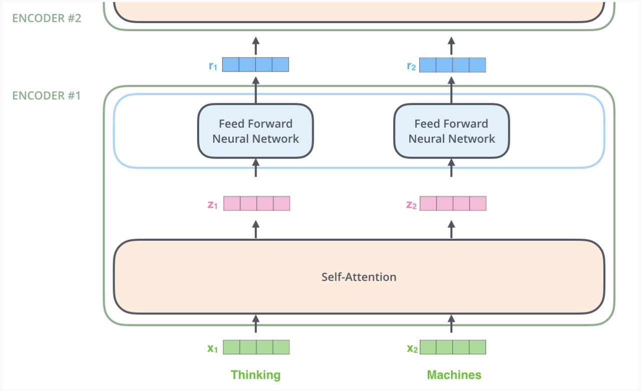
Transformers: Stacks of Encoders and Decoders

A transformer is a stack of ∼6 encoders and decoders. The encoders are identical in structure but do not share weights.



The Structure of the Encoders

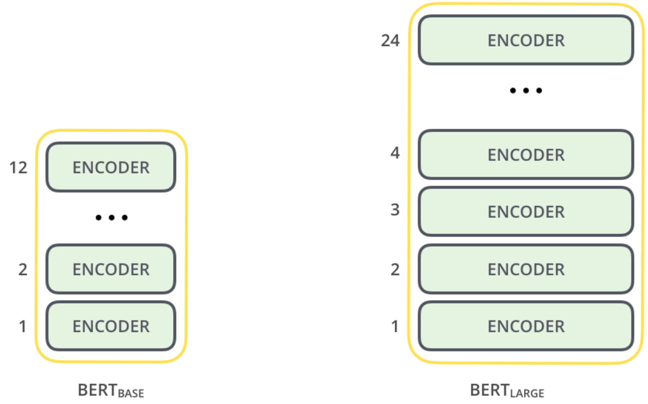
Each encoder takes input vectors. For the first encoder, these are the embeddings of the input words. For other encoders, these are the outputs of the preceding encoders. The inputs pass through a self attention layer, then through a feed forward neural net.



Bidirectional Encoder Representations from Transformers (BERT)

Because BERT is a language model, it only needs encoders so I won’t talk about decoders here.

BERT Is Basically a Trained Transformer Encoder Stack



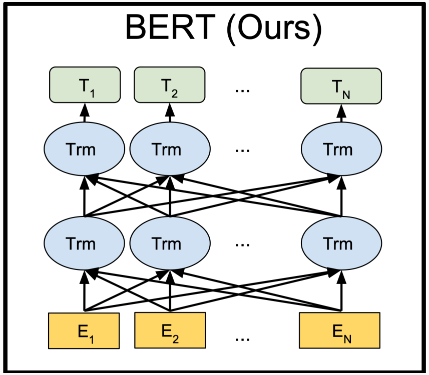
BERT Involves Many Uninteresting Numbers

BERT Comes in Two Sizes: BASE and LARGE. The parameters of these models are as follows:

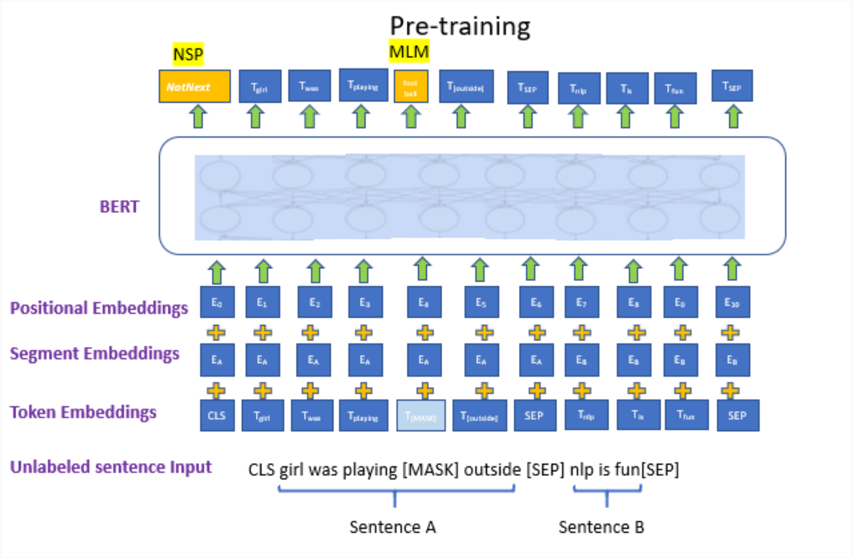
	$\text{BERT}_{\text{BASE}}$	$\text{BERT}_{\text{LARGE}}$
encoder layers	12	24
hidden units	768	1024
attention heads	12	16

BERT Is Bidirectional (or, Rather, Non-Directional)

- Traditional LMs are unidirectional—given some sequence of words $w_1, w_2, \ldots, w_t$, they try to predict w_{t+1}.
- BERT uses the whole sequence to predict words in that sequence.



Pretraining BERT: MLM and NSP



The Cloze Task

- The cloze task comes from psycholinguistics (the branch of linguistics and cognitive science that uses experimental methods to study how language works in human brains).
- It is a fill-in-the-blank task:

He drove the yellow ______ into the front of our house.

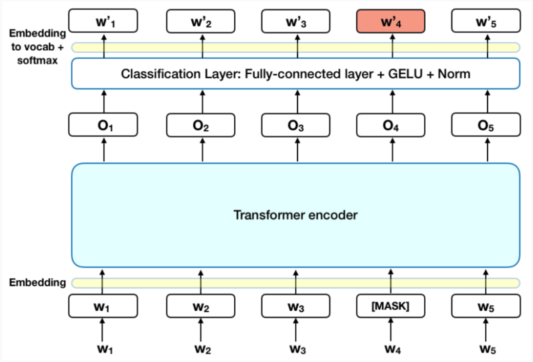
- Subjects are presented with these frames and asked to fill in the missing words
- This allows experimenters to assess what a speaker understands about grammar, semantics, etc.
- According to the original BERT paper, this task provided the inspiration for BERT's masked language modeling (MLM) training task.
- But compare various kinds of denoising algorithms.

BERT is Trained to Perform Masked Language Modeling

- Since, in BERT's innards, everything is literally connected to everything, each word would indirectly see itself
- Enter masking!
- In training, random word are concealed from BERT using the `[MASK]` token
- BERT is trained to guess these masked words
- Take the sentence: "the girl was playing outside."
    - Suppose that a `[MASK]` token is inserted in place of *outside*, thus masking it
    - We then have "the girl was playing `[MASK]`."
    - The model now cannot "see" *outside*; instead it is trained to generate it based on context
    - This is part of why BERT can be so good at representing words according to the context in which they occur (not just the distribution of identically-spelled tokens)

How BERT Is Trained to Perform Masked Language Modeling

1. Add classification layer
2. Multiply output vectors by embedding layer → vocabulary dimension
3. Calculate probabilities using softmax



w'1
w'2
w'3
w'4
w'5
Embedding to vocab + softmax
Classification Layer: Fully-connected layer + GELU + Norm
O1
O2
O3
O4
O5
Transformer encoder
Embedding
w1
w2
w3
[MASK]
w5
w1
w2
w3
w4
w5

GELU: A Note

GELU (from the last slide) is a Gaussian Error Linear Unit:

$$\mathrm{GELU}(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

where $\Phi(x)$ is the standard Gaussian cumulative distribution function and erf is the error function (a normalized form of the Gaussian function).

GELU is an activation function that is similar, in some respects, to ReLU. Unlike ReLU, GELU non-linearly weights inputs by their percentile (ReLU gates inputs by their sign).

BERT is also Trained to Perform Next Sentence Prediction

- Suppose you are given two pairs of sentences:
  1. Good pair
     1.1 Pickles are delicious.
     1.2 However, cucumbers make me burp.
  2. Bad pair
     2.1 Pickles are delicious.
     2.2 The NASDAC was up 10 points today amid heavy trading.
- For each pair, you (a human) can tell whether the first and the second are likely to occur in sequence
- BERT is trained to do the same
- As a result, BERT learns patterns that exist above the level of the sentence

Questions?