Mining Unstructured Data
10. RNN & Language Modeling

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
FIB

Outline

Outline

Language Modeling

Language Modeling is the task of predicting what word comes next.

- More formally: given a sequence of words, compute the probability distribution of the next word:

$$P(w_t|w_1, w_2, \ldots, w_{t-1})$$

  where w_t can be any word in the vocabulary.
- A system that does this is called a Language Model.

Ex: The students opened their w_n

- $w_n^1 =$ books
- $w_n^2 =$ laptops
- $w_n^3 =$ exams

You use LM Every Day (I)

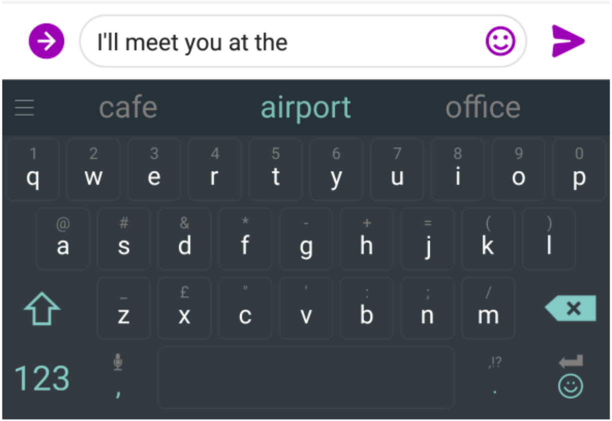


Figure: LM Example: Prediction in Google Keyboard

You use LM Every Day (II)

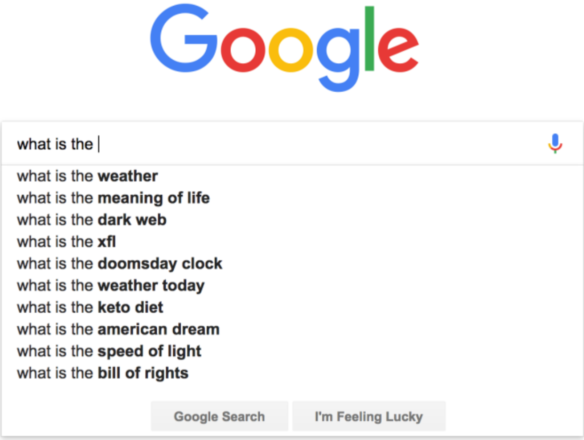


Figure: LM Example: Suggestions in Google Search

Hidden Markov Models for Language Modeling

- **Language Modeling:** predicting the probability of a sequence of words $W = w_1, w_2, ..., w_T$ given the context $C = w_{1-k}, ..., w_{t-1}$.
- **Remember**: We can use Hidden Markov Models (HMMs) to model **sequences of words** - or model language.
- Proposed Approach: **bigram HMM**. That is, use Hidden Markov Models to model the sequence of words as a sequence of hidden states, where each state represents a particular word and every edge represents the transition probability between two words.

Hidden Markov Models for Language Modeling (II)

- HMM Formulas:
  - State transition probabilities: $a_{i,j} = P(q_t = j | q_{t-1} = i)$
  - Emission probabilities: $b_i(w_t) = P(w_t | q_t = i)$
  - Initial state probabilities: $\pi_i = P(q_1 = i)$
- Applied to a Language Model:
  - Joint probability of words and states:

$$P(W, Q) = \pi_{q_1} b_{q_1}(w_1) \prod_{t=2}^{T} a_{q_{t-1},q_t} b_{q_t}(w_t)$$

  - Probability of words given context:

$$P(W|C) = \sum_{q_1,\ldots,q_T} P(W, Q) = \sum_{q_1} \pi_{q_1} b_{q_1}(w_1) \sum_{q_2} a_{q_1,q_2} b_{q_2}(w_2$$

  - Probability of next word given previous words:

$$P(w_t | w_{1:t-1}) \propto \sum_{q_t} a_{q_{t-1},q_t} b_{q_t}(w_t)$$

N-gram Language Models

- Language Modeling predicts the probability distribution of the next word in a given sequence of words.
- Pre-deep learning: **n-gram Language Model**
  - A n-gram is a chunk of n-consecutive words.
  - Example n-grams:
    1. Unigram: "the", "students", "opened", "their"
    2. Bigram: "the students", "students opened"
    3. Trigram: "the students opened", "students opened their"
    4. 4-gram: "the students opened their"
  - Collect statistics about how frequent different n-grams are and use them to predict the next word.
- Deep learning: Recurrent Neural Network, Transformer...

N-gram Language Models (II)

- Markov assumption: Language Modeling depends only on the preceding n-1 words.
- We can compute n-gram and (n-1)-gram probabilities by counting them in a large corpus.
- In Language Modeling, we want to predict the probability distribution of the next word given the previous words.
- Conditional probability is defined as:

$$P(w_n|w_{n-1}, w_{n-2}, ..., w_1) = \frac{P(w_n, w_{n-1}, w_{n-2}, ..., w_1)}{P(w_{n-1}, w_{n-2}, ..., w_1)}$$

- We can use n-gram Language Models to estimate the conditional probability of the next word given the preceding words.

N-gram Language Models (III)

$$P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)}) = P(\boldsymbol{x}^{(t+1)}|\overbrace{\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}}^{n\text{-1 words}})$$

(assumption)

Figure: Markov Assumption for n-gram models

prob of a n-gram

prob of a (n-1)-gram

$$= \frac{\boxed{P(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}{\boxed{P(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}}$$

Figure: Components of $P(w_n|w_{n-1},w_{n-2},...,w_1)$

N-gram Language Models (IV)

~~as the proctor started the clock, the~~ students opened their ______

discard

condition on this

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\text{count(students opened their } \boldsymbol{w})}{\text{count(students opened their)}}$$

- For example, suppose that in the corpus:
  1. "students opened their" occurred 1000 times
  2. "students opened their books"occurred 400 times: $p(books|students\_opened\_their) = 0.4$
  3. "students opened their exams" occurred 100 times: $p(exams|students\_opened\_their) = 0.1$

Outline

Sparsity Problem in N-gram Language Models

Sparsity Problem 1

Problem: What if *"students opened their w"* never occurred in data? Then w has probability 0!

(Partial) Solution: Add small δ to the count for every $w \in V$. This is called *smoothing*.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\boxed{\text{count}(\text{students opened their } \boldsymbol{w})}}{\boxed{\text{count}(\text{students opened their})}}$$

Sparsity Problem 2

Problem: What if *"students opened their"* never occurred in data? Then we can't calculate probability for *any* w!

(Partial) Solution: Just condition on *"opened their"* instead. This is called *backoff*.

Sparsity Problem in N-gram Language Models (II)

- **Remember**, solution 1 - **Add-k smoothing**: add a small constant k to the count of each n-gram.

$$P_{\text{add-k}}(w_n|w_{n-1},\ldots,w_{n-k+1}) = \frac{c(w_{n-k+1},\ldots,w_n)+k}{c(w_{n-k+1},\ldots,w_{n-1})+Vk}$$

  - $c(w_{n-k+1},\ldots,w_n)$ is the count of the n-gram $w_{n-k+1},\ldots,w_n$ in the corpus.
  - $c(w_{n-k+1},\ldots,w_{n-1})$ is the count of the (n-1)-gram $w_{n-k+1},\ldots,w_{n-1}$ in the corpus.
  - V is the vocabulary size
  - k is the smoothing parameter
  - Therefore, the probability of an n-gram with zero count in the corpus is never zero, and the sum of the probabilities over all possible n-grams is always equal to 1.

Sparsity Problem in N-gram Language Models (III)

- Solution 2 - **backoff**: recursively estimating the probability of an n-gram using lower-order n-grams when the count of the higher-order n-gram is zero or very small.

$$P_{\text{bo}}(w_n|w_{n-1},\dots,w_{n-k+1}) =$$

$$\begin{cases} \alpha_{w_{n-k+1},\dots,w_{n-1}} P(w_n|w_{n-1},\dots,w_{n-k+2}) & \text{if } c(w_{n-k+1},\dots,w_n)>0 \\ \beta_{w_{n-k+2},\dots,w_{n-1}} P_{\text{bo}}(w_n|w_{n-1},\dots,w_{n-k+2}) & \text{otherwise} \end{cases}$$

Where:

- $\alpha_{w_{n-k+1},\dots,w_{n-1}}$ and $\beta_{w_{n-k+2},\dots,w_{n-1}}$ are normalization constants that ensure that the probabilities sum to 1.
- $P(w_n|w_{n-1},\dots,w_{n-k+2})$ is the probability of the (n-1)-gram $w_{n-k+2},\dots,w_{n-1}$

Sparsity Problem in N-gram Language Models (IV)

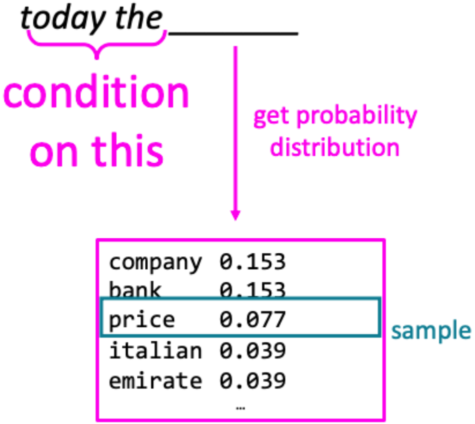
Storage: Need to store count for all *n*-grams you saw in the corpus.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\boxed{\text{count}(\text{students opened their } \boldsymbol{w})}}{\text{count}(\text{students opened their})}$$

Outline

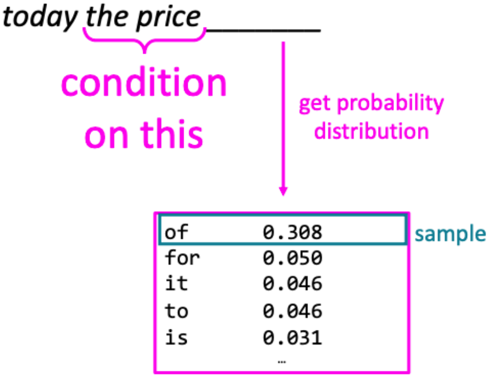
Generation with n-gram Language Models

You can also use Language Models to generate text:



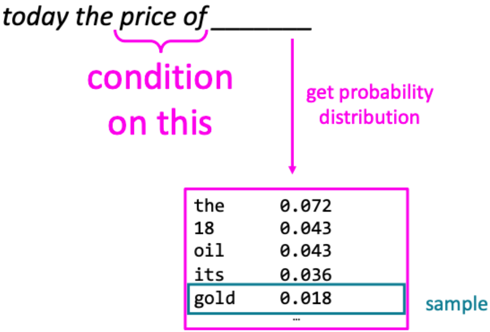
Generation with n-gram Language Models (II)

You can also use Language Models to generate text:



Generation with n-gram Language Models (III)

You can also use Language Models to generate text:



Generation with n-gram Language Models (IV)

You can also use Language Models to generate text:

Today the price of gold per ton, while production of shoe lasts and the shoe industry, the bank intervened just after it considered and rejected an IMF demand to rebuild depleted European stocks, sept 30 end primary 76 c a share.

- Grammatically Correct!
- But incoherent. We need to consider more than 3 words at a time if we want to model language well. But increasing n worsens the sparsity problem and increases model size.

Outline

Neural Language Model

- Recall the Language Modeling task:
  - Input: sequence of words: $w_1, w_2, ..., w_n$
  - Output: probability distribution of the next words: $P(w_{n+1}|w_1, w_2, ..., w_n)$
- How about a window-based neural model?
  - We saw this applied to Named Entity Recognition:

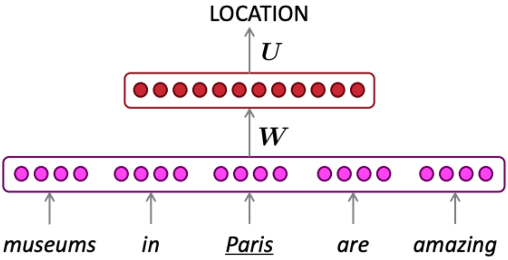


Figure: Window-based neural model for Named Entity Recognition

Neural Language Model (II)

output distribution
$$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings
$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

words / one-hot vectors
$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$$

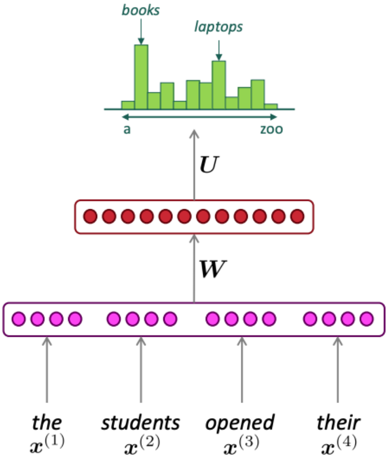


Figure: Fixed window language model. Approximately: Y. Bengio, et al. (2000/2003): A Neural Probabilistic Language Model

Neural Language Model (III)

- Improvements over n-gram LM:
  - No sparsity problem
  - Don't need to store all observed n-grams
- Remaining problems
  - Fixed window is too small
  - Enlarging window enlarges W
  - Each word vector gets multiplied by different weights in W.

Recurrent Neural Networks

We need a neural architecture that can process any length input.

- N-gram Language Model: $P(w_n|w_{n-1}, w_{n-2}, ..., w_{n-N+1})$
- Window-based neural model: $P(w_{n+1}|w_n, w_{n-1}, ..., w_{n-m+1})$
- Recurrent Neural Network: $\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$

Outline

Recurrent Neural Networks for Language Modeling

- RNN language models are a type of language model that use RNNs to predict the next word in a sequence
- **Core idea:** Apply the same weights W repeatedly

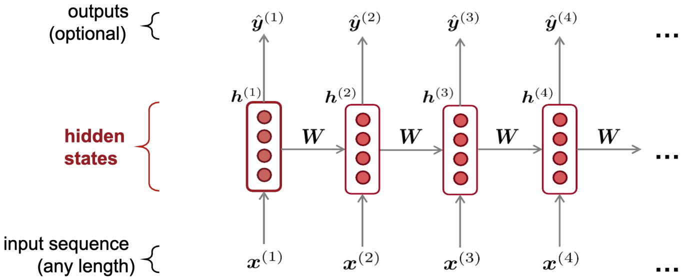


Figure: Representation of a basic RNN

RNN Basics

Remember that:

- RNNs are designed to model sequential data by processing one element of the sequence at a time, while maintaining an internal state
- At each time step t, the RNN takes in an input vector x_t and a hidden state vector h_{t-1}, and produces an output vector y_t and a new hidden state vector h_t
- The hidden state vector serves as a memory of the previous elements in the sequence, allowing the RNN to capture long-term dependencies

RNN Language Modeling

- In RNN language modeling, the input sequence consists of words $x_1, x_2, \ldots, w_n$
- At each time step, the RNN takes in the current x_t and the previous hidden state vector h_{t-1}, and produces a probability distribution over the vocabulary of possible next words
- The output vector y_t is a probability distribution over the vocabulary
- The RNN is trained to minimize the negative log-likelihood of the true next word given the previous words in the sequence

$$y_t = \text{softmax}(W_{hy} h_t + b_y)$$

$$\mathcal{L} = -\sum_{t=1}^{T} \log y_{t,\text{true}}$$

RNN Language Modeling (II)

RNN For Language Modeling
Recurrent Neural Networks for Language Modeling
RNN with Memory Units
Other RNNs
ELMO

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \mathrm{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

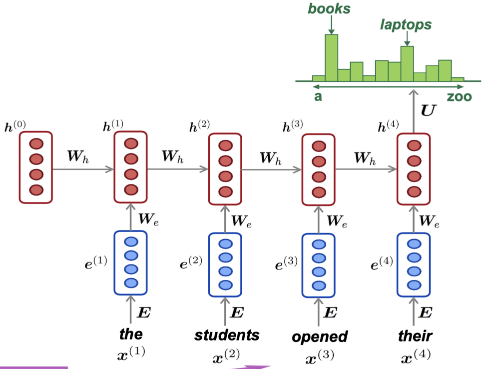
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$



Note: this input sequence could be much longer now!

RNN Language Modeling (III)

RNN advantages:

- Can process any length input
- Computation for step t can (in theory) use information from many steps back
- Model size doesn't increase for longer input
- Same weights applied on every timestep, so there is symmetry in how inputs are processed

RNN disadvantages:

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

Outline

Training a RNN Language Model

- Get a big corpus of text which is a sequence of words
- Feed into RNN-LM; compute output distribution for every step t
  - i.e. predict probability dist of every word, given words so far
- Loss function on step t is cross-entropy between predicted probability distribution P_t and the true next word w_{t+1} (one-hot for w_{t+1}):

$$\mathcal{L}_t = -\sum_{i=1}^{|V|} 1_{\{w_{t+1}=i\}} \log P_t(i|w_1, \ldots, w_t)$$

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

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i}\mathcal{L}_t = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i}\sum_{j=1}^{|V|} 1_{\{w_{t+1}=j\}} \log P_t(j|w_1, \ldots, w_t)$$

Training a RNN Language Model (II)

- **However**: Computing loss and gradients across entire corpus is too expensive!
- In practice, consider as a sentence (or a document)
- Recall: **Stochastic Gradient Descent** allows us to compute loss and gradients for small chunk of data, and update.
- Compute loss $\mathcal{L}_s$ for a sentence (actually, a batch of sentences), compute gradients and update weights. Repeat.
  - Stochastic Gradient Descent update:

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_t} \mathcal{L}_s$$

Outline

Evaluating Language Models

- The standard evaluation metric for Language Models is perplexity.
- This is equal to the exponential of the cross-entropy loss:

$$\text{Perplexity}(P) = \exp\left\{\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i}\mathcal{L}_t\right\}$$

$$= \exp\left\{-\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i}\sum_{j=1}^{|V|} 1_{\{w_{t+1}=j\}} \log P_t(j|w_1,\ldots,w_t)\right\}$$

- Lower perplexity is better
  - Intuitively, perplexity measures how surprised the model is to see the next word in a sequence
  - A lower perplexity means that the model is better at predicting the next word in the sequence, i.e., it is less surprised by the next word

Evaluating Language Models (II)

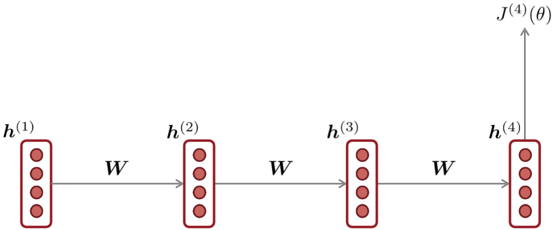
- Perplexity can be seen as the Inverse probability of the corpus according to the LM:

$$\text{Perplexity}(P) = \left(\prod_{i=1}^{N} \prod_{t=1}^{T_i} P(w_{t+1}|w_1, \ldots, w_t) \right)^{-\frac{1}{N \cdot T}}$$

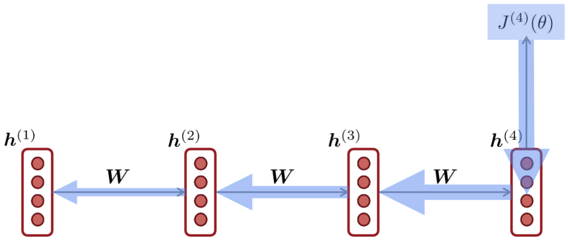
- Normalization factor 1/T is used for comparability across corpora of different lengths.

Outline

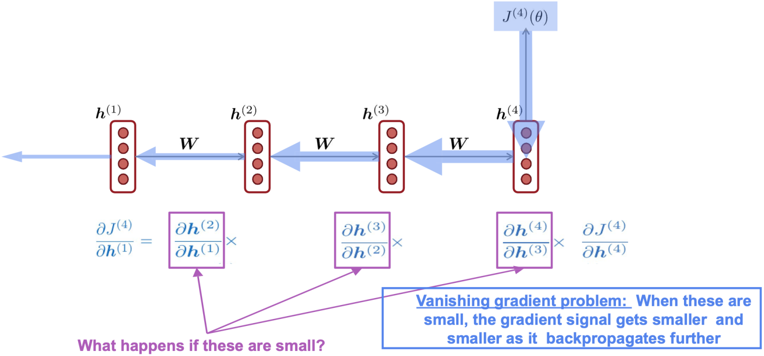
Vanishing gradient on RNN-LM



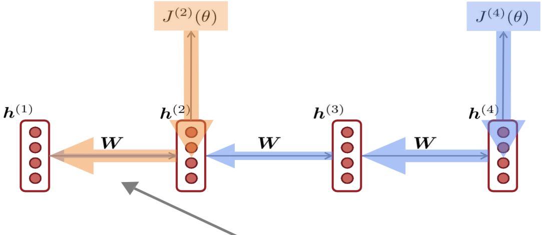
Vanishing gradient on RNN-LM (II)



Vanishing gradient on RNN-LM (III)



Vanishing gradient on RNN-LM (IV)



Gradient signal from far away is lost because it's much smaller than **gradient signal from close-by**.

So model weights are only updated only with respect to **near effects**, not **long-term effects**.

Effect of vanishing gradient on RNN-LM

- LM task: When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her __________.
- To learn from this training example, the RNN-LM needs to model the dependency between "tickets" on the 7th step and the target word "tickets" at the end
- But if gradient is small, the model can't learn this dependency
  - So, the model is unable to predict similar long-distance dependencies at test time

Why is exploding gradient a problem?

- If the gradient becomes too big, then the SGD update step becomes too big:

$$\Delta\theta = -\alpha \nabla_\theta J(\theta)$$

- This can cause bad updates: we take too large a step and reach a bad parameter configuration (with large loss)
- In the worst case, this will result in Inf or NaN in your network (then you have to restart training from an earlier checkpoint)

Gradient clipping: solution for exploding gradient

- Gradient clipping: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update
  - Intuition: take a step in the same direction, but a smaller step

$$\nabla_\theta J(\theta) \leftarrow \frac{\max\left(\|\nabla_\theta J(\theta)\|, \text{threshold}\right)}{\|\nabla_\theta J(\theta)\|} \nabla_\theta J(\theta)$$

Outline

Recurrent NN Language Models

How to fix the vanishing gradient problem?

- The main problem is that it's too difficult for the RNN to learn to preserve information over many timesteps.
- In a vanilla RNN, the hidden state is constantly being rewritten.

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t)$$

- How about a RNN with separate memory?

Recurrent NN Language Models

Long Short-Term Memory RNNs (LSTMs)

- A type of RNN proposed by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradients problem.
- On step t, there is a hidden state h_t and a cell state c_t.
  - Both are vectors length n.
  - The cell stores long-term information.
  - The LSTM can read, erase and write information from the cell.
- The selection of which information is erased/written/read is controlled by three corresponding gates.
  - The gates are also vectors length n.
  - On each timestep, each element of the gates can be open (1), closed (0), or somewhere in-between.
  - The gates are dynamic: their value is computed based on the current context.

Outline

Long Short-Term Memory RNNs (LSTMs)

We have a sequence of inputs $\{x^{(1)}, x^{(2)}, ..., x^{(T)}\}$, and we will compute a sequence of hidden states $\{h^{(1)}, h^{(2)}, ..., h^{(T)}\}$ and cell states $\{c^{(1)}, c^{(2)}, ..., c^{(T)}\}$. On timestep t:

- Forget gate: $\mathbf{f}^{(t)} = \sigma(\mathbf{W_f}[h^{(t-1)}, x^{(t)}] + \mathbf{b_f})$ controls what is kept vs forgotten, from previous cell state
- Input gate: $\mathbf{i}^{(t)} = \sigma(\mathbf{W_i}[h^{(t-1)}, x^{(t)}] + \mathbf{b_i})$ controls what parts of the new cell content are written to cell
- Output gate: $\mathbf{o}^{(t)} = \sigma(\mathbf{W_o}[h^{(t-1)}, x^{(t)}] + \mathbf{b_o})$ controls what parts of cell are output to hidden state

Long Short-Term Memory RNNs (LSTMs) (II)

- New cell content: $\tilde{\mathbf{c}}^{(t)} = \tanh(\mathbf{W_c}[h^{(t-1)}, x^{(t)}] + \mathbf{b_c})$ this is the new content to be written to the cell
- Cell state: erase ("forget") some content from last cell state, and write ("input") some new cell content:

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \tilde{\mathbf{c}}^{(t)}$$

- Hidden state: read ("output") some content from the cell:

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh(\mathbf{c}^{(t)})$$

- $\odot$: Gates are applied using element-wise product
- σ: Sigmoid goes returns values from 0 to 1

Long Short-Term Memory RNNs (LSTMs) (III)

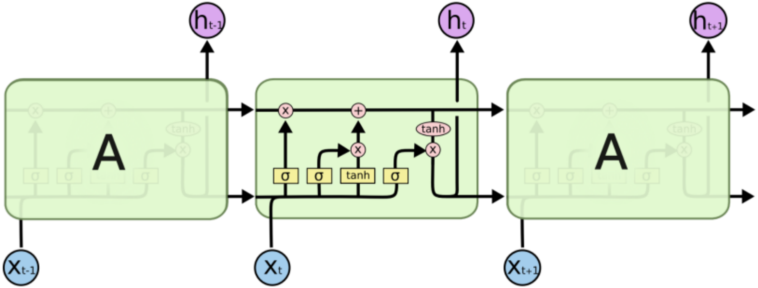


Figure: Rerpresentation of a LSTM

Outline

Vanishing Gradient in LSTMs

- The LSTM architecture makes it easier for the RNN to preserve information over many timesteps
  - For example, if the forget gate is set to 1 for a cell dimension and the input gate set to 0, then the information of that cell is preserved indefinitely.
  - In contrast, it's harder for vanilla RNN to learn a recurrent weight matrix W_h that preserves info in hidden state.
- LSTM doesn't guarantee that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies.

Vanishing Gradient in LSTMs (II)

- Is vanishing/exploding gradient just a RNN problem?
  - No! It can be a problem for all neural architectures (including feed-forward and convolutional), especially very deep ones.
  - Due to chain rule/nonlinear function, gradient can become vanishingly small as it backpropagates.
  - Thus, lower layers are learned very slowly (hard to train).
- Solution: lots of new deep feedforward/convolutional architectures that add more direct connections (allowing the gradient to flow).

$$y = f(x) + x$$

- A residual connection can be added to a layer $f(x)$ by adding the input x to the output.
- This allows the gradient to flow directly from the output to the input, avoiding vanishing/exploding gradients.

LSTMs: Real-World Success

- In 2013–2015, LSTMs started achieving state-of-the-art results in various tasks, including:

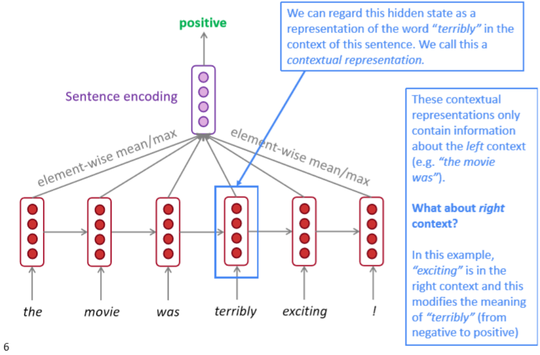
  Handwriting recognition, Speech recognition, Machine translation, Parsing, Image captioning, Language models

- LSTMs became the dominant approach for most NLP tasks.
- However, now (2023), other approaches (such as Transformers) have become dominant for many tasks, as evidenced by the decline in the use of RNNs (including LSTMs) in conferences like WMT.
  - For example, in WMT 2016, "RNN" was mentioned 44 times in the summary report, while in WMT 2019, "RNN" was mentioned only 7 times, and "Transformer" was mentioned 105 times.
  - This shift is partly due to the ability of Transformer models to parallelize better than LSTMs, making them faster and more efficient for some tasks.

Outline

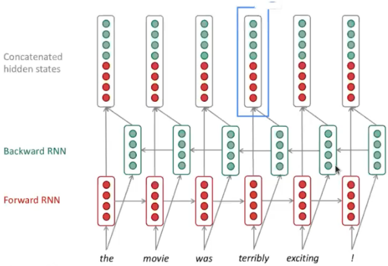
Outline

Bidirectional RNNs

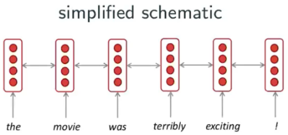


- What about the right context? In the example shown, the word "exciting" is in the right context and modifies the meaning of "terribly" from negative to positive.
- Solution: Bidirectional RNNs (Bi-RNNs) that process the input sequence in both directions simultaneously.

Bidirectional RNNs (II)



(Credit: Stanford CS224N)



(Credit: Stanford CS224N)

Bidirectional RNNs (III)

A Bidirectional RNN (Bi-RNN) is a combination of two RNNs that process the input sequence in opposite directions. The output of each RNN is a sequence of hidden states:

- **Forward RNN**: processes the input seq. from left to right

$$h_t^f = \sigma(W_{xh}^f x_t + W_{hh}^f h_{t-1}^f + b_h^f); y_t = \sigma(W_{hy}^f h_t^f + b_y)$$

- **Backward RNN**: processes the input seq. from right to left

$$h_t^b = \sigma(W_{xh}^b x_t + W_{hh}^b h_{t+1}^b + b_h^b); y_t = \sigma(W_{hy}^b h_t^b + b_y)$$

- The final output at each timestep is obtained by concatenating the hidden states from both directions:

$$h_t = [h_t^f; h_t^b]$$

The parameters of a Bi-RNN are learned through backpropagation through time (BPTT), which involves computing gradients for both directions and combining them using element-wise addition.

Bidirectional RNNs (IV)

Note: Bidirectional RNNs are only applicable if you have access to the **entire input sequence**.

- They are not applicable to Language Modeling, because in LM you only have left context available.
- LSTMs became the dominant approach for most NLP tasks.

If you do have the entire input sequence (e.g., any kind of encoding), **bidirectionality** is powerful (you should use it by default).

For example, **BERT** (Bidirectional Encoder Representations from Transformers) is a powerful pretrained contextual representation system built on bidirectionality.

Outline

Multi-layer RNNs

RNNs are already **“deep”** on one dimension (they unroll over many timesteps).

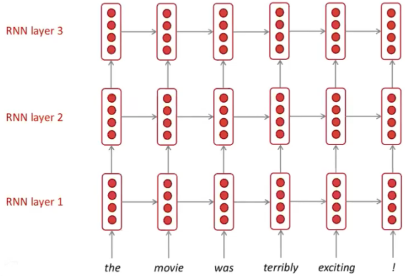
We can also make them **“deep”** in another dimension by applying multiple RNNs – this is a **multi-layer RNN**.

This allows the network to compute more **complex representations**.

- The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.

Multi-layer RNNs are also called **stacked RNNs**.

Multi-layer RNNs (II)



(Credit: Stanford CS231N)

Figure: Representation of a multi-layer RNN

Multi-layer RNNs (III)

- High-performing RNNs are often **multi-layer** (but aren't as deep as convolutional or feed-forward networks)
- **For example:** In a 2017 paper, Britz et al find that for Neural Machine Translation, 2 to 4 layers is best for the encoder RNN, and 4 layers is best for the decoder RNN
- **Transformer-based networks** (e.g., BERT) are usually deeper, like **12 or 24 layers**.
- A multi-layer RNN is a network that applies multiple RNNs, allowing the network to compute more complex representations. The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.

Outline

ELMo: Embeddings from Language Models

- ELMo is a type of deep contextualized word representation that models both:
  1. complex characteristics of word use (e.g., syntax and semantics)
  2. how these uses vary across linguistic contexts (i.e., to model polysemy)
- ELMo embeddings are learned functions of the internal states of a deep bidirectional language model (biLM), which is pre-trained on a large text corpus
- A biLM combines both a forward and backward LM and jointly maximizes the log likelihood of both directions

ELMo

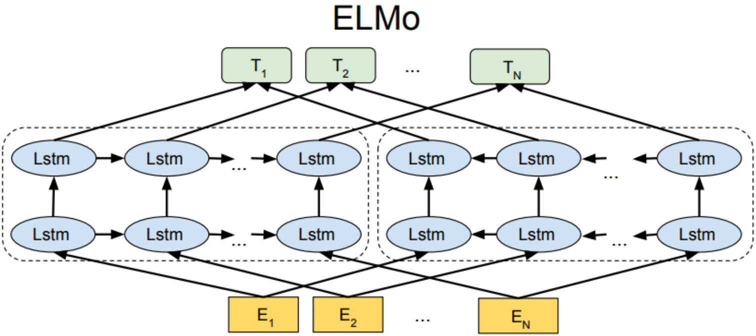
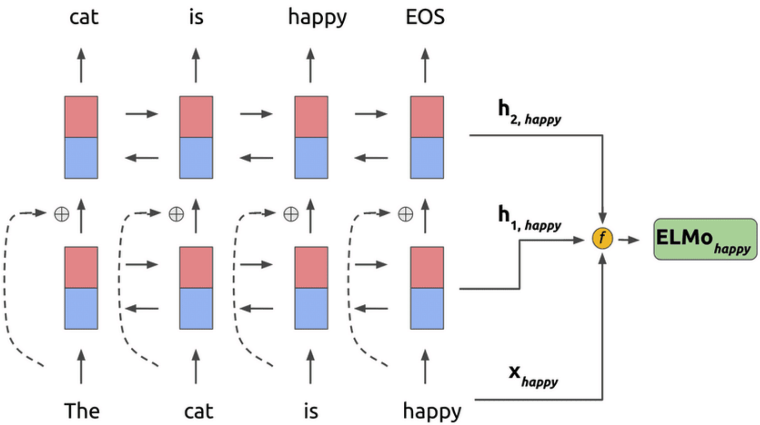


Figure: Representation of ELMo

ELMo: How to use it?

- To add ELMo to an existing NLP system, we freeze the weights of the biLM and then concatenate the ELMo vector ELMoktask with x_k and pass the ELMo enhanced representation [x_k;ELMo$_{k,task}$] into the task RNN
- Here x_k is a context-independent token representation for each token position
- The ELMo vector ELMo$_{k,task}$ is computed as a weighted average of the internal states of the biLM
- The weights are learned for each task as scalar parameters

ELMo for Contextual Word Embeddings



ELMo: Why is it good?

- ELMo representations are:
  1. Contextual: The representation for each word depends on the entire context in which it is used
  2. Deep: The word representations combine all layers of a deep pre-trained neural network
  3. Character based: ELMo representations are purely character based, allowing the network to use morphological clues to form robust representations for out-of-vocabulary tokens unseen in training
- ELMo significantly improves the state-of-the-art for a broad range of challenging NLP problems, including question answering, textual entailment, sentiment analysis, named entity recognition, etc.

ELMo: Architecture and Training Objectives

- ELMo uses a two-layer bi-directional LSTM network as its architecture
- Each layer has 4096 units and 512-dimensional projections
- The input to the network is a sequence of characters, which are embedded into a 16-dimensional vector
- A convolutional layer with 2048 filters of width 1 to 7 applied to the input character embeddings. The max-pooled output is then re-projected to a 512-dimensional vector.
- The network is pre-trained on a large corpus with the following training objective:
    - Language modeling: predict the next word given the previous words (forward LM) and predict the previous word given the next words (backward LM)