Mining Unstructured Data

10. Recurrent NN Language Models

Javier Ferrando

Based on Stanford CS224N
Recap
Binary Logistic Regression

- Classify each data point $x$ as one of two classes $(c_0, c_1)$
- We model the conditional probability of a class given the input data point $x$: $p(c_1|x)$
- We use the *sigmoid* function to do so:
  $$p(c_1|x; \theta) = \sigma(z) = \frac{1}{1 + e^{-z}}$$
- We get a linear decision boundary
Multinomial Logistic Regression

- Multinomial Logistic Regression lets us classify between more than 2 classes (multi-class classification)

\[
p(c_k|x; \theta) = \frac{e^{z_k}}{\sum_j e^{z_j}}
\]

\[z = Wx\]
Neural Network = several logistic regression at the same time

\[ p(c_k \mid x; \theta) = \frac{e^{z_k}}{\sum_j e^{z_j}} \]
Neural Network = several logistic regression at the same time

Multi-layer perceptron (MLP)

\[ p(c_k|\mathbf{x}; \theta) = \frac{e^{z_k}}{\sum_j e^{z_j}} \]
Index

- Language Modeling
  - N-gram Language Models
  - Neural Language Models

- RNN Language Models
  - Simple RNN
  - RNN training
  - Generating text with RNNs
  - Problems with RNNs

- Fancy RNNs
  - LSTM
  - Bidirectional RNNs
  - Multi-layer RNNs
Language Modeling
Language Modeling

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

- More formally: given a sequence of words $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$:

$$P(x^{(t+1)} | x^{(t)}, \ldots, x^{(1)})$$

where $x^{(t+1)}$ can be any word in the vocabulary.

- A system that does this is called a **Language Model**.

The students opened their _______ 

books  

laptops  

exams
You use Language Models every day
You use Language Models every day

Google

what is the |
what is the weather
what is the meaning of life
what is the dark web
what is the xfl
what is the doomsday clock
what is the weather today
what is the keto diet
what is the american dream
what is the speed of light
what is the bill of rights
N-gram Language Models

- How to learn a Language Model
  - Pre-deep learning: n-gram Language Model
  - Deep learning: Recurrent Neural Network, Transformer...

- A n-gram is a chunk of n-consecutive words:
  - Unigram: “the”, “students”, “opened”, “their”
  - Bigram: “the students”, “students opened”
  - Trigram: “the students opened”, “students opened their”
  - 4-gram: “the students opened their”

- Idea: collect statistics about how frequent different n-grams are and use these to predict next word
N-gram Language Models

- Markov assumption: $x^{(t+1)}$ depends only on the preceding n-1 words

\[
P(x^{(t+1)}|x^{(t)}, \ldots, x^{(1)}) = P(x^{(t+1)}|x^{(t)}, \ldots, x^{(t-n+2)})
\]

(proposition)

- We can get n-gram and (n-1)-gram probabilities by counting them in a large corpus:

\[
\approx \frac{\text{count}(x^{(t+1)}, x^{(t)}, \ldots, x^{(t-n+2)})}{\text{count}(x^{(t)}, \ldots, x^{(t-n+2)})}
\]
N-gram Language Models

Suppose we are learning a 4-gram Language Model.

\[
P(w | \text{students opened their}) = \frac{\text{count}(\text{students opened their } w)}{\text{count}(\text{students opened their})}
\]

For example, suppose that in the corpus:

- “students opened their” occurred 1000 times
- “students opened their books” occurred 400 times
  \[\rightarrow P(\text{books} | \text{students opened their}) = 0.4\]
- “students opened their exams” occurred 100 times
  \[\rightarrow P(\text{exams} | \text{students opened their}) = 0.1\]
Sparsity problems 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 $\delta$ to the count for every $w \in V$. This is called smoothing.

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

**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.
Storage problems with n-gram Language Models

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

$$P(w|\text{students opened their}) = \frac{\text{count(students opened their } w)}{\text{count(students opened their)}}$$
Generating text with n-gram Language Models

You can also use Language Models to generate text
Generating text with n-gram Language Models

You can also use Language Models to generate text
Generating text with n-gram Language Models

You can also use Language Models to generate text

\[ \text{today the price of } \underline{\text{gold}} \text{ on this} \]

get probability distribution

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>0.072</td>
</tr>
<tr>
<td>18</td>
<td>0.043</td>
</tr>
<tr>
<td>oil</td>
<td>0.043</td>
</tr>
<tr>
<td>its</td>
<td>0.036</td>
</tr>
<tr>
<td>gold</td>
<td>0.018</td>
</tr>
</tbody>
</table>
Generating text with n-gram Language Models

You can also use Language Models to generate text

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

Grammatically correct!

But incoherent. We need to consider more than three words at a time if we want to model language well. But increasing \( n \) worsens sparsity problem, and increases model size.
Neural Language Model

- Recall the Language Modeling task:
  - Input: sequence of words: $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$
  - Output: probability distribution of the next words: $P(x^{(t+1)} | x^{(t)}, \ldots, x^{(1)})$

- How about a window-based neural model?
  - We saw this applied to Named Entity Recognition:
A fixed-window neural Language Model


output distribution
\[ \hat{y} = \text{softmax}(U h + b_2) \in \mathbb{R}^{V} \]

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

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

words / one-hot vectors
\[ \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)} \]
N-gram Language Models

- 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 $W$.

We need a neural architecture that can process any length input.
RNN Language Models
Recurrent Neural Networks (RNN)

Core idea: Apply the same weights repeatedly $W$.
A RNN Language Model

output distribution
\[ \hat{y}^{(t)} = \text{softmax}\left(U h^{(t)} + b_2\right) \in \mathbb{R}^{V} \]

hidden states
\[ h^{(t)} = \sigma\left(W_h h^{(t-1)} + W_e e^{(t)} + b_1\right) \]

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

word embeddings
\[ e^{(t)} = E x^{(t)} \]

words / one-hot vectors
\[ x^{(t)} \in \mathbb{R}^{V} \]

Note: this input sequence could be much longer now!
A RNN Language Model

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

\[
\hat{y}^{(4)} = P(x^{(5)} | \text{the students opened their books})
\]
A RNN Language Model

RNN disadvantages:
- Recurrent computation is slow
- In practice, difficult to access information from many steps back
Training an RNN Language Model

- Get a **big corpus of text** which is a sequence of words \( x^{(1)}, x^{(2)}, \ldots, x^{(t)} \)
- 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 **cross-entropy** between predicted probability distribution \( \hat{y}^{(t)} \), and the true next word \( y^{(t)} \)(one-hot for \( x^{(t+1)} \)):

\[
J^{(t)}(\theta) = CE(y^{(t)}, \hat{y}^{(t)}) = -\sum_{w \in V} y_w^{(t)} \log \hat{y}_w^{(t)} = -\log \hat{y}_{x_{t+1}}^{(t)}
\]

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

\[
J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{y}_{x_{t+1}}^{(t)}
\]
Training a RNN Language Model

= negative log prob of “students”

Loss $\rightarrow J^{(1)}(\theta)$ $\rightarrow J^{(2)}(\theta)$ $\rightarrow J^{(3)}(\theta)$ $\rightarrow J^{(4)}(\theta)$

Predicted prob dists $\rightarrow y^{(1)} \rightarrow U \rightarrow h^{(1)}$ $\rightarrow y^{(2)} \rightarrow U \rightarrow h^{(2)}$ $\rightarrow y^{(3)} \rightarrow U \rightarrow h^{(3)}$ $\rightarrow y^{(4)} \rightarrow U \rightarrow h^{(4)}$

Corpus $\rightarrow$ the $\rightarrow$ students $\rightarrow$ opened $\rightarrow$ their $\rightarrow$ exams $\rightarrow$ ...
Training a RNN Language Model

Loss → $J^{(1)}(\theta)$ → $y^{(1)}$ → $U$ → $h^{(1)}$ → $W_h$ → $W_e$ → $e^{(1)}$...

Predicted prob dists

= negative log prob of “opened”

Corpus → the → students → opened → their → exams → ...

$J^{(2)}(\theta)$ → $y^{(2)}$ → $U$ → $h^{(2)}$ → $W_h$ → $W_e$ → $e^{(2)}$...

$J^{(3)}(\theta)$ → $y^{(3)}$ → $U$ → $h^{(3)}$ → $W_h$ → $W_e$ → $e^{(3)}$...

$J^{(4)}(\theta)$ → $y^{(4)}$ → $U$ → $h^{(4)}$ → $W_h$ → $W_e$ → $e^{(4)}$...

...
Training a RNN Language Model

![Diagram of a RNN Language Model]

- **Corpus**: the sequence of words from the dataset.
- **Predicted prob dists**: The probability distribution of the next word given the previous context.
- **Loss**: The negative log probability of the correct word, calculated as the loss function for the model.

The model iteratively updates its hidden state ($h^{(t)}$) using the current input ($x^{(t)}$) and the previous hidden state ($h^{(t-1)}$) through weight matrices ($W_h$, $W_e$). The output probability distribution ($y^{(t)}$) is then used to calculate the loss for the current step. This process is repeated for each word in the corpus, updating the model's parameters to minimize the overall loss.
Training a RNN Language Model

Corpus → the students opened their exams → Loss

Predicted prob dists

Loss → $J^{(1)}(\theta)$ → $y^{(1)}$ → $U$ → $h^{(1)}$ → $W_h$ → $e^{(1)}$ → $E$

$J^{(2)}(\theta)$ → $y^{(2)}$ → $U$ → $h^{(2)}$ → $W_h$ → $e^{(2)}$ → $E$

$J^{(3)}(\theta)$ → $y^{(3)}$ → $U$ → $h^{(3)}$ → $W_h$ → $e^{(3)}$ → $E$

$J^{(4)}(\theta)$ → $y^{(4)}$ → $U$ → $h^{(4)}$ → $W_h$ → $e^{(4)}$ → $E$

= negative log prob of “exams”
Training a RNN Language Model

Loss $\rightarrow J^{(1)}(\theta) + J^{(2)}(\theta) + J^{(3)}(\theta) + J^{(4)}(\theta) + \ldots \rightarrow \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta)$

Teacher forcing

Corpus $\rightarrow$ the $^{(1)}$ students $^{(2)}$ opened $^{(3)}$ their $^{(4)}$ exams $^{(5)}$...
Training an RNN Language Model

- However: Computing loss and gradients across entire corpus \( x^{(1)}, \ldots, x^{(T)} \) is too expensive!

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

- In practice, consider \( x^{(1)}, \ldots, x^{(T)} \) 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 \( J(\theta) \) for a sentence (actually, a batch of sentences), compute gradients and update weights. Repeat.
Training an RNN Language Model

How do we calculate this?

Answer: Backpropagate over timesteps $i=t,...,0$, summing gradients as you go. This algorithm is called “backpropagation through time”
Generating text

Recurrent NN Language Models
Generating text

- You can train an RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on Obama speeches:

   *The United States will step up to the cost of a new challenges of the American people that will share the fact that we created the problem. They were attacked and so that they have to say that all the task of the final days of war that I will not be able to get this done.*
Generating text

- You can train an RNN-LM on any kind of text, then generate text in that style.
- RNN-LM trained on recipes:

Place each pasta over layers of lumps. Shape mixture into the moderate oven and simmer until firm. Serve hot in bodied fresh, mustard, orange and cheese.

Combine the cheese and salt together the dough in a large skillet; add the ingredients and stir in the chocolate and pepper.
Evaluating Language Models

- The standard evaluation metric for Language Models is perplexity.

$$\text{perplexity} = \prod_{t=1}^{T} \left( \frac{1}{P_{LM}(x^{(t+1)} \mid x^{(t)}, \ldots, x^{(1)})} \right)^{1/T}$$

- This is equal to the exponential of the cross-entropy loss:

$$= \prod_{t=1}^{T} \left( \frac{1}{\hat{y}_{x_{t+1}}^{(t)}} \right)^{1/T} = \exp \left( \frac{1}{T} \sum_{t=1}^{T} -\log \hat{y}_{x_{t+1}}^{(t)} \right) = \exp(J(\theta))$$

Lower perplexity is better
Problems with Vanishing and Exploding Gradients
Problems with Vanishing and Exploding Gradients
Problems with Vanishing and Exploding Gradients

- **Vanishing gradient problem:** When these are small, the gradient signal gets smaller and smaller as it backpropagates further.

- **Exploding gradient problem:** When these are large, the gradient signal grows exponentially as it backpropagates further.
Problems with Vanishing and Exploding Gradients

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:

\[ \theta_{new} = \theta_{old} - \alpha \nabla_{\theta} J(\theta) \]

- 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

```
Algorithm 1 Pseudo-code for norm clipping

\[ \hat{g} \leftarrow \frac{\partial \mathcal{L}}{\partial \theta} \]

if $\|\hat{g}\| \geq$ threshold then

\[ \hat{g} \leftarrow \frac{\text{threshold}}{\|\hat{g}\|} \hat{g} \]

end if
```

- **Intuition**: take a step in the same direction, but a smaller step
Fancy RNNs
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

  \[ h^{(t)} = \sigma \left( W_h h^{(t-1)} + W_x a^{(t)} + b \right) \]

- How about a RNN with separate memory?
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
Long Short-Term Memory RNNs (LSTMs)

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

**Forget gate**: controls what is kept vs forgotten, from previous cell state

**Input gate**: controls what parts of the new cell content are written to cell

**Output gate**: controls what parts of cell are output to hidden state

**New cell content**: 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

**Hidden state**: read (“output”) some content from the cell

### Sigmoid function: all gate values are between 0 and 1

$$f(t) = \sigma \left( W_f h^{(t-1)} + U_f x^{(t)} + b_f \right)$$

$$i(t) = \sigma \left( W_i h^{(t-1)} + U_i x^{(t)} + b_i \right)$$

$$o(t) = \sigma \left( W_o h^{(t-1)} + U_o x^{(t)} + b_o \right)$$

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

$$\tilde{c}(t) = \tanh \left( W_c h^{(t-1)} + U_c x^{(t)} + b_c \right)$$

$$h(t) = o(t) \circ \tanh c(t)$$

All these are vectors of same length $n$.

Gates are applied using element-wise product.
Long Short-Term Memory RNNs (LSTMs)

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/
Long Short-Term Memory RNNs (LSTMs)

1. **Forget some cell content**
2. **Compute the forget gate**
3. **Compute the input gate**
4. **Compute the new cell content**
5. **Compute the output gate**
6. **Write some new cell content**
7. **Output some cell content to the hidden state**
Long Short-Term Memory RNNs (LSTMs)

- The LSTM architecture makes it easier for the RNN to preserve information over many timesteps
  - e.g. 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
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 / choice of nonlinearity 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 (thus allowing the gradient to flow)
LSTMs: real-world success

- In 2013–2015, LSTMs started achieving state-of-the-art results
  - Successful tasks include handwriting recognition, speech recognition, machine translation, parsing, and image captioning, as well as language models
  - LSTMs became the dominant approach for most NLP tasks

- Now (2022), other approaches (e.g., Transformers) have become dominant for many tasks:
  - For example, in WMT (a Machine Translation conference + competition):
    - In WMT 2016, the summary report contains “RNN” 44 times
    - In WMT 2019: “RNN” 7 times, ”Transformer” 105 times
Bidirectional RNNs: motivation

Task: Sentiment Classification

We can regard this hidden state as a representation of the word “terribly” in the context of this sentence. We call this a contextual representation.

Sentence encoding

These contextual representations only contain information about the left context (e.g. “the movie was”).

What about right context?
In this example, “exciting” is in the right context and this modifies the meaning of “terribly” (from negative to positive).
Bidirectional RNNs

This contextual representation of “terribly” has both left and right context!
Bidirectional RNNs

On timestep $t$:

\[
\begin{align*}
\mathbf{h}^{(t)} &= \text{RNN}_{FW}(\mathbf{h}^{(t-1)}; \mathbf{x}^{(t)}) \\
\overleftarrow{\mathbf{h}}^{(t)} &= \text{RNN}_{BW}(\overleftarrow{\mathbf{h}}^{(t+1)}; \mathbf{x}^{(t)})
\end{align*}
\]

Concatenated hidden states

We regard this as “the hidden state” of a bidirectional RNN. This is what we pass on to the next parts of the network.

This is a general notation to mean “compute one forward step of the RNN” – it could be a vanilla, LSTM or GRU computation.

Generally, these two RNNs have separate weights.
Bidirectional RNNs

The two-way arrows indicate bidirectionality and the depicted hidden states are assumed to be the concatenated forwards+backwards states.
Bidirectional RNNs

- 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 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.
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

The hidden states from RNN layer $i$ are the inputs to RNN layer $i+1$.
Multi-layer RNNs

- 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.