Lecture 6:
Language Models and
Recurrent Neural Networks

Abigail See
Overview

Today we will:

• Introduce a new NLP task
  • Language Modeling

• Introduce a new family of neural networks
  • Recurrent Neural Networks (RNNs)

These are two of the most important ideas for the rest of the class!
Language Modeling

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

  \[ \text{the students opened their } \underline{	ext{_____}} \]

- 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 \( V = \{w_1, \ldots, w_{|V|}\} \)

- A system that does this is called a **Language Model**.
You can also think of a Language Model as a system that assigns probability to a piece of text.

For example, if we have some text $x^{(1)}, \ldots, x^{(T)}$, then the probability of this text (according to the Language Model) is:

$$P(x^{(1)}, \ldots, x^{(T)}) = P(x^{(1)}) \times P(x^{(2)} \mid x^{(1)}) \times \cdots \times P(x^{(T)} \mid x^{(T-1)}, \ldots, x^{(1)})$$

$$= \prod_{t=1}^{T} P(x^{(t)} \mid x^{(t-1)}, \ldots, x^{(1)})$$

This is what our LM provides
You use Language Models every day!

I'll meet you at the cafe.
You use Language Models every day!

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

Google Search  I'm Feeling Lucky
n-gram Language Models

the students opened their ______

• **Question**: How to learn a Language Model?
• **Answer** (pre- Deep Learning): learn a *n*-gram Language Model!

• **Definition**: A *n*-gram is a chunk of *n* consecutive words.
  • unigrams: “the”, “students”, “opened”, ”their”
  • bigrams: “the students”, “students opened”, “opened their”
  • trigrams: “the students opened”, “students opened their”
  • 4-grams: “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

- First we make a **simplifying assumption**: \( x^{(t+1)} \) depends only on the preceding \( n-1 \) words.

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

**Question**: How do we get these \( n \)-gram and \( (n-1) \)-gram probabilities?

**Answer**: By **counting** them in some large corpus of text!

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

Suppose we are learning a 4-gram Language Model.

as the proctor started the clock, the students opened their _____
discard

For example, suppose that in the corpus:

- “students opened their” occurred 1000 times
- “students opened their books” occurred 400 times
  - $\rightarrow P(\text{books} \mid \text{students opened their}) = 0.4$
- “students opened their exams” occurred 100 times
  - $\rightarrow P(\text{exams} \mid \text{students opened their}) = 0.1$

Should we have discarded the “proctor” context?
Sparsity Problems with n-gram Language Models

Sparsity Problem 1

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

**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 $w$!

**Solution:** Just condition on “opened their” instead. This is called backoff.

Note: Increasing $n$ makes sparsity problems worse. Typically we can’t have $n$ bigger than 5.
Storage Problems with \textit{n}-gram Language Models

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

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

Increasing \textit{n} or increasing corpus increases model size!
n-gram Language Models in practice

• You can build a simple trigram Language Model over a 1.7 million word corpus (Reuters) in a few seconds on your laptop*

Today the ________

get probability distribution

<table>
<thead>
<tr>
<th></th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>company</td>
<td>0.153</td>
</tr>
<tr>
<td>bank</td>
<td>0.153</td>
</tr>
<tr>
<td>price</td>
<td>0.077</td>
</tr>
<tr>
<td>italian</td>
<td>0.039</td>
</tr>
<tr>
<td>emirate</td>
<td>0.039</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Sparsity problem: not much granularity in the probability distribution

Otherwise, seems reasonable!

* Try for yourself: [https://nlpforhackers.io/language-models/](https://nlpforhackers.io/language-models/)
Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.

\[\text{today the } \underline{\text{______}}\]

condition on this

get probability distribution

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>company</td>
<td>0.153</td>
</tr>
<tr>
<td>bank</td>
<td>0.153</td>
</tr>
<tr>
<td>price</td>
<td>0.077</td>
</tr>
<tr>
<td>italian</td>
<td>0.039</td>
</tr>
<tr>
<td>emirate</td>
<td>0.039</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

sample
Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.

```
today the price_______
condition on this
```

get probability distribution

<table>
<thead>
<tr>
<th>Sample</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>of</td>
<td>0.308</td>
</tr>
<tr>
<td>for</td>
<td>0.050</td>
</tr>
<tr>
<td>it</td>
<td>0.046</td>
</tr>
<tr>
<td>to</td>
<td>0.046</td>
</tr>
<tr>
<td>is</td>
<td>0.031</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.

\[ \text{today the price of } \underline{\text{________}} \]

(condition on this) get probability distribution

<table>
<thead>
<tr>
<th>Word</th>
<th>Probability</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>

... sample
Generating text with a n-gram Language Model

- You can also use a Language Model to generate text.

\[ \text{today the price of gold ______} \]
Generating text with a n-gram Language Model

• You can also use a Language Model 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.

Surprisingly grammatical!

...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...
How to build a *neural* Language Model?

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

- How about a window-based neural model?
  - We saw this applied to Named Entity Recognition in Lecture 3:
A fixed-window neural Language Model

as the proctor started the clock
discard

the students opened their ______

fixed window
A fixed-window neural Language Model

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

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

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

words / one-hot vectors
\[ x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)} \]
A fixed-window neural Language Model

**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 $W$
- Window can never be large enough!
- $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in $W$. No symmetry in how the inputs are processed.

We need a neural architecture that can process any length input.
Recurrent Neural Networks (RNN)
A family of neural architectures

Core idea: Apply the same weights \( W \) repeatedly

outputs (optional) \( \{ \)

hidden states \( \{ \)

input sequence (any length) \( \{ \)

\[ \begin{align*}
\hat{y}^{(1)} & \quad h^{(1)} \\
\hat{y}^{(2)} & \quad h^{(2)} \\
\hat{y}^{(3)} & \quad h^{(3)} \\
\hat{y}^{(4)} & \quad h^{(4)} \\
\end{align*} \]

\[ \begin{align*}
x^{(1)} & \quad W \\
x^{(2)} & \quad W \\
x^{(3)} & \quad W \\
x^{(4)} & \quad W \\
\end{align*} \]

\[ \begin{align*}
\cdots
\end{align*} \]
A RNN Language Model

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

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

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, but this slide doesn’t have space!
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.

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

More on these later in the course
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 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

\[ J^{(1)}(\theta) \]

= negative log prob
of “students”

Loss →

\[ \hat{y}^{(1)} \]

\[ J^{(2)}(\theta) \]

\[ \hat{y}^{(2)} \]

\[ J^{(3)}(\theta) \]

\[ \hat{y}^{(4)} \]

Predicted prob dists

\[ h^{(0)} \]

\[ W_h \]

\[ h^{(1)} \]

\[ W_h \]

\[ U \]

\[ h^{(2)} \]

\[ W_h \]

\[ U \]

\[ h^{(3)} \]

\[ W_h \]

\[ U \]

\[ h^{(4)} \]

\[ W_h \]

\[ U \]

\[ \ldots \]

Corpus →

\[ x^{(1)} \]

\[ the \]

\[ E \]

\[ x^{(2)} \]

\[ students \]

\[ E \]

\[ x^{(3)} \]

\[ opened \]

\[ E \]

\[ x^{(4)} \]

\[ their \]

\[ E \]

\[ exams \]

\[ \ldots \]
Training a RNN Language Model

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

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

Corpus $\rightarrow x^{(1)} \rightarrow x^{(2)} \rightarrow x^{(3)} \rightarrow x^{(4)} \rightarrow \text{exams}$

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

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

Predicted prob dists $\rightarrow \hat{y}^{(1)} \rightarrow \hat{y}^{(2)} \rightarrow \hat{y}^{(3)} \rightarrow \hat{y}^{(4)}$ $\rightarrow U \rightarrow h^{(1)} \rightarrow W_h \rightarrow e^{(1)} \uparrow E \rightarrow x^{(1)}$ $\rightarrow \text{the}$ $\rightarrow \text{students}$ $\rightarrow \text{opened}$ $\rightarrow \text{their}$ $\rightarrow \text{exams}$ $\rightarrow \text{...}$

Factors:

- Loss
- Predicted prob dists
- Corpus

Equation:

$= \text{negative log prob of “their”}$
Training a RNN Language Model

- Loss: $J^{(1)}(\theta) \rightarrow J^{(2)}(\theta) \rightarrow J^{(3)}(\theta) \rightarrow J^{(4)}(\theta)$
- Predicted prob dists: $\hat{y}^{(1)} \rightarrow \hat{y}^{(2)} \rightarrow \hat{y}^{(3)} \rightarrow \hat{y}^{(4)}$
- Corpus: $x^{(1)}$ (the) $x^{(2)}$ (students) $x^{(3)}$ (opened) $x^{(4)}$ (their) exams

$= \text{negative log prob of “exams”}$

$h^{(0)} \rightarrow W_h \rightarrow h^{(1)} \rightarrow W_h \rightarrow h^{(2)} \rightarrow W_h \rightarrow h^{(3)} \rightarrow W_h \rightarrow h^{(4)} \rightarrow W_h \rightarrow \ldots$

$W_e \rightarrow e^{(1)} \rightarrow E \rightarrow x^{(1)}$ $W_e \rightarrow e^{(2)} \rightarrow E \rightarrow x^{(2)}$ $W_e \rightarrow e^{(3)} \rightarrow E \rightarrow x^{(3)}$ $W_e \rightarrow e^{(4)} \rightarrow E \rightarrow x^{(4)}$
Training a RNN Language Model

\[
J(θ) + J(2)(θ) + J(3)(θ) + J(4)(θ) + \ldots = \frac{1}{T} \sum_{t=1}^{T} J(t)(θ)
\]

Corpus \( \{x(1), x(2), x(3), x(4), \ldots\} \)

Predicted prob dists

Loss

\( h(0) \rightarrow W_h \rightarrow h(1) \rightarrow U \rightarrow \hat{y}(1) \)

\( h(1) \rightarrow W_h \rightarrow h(2) \rightarrow U \rightarrow \hat{y}(2) \)

\( h(2) \rightarrow W_h \rightarrow h(3) \rightarrow U \rightarrow \hat{y}(3) \)

\( h(3) \rightarrow W_h \rightarrow h(4) \rightarrow U \rightarrow \hat{y}(4) \)

\( \hat{y}(1) \rightarrow W_e \rightarrow e(1) \rightarrow E \rightarrow \text{the} \)

\( \hat{y}(2) \rightarrow W_e \rightarrow e(2) \rightarrow E \rightarrow \text{students} \)

\( \hat{y}(3) \rightarrow W_e \rightarrow e(3) \rightarrow E \rightarrow \text{opened} \)

\( \hat{y}(4) \rightarrow W_e \rightarrow e(4) \rightarrow E \rightarrow \text{their} \)

\( \text{exams} \ldots \)
Training a 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.
Question: What’s the derivative of $J^{(t)}(\theta)$ w.r.t. the repeated weight matrix $W_h$?

Answer: 
$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial W_h} \right|_{(i)}$$

“The gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears”

Why?
Multivariable Chain Rule

- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

Derivative of composition function

Source:
Backpropagation for RNNs: Proof sketch

- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}$$

Derivative of composition function

In our example:

Apply the multivariable chain rule:

$$\frac{\partial J^{(t)}}{\partial W_h} = \left( \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h} \right) \frac{\partial W_h}{\partial W_h} = 1$$

Source:
Backpropagation for RNNs

Answer: Backpropagate over timesteps \( i=t,\ldots,0 \), summing gradients as you go. This algorithm is called “backpropagation through time”

**Question:** How do we calculate this?

\[
\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \left. \frac{\partial J^{(t)}}{\partial W_h} \right|_{(i)}
\]
Generating text with a RNN Language Model

Just like a n-gram Language Model, you can use a RNN Language Model to generate text by repeated sampling. Sampled output is next step’s input.
Generating text with a RNN Language Model

- Let’s have some fun!
- You can train a 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.

Source: https://medium.com/@samim/obama-rnn-machine-generated-political-speeches-c8abd18a2ea0
Generating text with a RNN Language Model

• Let’s have some fun!
• You can train a RNN-LM on any kind of text, then generate text in that style.
• RNN-LM trained on *Harry Potter*:

“Sorry,” Harry shouted, panicking—“I’ll leave those brooms in London, are they?”

“No idea,” said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry’s shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn’t felt it seemed. He reached the teams too.

Generating text with a RNN Language Model

• Let’s have some fun!
• You can train a RNN-LM on any kind of text, then generate text in that style.
• RNN-LM trained on recipes:

  Title: CHOCOLATE RANCH BARBECUE
  Categories: Game, Casseroles, Cookies, Cookies
  Yield: 6 Servings

  2 tb Parmesan cheese — chopped
  1 c Coconut milk
  3 Eggs, beaten

  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.

  Source: https://gist.github.com/nylki/1efbaa36635956d35bcc
Generating text with a RNN Language Model

• Let’s have some fun!
• You can train a RNN-LM on any kind of text, then generate text in that style.
• RNN-LM trained on **paint color names**:

![Paint Color Names](source)

This is an example of a **character-level RNN-LM** (predicts what character comes next)

Evaluating Language Models

• The standard evaluation metric for Language Models is perplexity.

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

Inverse probability of corpus, according to Language Model

Normalized by number of words

• This is equal to the exponential of the cross-entropy loss \( J(\theta) \):

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

Lower perplexity is better!
RNNs have greatly improved perplexity

<table>
<thead>
<tr>
<th>Model</th>
<th>Perplexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolated Kneser-Ney 5-gram (Chelba et al., 2013)</td>
<td>67.6</td>
</tr>
<tr>
<td>RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013)</td>
<td>51.3</td>
</tr>
<tr>
<td>RNN-2048 + BlackOut sampling (Ji et al., 2015)</td>
<td>68.3</td>
</tr>
<tr>
<td>Sparse Non-negative Matrix factorization (Shazeer et al., 2015)</td>
<td>52.9</td>
</tr>
<tr>
<td>LSTM-2048 (Jozefowicz et al., 2016)</td>
<td>43.7</td>
</tr>
<tr>
<td>2-layer LSTM-8192 (Jozefowicz et al., 2016)</td>
<td>30</td>
</tr>
<tr>
<td><strong>Ours small</strong> (LSTM-2048)</td>
<td>43.9</td>
</tr>
<tr>
<td><strong>Ours large</strong> (2-layer LSTM-2048)</td>
<td>39.8</td>
</tr>
</tbody>
</table>

Perplexity improves (lower is better)

Source: [https://research.fb.com/building-an-efficient-neural-language-model-over-a-billion-words/](https://research.fb.com/building-an-efficient-neural-language-model-over-a-billion-words/)
Why should we care about Language Modeling?

• Language Modeling is a \textbf{benchmark task} that helps us \textit{measure our progress} on understanding language

• Language Modeling is a \textbf{subcomponent} of many NLP tasks, especially those involving \textit{generating text} or \textit{estimating the probability of text}:
  
  • Predictive typing
  • Speech recognition
  • Handwriting recognition
  • Spelling/grammar correction
  • Authorship identification
  • Machine translation
  • Summarization
  • Dialogue
  • etc.
Recap

- **Language Model**: A system that predicts the next word

- **Recurrent Neural Network**: A family of neural networks that:
  - Take sequential input of any length
  - Apply the same weights on each step
  - Can optionally produce output on each step

- Recurrent Neural Network ≠ Language Model

- We’ve shown that RNNs are a great way to build a LM.

- But RNNs are useful for much more!
RNNs can be used for tagging
e.g. part-of-speech tagging, named entity recognition
RNNs can be used for sentence classification
e.g. sentiment classification

How to compute sentence encoding?
RNNs can be used for sentence classification

e.g. sentiment classification

How to compute sentence encoding?

Basic way:
Use final hidden state

Sentence encoding
RNNs can be used for sentence classification

e.g. sentiment classification

How to compute sentence encoding?

Usually better:
Take element-wise max or mean of all hidden states
RNNs can be used as an encoder module

e.g. question answering, machine translation, many other tasks!

Here the RNN acts as an encoder for the Question (the hidden states represent the Question). The encoder is part of a larger neural system.

**Question: what nationality was Beethoven?**

**Context:** Ludwig van Beethoven was a German composer and pianist. A crucial figure ...

**Answer:** German
RNN-LMs can be used to generate text
e.g. speech recognition, machine translation, summarization

This is an example of a conditional language model.
We’ll see Machine Translation in much more detail later.
A note on terminology

RNN described in this lecture = “vanilla RNN”

Next lecture: You will learn about other RNN flavors like GRU and LSTM and multi-layer RNNs

By the end of the course: You will understand phrases like “stacked bidirectional LSTM with residual connections and self-attention”
Next time

- **Problems** with RNNs!
  - Vanishing gradients

  
  **motivates**

- **Fancy RNN** variants!
  - LSTM
  - GRU
  - multi-layer
  - bidirectional
Lecture 7:
Vanishing Gradients and Fancy RNNs

Abigail See
Overview

• Last lecture we learned:
  • Recurrent Neural Networks (RNNs) and why they’re great for Language Modeling (LM).

• Today we’ll learn:
  • Problems with RNNs and how to fix them
  • More complex RNN variants

• Next lecture we’ll learn:
  • How we can do Neural Machine Translation (NMT) using an RNN-based architecture called sequence-to-sequence with attention
Today’s lecture

- **Vanishing gradient problem**

- Two new types of RNN: LSTM and GRU

- Other fixes for vanishing (or exploding) gradient:
  - Gradient clipping
  - Skip connections

- More fancy RNN variants:
  - Bidirectional RNNs
  - Multi-layer RNNs

Lots of important definitions today!
Vanishing gradient intuition
Vanishing gradient intuition

\[
\frac{\partial J^{(4)}}{\partial h^{(1)}} = ?
\]
Vanishing gradient intuition

\[
\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial J^{(4)}}{\partial h^{(2)}}
\]

chain rule!
Vanishing gradient intuition

\[
\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial J^{(4)}}{\partial h^{(3)}}
\]

chain rule!
Vanishing gradient intuition

\[
\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}
\]

chain rule!
Vanishing gradient intuition

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

What happens if these are small?

\[
\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(4)}}{\partial h^{(3)}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}
\]
Vanishing gradient proof sketch

• Recall: 
  \[ h^{(t)} = \sigma \left( W_h h^{(t-1)} + W_x x^{(t)} + b_1 \right) \]

• Therefore: 
  \[ \frac{\partial h^{(t)}}{\partial h^{(t-1)}} = \text{diag} \left( \sigma' \left( W_h h^{(t-1)} + W_x x^{(t)} + b_1 \right) \right) W_h \] (chain rule)

• Consider the gradient of the loss \( J^{(i)}(\theta) \) on step \( i \), with respect to the hidden state \( h^{(j)} \) on some previous step \( j \).

\[
\frac{\partial J^{(i)}(\theta)}{\partial h^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \prod_{j < t \leq i} \frac{\partial h^{(t)}}{\partial h^{(t-1)}}
\]

\[
= \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} W_h^{(i-j)} \prod_{j < t \leq i} \text{diag} \left( \sigma' \left( W_h h^{(t-1)} + W_x x^{(t)} + b_1 \right) \right)
\]

If \( W_h \) is small, then this term gets vanishingly small as \( i \) and \( j \) get further apart.

Vanishing gradient proof sketch

• Consider matrix L2 norms:

\[ \left\| \frac{\partial J^{(i)}(\theta)}{\partial h^{(j)}} \right\| \leq \left\| \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \right\| \left\| W_h \right\|^{(i-j)} \prod_{j < t \leq i} \right\| \text{diag} \left( \sigma' \left( W_h h^{(t-1)} + W_x x^{(t)} + b_1 \right) \right) \right\|

• Pascanu et al showed that that if the largest eigenvalue of $W_h$ is less than 1, then the gradient $\left\| \frac{\partial J^{(i)}(\theta)}{\partial h^{(j)}} \right\|$ will shrink exponentially
  • Here the bound is 1 because we have sigmoid nonlinearity

• There’s a similar proof relating a largest eigenvalue $>1$ to exploding gradients

Why is vanishing gradient a problem?

Gradient signal from faraway 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.
Why is vanishing gradient a problem?

- **Another explanation:** Gradient can be viewed as a measure of the effect of the past on the future.

- If the gradient becomes vanishingly small over longer distances (step $t$ to step $t+n$), then we can’t tell whether:
  1. There’s no dependency between step $t$ and $t+n$ in the data
  2. We have wrong parameters to capture the true dependency between $t$ and $t+n$
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
Effect of vanishing gradient on RNN-LM

- **LM task:** The writer of the books ____

- **Correct answer:** The writer of the books is planning a sequel

- **Syntactic recency:** The writer of the books is (correct)

- **Sequential recency:** The writer of the books are (incorrect)

- Due to vanishing gradient, RNN-LMs are better at learning from **sequential recency** than **syntactic recency**, so they make this type of error more often than we’d like [Linzen et al 2016]

Why is \textit{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 \textit{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 \texttt{Inf} or \texttt{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

\[
\hat{g} \leftarrow \frac{\partial E}{\partial \theta} \\
\text{if } \|\hat{g}\| \geq \text{threshold then} \\
\hat{g} \leftarrow \frac{\text{threshold}}{\|\hat{g}\|} \hat{g} \\
\text{end if}
\]

**Algorithm 1** Pseudo-code for norm clipping

• **Intuition**: take a step in the same direction, but a smaller step

Gradient clipping: solution for exploding gradient

- This shows the loss surface of a simple RNN (hidden state is a scalar not a vector)
- The “cliff” is dangerous because it has steep gradient
- On the left, gradient descent takes two very big steps due to steep gradient, resulting in climbing the cliff then shooting off to the right (both bad updates)
- On the right, gradient clipping reduces the size of those steps, so effect is less drastic

How to fix 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

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

• How about a RNN with separate memory?
Long Short-Term Memory (LSTM)

- 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 erase, write and read 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 (LSTM)

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

$$f^{(t)} = \sigma \left( W_f h^{(t-1)} + U_f x^{(t)} + b_f \right)$$
$$i^{(t)} = \sigma \left( W_i h^{(t-1)} + U_i x^{(t)} + b_i \right)$$
$$o^{(t)} = \sigma \left( W_o h^{(t-1)} + U_o x^{(t)} + b_o \right)$$

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

All these are vectors of same length $n$.

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

You can think of the LSTM equations visually like this:

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

You can think of the LSTM equations visually like this:

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/
How does LSTM solve vanishing gradients?

• The LSTM architecture makes it easier for the RNN to preserve information over many timesteps
  • e.g. if the forget gate is set to remember everything on every timestep, then the info in the cell is preserved indefinitely
  • By 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
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, image captioning
  - LSTM became the dominant approach

- Now (2019), other approaches (e.g. Transformers) have become more dominant for certain tasks.
  - For example in WMT (a MT conference + competition):
    - In WMT 2016, the summary report contains ”RNN” 44 times
    - In WMT 2018, the report contains “RNN” 9 times and “Transformer” 63 times

Gated Recurrent Units (GRU)

- Proposed by Cho et al. in 2014 as a simpler alternative to the LSTM.
- On each timestep $t$ we have input $x^{(t)}$ and hidden state $h^{(t)}$ (no cell state).

**Update gate:** controls what parts of hidden state are updated vs preserved

**Reset gate:** controls what parts of previous hidden state are used to compute new content

**New hidden state content:** reset gate selects useful parts of prev hidden state. Use this and current input to compute new hidden content.

**Hidden state:** update gate simultaneously controls what is kept from previous hidden state, and what is updated to new hidden state content

How does this solve vanishing gradient? Like LSTM, GRU makes it easier to retain info long-term (e.g. by setting update gate to 0)

\[
\begin{align*}
  u^{(t)} &= \sigma \left( W_u h^{(t-1)} + U_u x^{(t)} + b_u \right) \\
  r^{(t)} &= \sigma \left( W_r h^{(t-1)} + U_r x^{(t)} + b_r \right) \\
  \tilde{h}^{(t)} &= \tanh \left( W_h (r^{(t)} \circ h^{(t-1)}) + U_h x^{(t)} + b_h \right) \\
  h^{(t)} &= (1 - u^{(t)}) \circ h^{(t-1)} + u^{(t)} \circ \tilde{h}^{(t)}
\end{align*}
\]
LSTM vs GRU

• Researchers have proposed many gated RNN variants, but LSTM and GRU are the most widely-used.

• The biggest difference is that GRU is quicker to compute and has fewer parameters.

• There is no conclusive evidence that one consistently performs better than the other.

• LSTM is a good default choice (especially if your data has particularly long dependencies, or you have lots of training data).

• Rule of thumb: start with LSTM, but switch to GRU if you want something more efficient.
Is vanishing/exploding gradient just a RNN problem?

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

For example:
- Residual connections aka “ResNet”
- Also known as skip-connections
- The identity connection preserves information by default
- This makes deep networks much easier to train

Is vanishing/exploding gradient just a RNN problem?

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

For example:

- Dense connections aka “DenseNet”
- Directly connect everything to everything!

Is vanishing/exploding gradient just a RNN problem?

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

For example:
- Highway connections aka “HighwayNet”
- Similar to residual connections, but the identity connection vs the transformation layer is controlled by a dynamic gate
- Inspired by LSTMs, but applied to deep feedforward/convolutional networks

Is vanishing/exploding gradient just a RNN problem?

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

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

• **Conclusion**: Though vanishing/exploding gradients are a general problem, **RNNs are particularly unstable** due to the repeated multiplication by the **same** weight matrix [Bengio et al, 1994]

Recap

• Today we’ve learnt:
  • **Vanishing gradient problem**: what it is, why it happens, and why it’s bad for RNNs
  • **LSTMs and GRUs**: more complicated RNNs that use gates to control information flow; they are more resilient to vanishing gradients

• Remainder of this lecture:
  • **Bidirectional** RNNs
  • **Multi-layer** RNNs

Both of these are pretty simple
Sentence encoding

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.

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)

Task: Sentiment Classification
Bidirectional RNNs

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

Concatenated hidden states

Backward RNN

Forward RNN

the movie was terribly exciting!
On timestep $t$:

- **Forward RNN**
  $$\overrightarrow{h}(t) = \text{RNN}_{FW}(\overrightarrow{h}(t-1), x(t))$$

- **Backward RNN**
  $$\overleftarrow{h}(t) = \text{RNN}_{BW}(\overleftarrow{h}(t+1), x(t))$$

- **Concatenated hidden states**
  $$h(t) = [\overrightarrow{h}(t); \overleftarrow{h}(t)]$$

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

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

Generally, these two RNNs have separate weights.
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.

- 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.
  - You will learn more about BERT later in the course!
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 $i+1$

```
RNN layer 3

RNN layer 2

RNN layer 1

the       movie       was       terribly       exciting       !
```
Multi-layer RNNs in practice

- 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
  - However, skip-connections/dense-connections are needed to train deeper RNNs (e.g. 8 layers)

- Transformer-based networks (e.g. BERT) can be up to 24 layers
  - You will learn about Transformers later; they have a lot of skipping-like connections
In summary

Lots of new information today! What are the **practical takeaways**?

1. LSTMs are powerful but GRUs are faster
2. Clip your gradients
3. Use bidirectionality when possible
4. Multi-layer RNNs are powerful, but you might need skip/dense-connections if it’s deep