Training Neural Networks
Intuition: training a 2-layer Network

Actual answer $y$

System output $\hat{y}$

Training instance $x_1, x_n$

Loss function $L(\hat{y}, y)$

Data flows forward

Loss flows backward
Intuition: Training a 2-layer network

For every training tuple \((x, y)\):

- Run **forward** computation to find our estimate \(\hat{y}\).
- Run **backward** computation to update weights:
  - For every output node:
    - Compute loss \(L\) between \(y\) and \(\hat{y}\).
    - For every weight from hidden layer to output layer:
      - Update the weight.
  - Assess how much blame it deserves.
  - For every weight \(w\) from input layer to the hidden layer:
    - Update the weight proportionally to its blame.
Loss Function: a measure of how far off the current answer is from the right answer.

For binary logistic regression, we use cross entropy loss:

\[
L_{CE} (\hat{y}, y) = - \left[ y \log \sigma (wx + b) + (1-y) \log (1 - \sigma (wx + b)) \right]
\]
Loss Function: a measure of how far off the current answer is from the right answer.

For multinominal classification, we use cross entropy loss:

$$L_{CE}(\hat{y}, y) = - \log \frac{e^{\hat{z}_y}}{\sum_{i} e^{\hat{z}_i}}$$

Takeaway:
cross-entropy loss = the log of the output probability of the correct class.
Gradient descent for weight updates

The derivative of the loss function with respect to weights tells us how to adjust the weights to make better predictions.

Derivative of the loss function:

$$ \frac{\partial L(f(x;w), y)}{\partial w} $$

We want to move the weights in the opposite direction of the gradient:

$$ w^{t+1} = w^t - \text{LR} \left( \frac{\partial L(f(x;w), y)}{\partial w} \right) $$

For logistic regression:

$$ \frac{\partial L_{\text{CE}}(w, b)}{\partial w_j} = (\hat{y} - y) x_j = \sigma'(w x_j + b) y x_j $$
Where did that derivative come from?

Each node takes an upstream gradient, multiplies it by the local gradient (the gradient of its output with respect to its input), and uses the chain rule to compute a downstream gradient to be passed on to a prior node.

A node may have multiple local gradients if it has multiple inputs.
Why Computation Graphs

For training, we need the derivative of the loss with respect to each weight in every layer of the network.

**Problem**: the derivatives on the prior slide only give the updates for one weight layer: the last one, since loss is computed only at the very end of the network!

**Solution**: error backpropagation (Rumelhart, Hinton, Williams, 1986)

- Backprop is a special case of backward differentiation
- Which relies on **computation graphs**.
Computation Graphs
Computation Graphs

A computation graph represents the process of computing a mathematical expression

\[ L(a, b, c) = c(a + 2b) \]

Computations:
- \( d = 2 \times b \)
- \( e = a + d \)
- \( L = c \times e \)
The importance of the computation graph comes from the backward pass. This is used to compute the derivatives that we’ll need for the weight update.
The chain rule

Computing the derivative of a composite function:

\[ f(x) = u(v(x)) \]

\[
\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}
\]

\[ f(x) = u(v(w(x))) \]

\[
\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}
\]
Example

\[
\frac{\partial L}{\partial a} - \frac{\partial L}{\partial e} \frac{\partial e}{\partial a} = c
\]

\[
\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e} \frac{\partial e}{\partial d} \frac{\partial d}{\partial b}
\]

\[
L = ce
\]

\[
\frac{\partial L}{\partial e} = c
\]

\[
\frac{\partial e}{\partial d} = 1
\]

\[
\frac{\partial e}{\partial d} = 1
\]

\[
d = 2b
\]

\[
\frac{\partial d}{\partial b} = 2
\]

\[
w_{t+1} = w_t - LR \left( \frac{\partial L}{\partial w_t} \right)
\]
Summary

For training, we need the derivative of the loss with respect to weights in early layers of the network.

- But loss is computed only at the very end of the network!

Solution: \textit{backward differentiation}

Given a computation graph and the derivatives of all the functions in it we can automatically compute the derivative of the loss with respect to these early weights.
Language Modeling
Language Modeling

Given a sequence, predicts what comes next:

I write this sitting in the kitchen …
Language Modeling

We will talk about this in terms of predicting the next word in a sentence (language generation).

But it turns out to be a very useful task. We could predict any kind of sequence. It is also useful for learning representations.

Language models are used in:
- Google search
- Auto-correction / auto-complete
- Speech recognition
- Voice assistants
N-gram Language Models
Most modern applications use neural network language models. But to understand the language modeling task, we will start with the workhorse of the language model world: the n-gram language model.
A language model is a model that computes the probability of a sentence in a language:

\[ p(w_0, \ldots, w_n) \]

A language model can also be used to compute the probability of the next word in a sentence:

\[ p(w_n | w_0, \ldots, w_{n-1}) \]
Sentence probability

How do we guess what the next word is?

I write this sitting in the kitchen...
How do we guess what the next word is?

I write this sitting in the kitchen...

Answer: try out different words and compare their likelihood.

\[ p(I, \text{write}, \text{this}, \text{sitting}, \text{in}, \text{the}, \text{kitchen}, \text{sink}) \]

\[ \textbf{versus} \]

\[ p(I, \text{write}, \text{this}, \text{sitting}, \text{in}, \text{the}, \text{kitchen}, \text{knife}) \]

\[ \textbf{versus} \]

\[ p(I, \text{write}, \text{this}, \text{sitting}, \text{in}, \text{the}, \text{kitchen}, \text{chair}) \]
p(I, write, this, sitting, in, the, kitchen, sink)

Normally, events aren’t ordered in probability notation. But when we are working with language models, this is shorthand for:
p(I, write, this, sitting, in, the, kitchen, sink)

Normally, events aren’t ordered in probability notation. But when we are working with language models, this is shorthand for:

p(w_0=I, w_1=write, w_2=this, w_3=sitting, w_4=in, w_5=the, w_6=kitchen, w_7=sink)
How do we calculate these probabilities?

\[
\begin{align*}
p(I, \text{write, this, sitting, in, the, kitchen, sink}) \\
\text{versus} \\
p(I, \text{write, this, sitting, in, the, kitchen, knife}) \\
\text{versus} \\
p(I, \text{write, this, sitting, in, the, kitchen, chair})
\end{align*}
\]
If we had a big corpus, we could see how often each of these sentences occurred:

\[
\frac{\text{count(“I write this sitting in the kitchen sink”)}}{\text{count(“I write this sitting in the kitchen”)}} \quad \text{versus} \quad \frac{\text{count(“I write this sitting in the kitchen knife”)}}{\text{count(“I write this sitting in the kitchen”)}} \quad \text{versus} \quad \frac{\text{count(“I write this sitting in the kitchen chair”)}}{\text{count(“I write this sitting in the kitchen”)}}
\]

This is called a \textit{maximum likelihood estimation}. 
Frequency Counts

But what if we never see any of these sentences?

0/ count(“I write this sitting in the kitchen”) versus
0/ count(“I write this sitting in the kitchen”) versus
0/ count(“I write this sitting in the kitchen”)

The probabilities would all be zero. But intuitively, some of these sentences still seem more likely than others…
The Chain Rule

Chain Rule of Probability:

\[ P(X_1 \ldots X_n) = P(X_1)P(X_2 \mid X_1)P(X_3 \mid X_{1:2}) \ldots P(X_n \mid X_{1:n-1}) \]

\[ = \prod_{k=1}^{n} P(X_k \mid X_{1:k-1}) \]

Applied to a sentence:

\[ P(w_{1:n}) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_{1:2}) \ldots P(w_n \mid w_{1:n-1}) \]

\[ = \prod_{k=1}^{n} P(w_k \mid w_{1:k-1}) \]

\[ P(\text{"I write this sitting in the kitchen sink"}) = \]

\[ P(\text{"I"})P(\text{"write"} \mid \text{"I"})P(\text{"this"} \mid \text{"I write"}) \ldots P(\text{"sink"} \mid \text{"I write this sitting in the kitchen"}) \]
The Chain Rule

Great, so now we have:

\[
P(\text{"I"})P(\text{"write"} \mid \text{"I"})P(\text{"this"} \mid \text{"I write"}) \ldots P(\text{"sink"} \mid \text{"I write this sitting in the kitchen"})
\]

\text{versus}

\[
P(\text{"I"})P(\text{"write"} \mid \text{"I"})P(\text{"this"} \mid \text{"I write"}) \ldots P(\text{"knife"} \mid \text{"I write this sitting in the kitchen"})
\]

\text{versus}

\[
P(\text{"I"})P(\text{"write"} \mid \text{"I"})P(\text{"this"} \mid \text{"I write"}) \ldots P(\text{"chair"} \mid \text{"I write this sitting in the kitchen"})
\]
The Chain Rule

But, since we never saw "I write this sitting in the kitchen sink", we still don’t have a way to calculate

\[ p(\text{"sink"} \mid \text{"I write this sitting in the kitchen"}) \]

What can we do?
Markov Assumption

For a very simple language model, we could estimate this conditional probability by looking at a smaller context window: a single word.

\[ p(\text{sink} \mid \text{I, write, this, sitting, in, the, kitchen}) \approx p(\text{sink} \mid \text{kitchen}) \]

Maybe we never see "I write this sitting in the kitchen sink". But we still see "kitchen sink".
A **bigram language model** is a language model that makes a **Markov assumption**: the probability of a word is conditioned solely on the previous word.

Bigram language model:

\[ p(w_n | w_{1:n-1}) \approx p(w_n | w_{n-1}) \]

\[ p(\text{"I write this sitting in the kitchen sink"}) \approx p(\text{"I"})p(\text{"write" | "I"})p(\text{"this" | "write"})p(\text{"sitting" | "this"})p(\text{"in" | "sitting"})p(\text{"the" | "in"})p(\text{"kitchen" | "the"})p(\text{"sink" | "kitchen"}) \]
Bigram language model

In a bigram model, we have:

\[
p('I')p('write' | 'I')p('this' | 'write')p('sitting' | 'this')p('in' | 'sitting')p('the' | 'in')p('kitchen' | 'the')p('sink' | 'kitchen')
\]

versus

\[
p('I')p('write' | 'I')p('this' | 'write')p('sitting' | 'this')p('in' | 'sitting')p('the' | 'in')p('kitchen' | 'the')p('knife' | 'kitchen')
\]

versus

\[
p('I')p('write' | 'I')p('this' | 'write')p('sitting' | 'this')p('in' | 'sitting')p('the' | 'in')p('kitchen' | 'the')p('chair' | 'kitchen')
\]
Maximum Likelihood Estimates: Bigram

Let’s get some estimates! I’ll use the Q corpus from the NLTK library to estimate our probabilities under a Maximum Likelihood Estimate.

\[ p(\text{"sink" | } \text{"kitchen"}) = \frac{4}{138} \]
\[ p(\text{"knife" | } \text{"kitchen"}) = \frac{2}{138} \]
\[ p(\text{"chair" | } \text{"kitchen"}) = \frac{1}{138} \]
Maximum Likelihood Estimates: Bigram

We can also look at the overall probability of the sentence:

\[
p(\text{"I write this sitting in the kitchen sink"}) = \\
p(\text{"I"})p(\text{"write" | "I"})p(\text{"this" | "write"})p(\text{"sitting" | "this"})p(\text{"in" | "sitting"})p(\text{"the" | "in"})p(\text{"kitchen" | "the"})p(\text{"sink" | "kitchen"})
\]

\[
p(\text{"I"}) = ?? \\
p(\text{"write" | "I"}) = 6/9329 \\
p(\text{"this" | "write"}) = 17/394 \\
p(\text{"sitting" | "this"}) = ??/18642 \\
p(\text{"in" | "sitting"}) = 53/257 \\
p(\text{"the" | "in"}) = 19738/75468 \\
p(\text{"kitchen" | "the"}) = 96/220868 \\
p(\text{"sink" | "kitchen"}) = 4/138 \]
Two questions left:
- What do we do about the first word?
- What do we do about missing words?
Maximum Likelihood Estimates: Bigram

What do we do about the **first word**? Two options:

✦ Take the overall probability of "I", the word count for "I" divided by number of tokens: $9329 \div 5058449$

*How many times did we see "I" compared to all other words?*

✦ Add a START token and condition "I" on this start token:

*How many times did sentences start with "I" compared to other words?*

\[
p("I") = \frac{9329}{5058449}
\]

\[
p("write" | "I") = \frac{6}{9329}
\]

\[
p("this" | "write") = \frac{17}{394}
\]

\[
p("sitting" | "this") = \frac{??}{18642}
\]

\[
p("in" | "sitting") = \frac{53}{257}
\]

\[
p("the" | "in") = \frac{19738}{75468}
\]

\[
p("kitchen" | "the") = \frac{96}{220868}
\]

\[
p("sink" | "kitchen") = \frac{4}{138}
\]
Maximum Likelihood Estimates: Bigram

What do we do about the **missing words**? This is harder. One easy solution is called **add-one (Laplace) smoothing**: just pretend we saw each word 1 more time. This changes zeros to ones.

\[
\begin{align*}
p("I") &= 9329 + 1 / 5058449 + V \\
p("write" | "I") &= 6 + 1 / 9329 + V \\
p("this" | "write") &= 17 + 1 / 394 + V \\
p("sitting" | "this") &= 0 + 1 / 18642 + V \\
p("in" | "sitting") &= 53 + 1 / 257 + V \\
p("the" | "in") &= 19738 + 1 / 75468 + V \\
p("kitchen" | "the") &= 96 + 1 / 220868 + V \\
p("sink" | "kitchen") &= 4 + 1 / 138 + V
\end{align*}
\]
**Smoothing** steals probability from words that are in the training data to give to unseen words.

Add-one smoothing is not a very good way of doing this. There are better options, but they are more complicated.
The intuition of smoothing (from Dan Klein)

When we have sparse statistics:

\[
P(w \mid \text{denied the})
\]

- 3 allegations
- 2 reports
- 1 claim
- 1 request
- 7 total

Steal probability mass to generalize better

\[
P(w \mid \text{denied the})
\]

- 2.5 allegations
- 1.5 reports
- 0.5 claims
- 0.5 request
- 2 other
- 7 total
More Context, Better Predictions
Generalizing to n-grams

We can improve our context by looking at larger window sizes. For instance, if we considered more context, we might be able to capture that "in the kitchen knife" is not a good completion because it is rare to be in knives.

Our bigram model predicts "knife" is a better completion than "chair" in our context, because kitchen knives are more frequent than kitchen chairs. "in" is outside its context window.
Generalizing the n-gram model

Generic n-gram model:

\[ P(w_n | w_{1:n-1}) \approx P(w_n | w_{n-N+1:n-1}) \]

where \( N \) is the context window.
Language Models for Language Generation
Language Generation

So far we have used language models to predict the next word in a sequence and estimate the probability of a sentence.

We can also use them to generate novel sentences.

We do this by sampling words according to their estimated probabilities.
Language Generation

\[
\begin{align*}
P(\text{english} \mid \text{want}) &= 0.0011 \\
P(\text{chinese} \mid \text{want}) &= 0.0065 \\
P(\text{to} \mid \text{want}) &= 0.66 \\
P(\text{eat} \mid \text{to}) &= 0.28 \\
P(\text{food} \mid \text{to}) &= 0 \\
P(\text{want} \mid \text{spend}) &= 0 \\
P(i \mid <s>) &= 0.25
\end{align*}
\]
Language Generation

- Choose a random bigram \(<s>, w\) according to its probability.
- Then choose a random bigram \((w, x)\) according to its probability.
- Repeat until we choose \(</s>\).

\(<s>
I
want
to
eat
Chinese
food
</s>

I want to eat Chinese food
Evaluation
Evaluation: How good is our model?

Does our language model prefer good sentences to bad ones?

Does it assign higher probability to “real” or “frequently observed” sentences than “ungrammatical” or “rarely observed” sentences?
Evaluation: How good is our model?

We train parameters of our model on a training set.

We test the model’s performance on data we haven’t seen.

- A test set is an unseen dataset that is different from our training set, totally unused.
- An evaluation metric tells us how well our model does on the test set.
Perplexity

The best language model is one that best predicts an unseen test set (gives the highest $P(\text{sentence})$).

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

Chain rule: $\text{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1 \ldots w_{i-1})}}$

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

Minimizing perplexity is the same as maximizing probability
In practice, we use log probabilities to avoid numerical underflow:

**Perplexity**: exponentiated token-level negative log-likelihood.

$$PP(W) = \exp\left(-\frac{1}{N} \sum_{i}^{N} \log p(w_i | w_{<i})\right)$$
Lower perplexity = better model

Training 38 million words, test 1.5 million words, WSJ

<table>
<thead>
<tr>
<th>N-gram Order</th>
<th>Unigram</th>
<th>Bigram</th>
<th>Trigram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perplexity</td>
<td>962</td>
<td>170</td>
<td>109</td>
</tr>
</tbody>
</table>