CSC413 Neural Networks and Deep Learning

Lecture 3: Automatic differentiation, distributed representation, and GloVe embedding

January 23/25, 2024

Table of Contents

Lecture Plan

Last week:

- From linear models to **multilayer perceptrons**
- Backpropagation to compute gradients efficiently

This week:

- Automatic differentiation
- Distributed representations
- GloVe embeddings

Section 1

Automatic Differentiation (Autodiff)

Derivatives in Machine Learning

The machine learning approach requires the minimization of some cost/loss function, which is often done using some variation of **gradient descent**.

$$\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{E}}{\partial \theta}$$

Approaches to computing derivatives:

1. Manually working out derivatives
2. Numeric differentiation (using finite difference approximations)
3. Symbolic differentiation (using expression manipulation)
4. **Automatic differentiation or algorithmic differentiation**

Terminology

- **Automatic differentiation (Autodiff)**: refers to a general way of taking a program, which computes a value, and automatically constructing a procedure for computing the derivatives of that value.
  - Convert the program into a sequence of primitive operations, which have specified routines for computing derivatives, and then computing gradients in a mechanical way via the chain rule.
  - Also used in computational fluid dynamics, atmospheric sciences, etc.
- **Backpropagation**: special case of autodiff where the *program* is a neural network forward pass.
- Autograd, JAX, PyTorch, TensorFlow are examples of particular implementations of autodiff, i.e., different libraries.

Backpropagation

Steps:

- Convert the computation into a sequence of **primitive operations**
    - Primitive operations have easily computed derivatives
- Build the computation graph
- Perform a forward pass: compute the values of each node
- Perform the backward pass: compute the derivative of the loss with respect to each node

Autodiff, more generally

We discuss how an automatic differentiation library could be implemented at the high level.

- build the computation graph
- **vector-Jacobian products (VJP)** for primitive ops
- perform the backward pass

You will probably never have to implement autodiff yourself but it is good to know its inner workings!

Key Insight: For any new deep learning model that we can come up with, if each step of our computation is differentiable, then we can train that model using gradient descent.

Scalar Example

```
def f(x):
    h = 1.5
    for i in range(3):
        h = x * 1.5 + h
    return x * h
```

Notation: x is the input, $y = f(x)$ is the output, we want to compute $\frac{dy}{dx}$

Automatic Differentiation Steps:

- convert the computation into a sequence of **primitive operations**
    - we need to be able to compute derivatives for these primitive operations
- build the computation graph
- perform forward pass
- perform backward pass

Scalar Example: Primitive Ops

```
def f(x):
    h = 1.5
    for i in range(3):
        h = x * 1.5 + h
    return x * h
```

Operations:

Scalar Example: Primitive Ops

```python
def f(x):
    h = 1.5
    for i in range(3):
        h = x * 1.5 + h
    return x * h
```

Operations:

```
h0 = 1.5
z1 = x * 1.5
h1 = z1 + h0
z2 = x * 1.5
h2 = z2 + h1
z3 = x * 1.5
h3 = z3 + h2
y  = x * h3
```

Scalar Example: Computation Graph

Exercise: Draw the computation graph:

```
h0 = 1.5
z1 = x * 1.5
h1 = z1 + h0
z2 = x * 1.5
h2 = z2 + h1
z3 = x * 1.5
h3 = z3 + h2
y  = x * h3
```

Based on the computation graph, we can compute $\frac{dy}{dx}$ via a forward and a backward pass.

Vector Inputs and Outputs

More generally, input/output to a computation may be **vectors**

```
def f(a, w): # a and w are both vectors with size 10
    h = a
    for i in range(3):
        h = np.dot(w, h) + h
    z = w * h # element wise multiplication
    return z
```

So we have $\mathbf{y} = \mathbf{f}(\mathbf{x})$ (in this example, $\mathbf{x}$ consists of values in both `a` and `w`)

Q: In our running example, what are the dimensions of $\mathbf{x}$ and $\mathbf{y}$?

The Jacobian Matrix

We wish to compute the partial derivative $\frac{\partial y_k}{\partial x_i}$ for each k and i, at some $\mathbf{x}$.

If we consider all k and i, these partial derivatives form the **Jacobian matrix** of $\mathbf{y}$ w.r.t. $\mathbf{x}$:

$$J_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1}(\mathbf{x}) & \dots & \frac{\partial y_1}{\partial x_n}(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1}(\mathbf{x}) & \dots & \frac{\partial y_m}{\partial x_n}(\mathbf{x}) \end{bmatrix}$$

Note that we usually want to avoid explicitly constructing the entries of this Jacobian one by one.

Why? Computing all the partial derivatives one by one is expensive, even with backprop.

Decomposing Into Primitive Operations

Suppose $f = f_2 \circ f_1$, so we have the computations $\mathbf{y} = f_2 \circ f_1(\mathbf{x})$, or in other words:

$$\mathbf{z} = f_1(\mathbf{x})$$
$$\mathbf{y} = f_2(\mathbf{z})$$

If f_1 and f_2 are primitive operations with simple Jacobians, we can apply the **Jacobian chain rule**:

$$J_{f_2 \circ f_1}(\mathbf{x}) = \mathbf{J}_{\mathbf{f_2}}(\mathbf{z})\mathbf{J}_{\mathbf{f_1}}(\mathbf{x})$$

Avoiding Jacobian Products

In practice, computing entries of Jacobians one by one is expensive and we try to avoid it:

- If the dimension of $\mathbf{y} = f(\mathbf{x})$ is small, use **reverse-mode automatic differentiation**
- If the dimension of $\mathbf{x}$ is small, use **forward-mode automatic differentiation**

Q: Which of these two cases apply to deep learning most often?

Reverse-Mode Automatic Differentiation

Suppose $\mathbf{y}$ is a scalar, and represents the loss $\mathcal{L}$ that we wish to minimize.

$$\mathbf{z} = f_1(\mathbf{x})$$
$$\mathcal{L} = f_2(\mathbf{z}) = \mathbf{y} \in \mathbb{R}$$

Then we have:

- $\bar{z} = \frac{\partial \mathcal{L}}{\partial z} = J_{f_2}(\mathbf{z})^{\mathbf{T}}$
- Since $\bar{x_j} = \sum_i \bar{z}_i \frac{\partial z_i}{\partial x_j}$
- ... we have $\bar{\mathbf{x}}^T = \bar{\mathbf{z}}^T J_{f_1}(\mathbf{x})$
- ... which is a **vector-Jacobian product**

Summary: For each primitive operation, we don't need to be able to compute entire Jacobian matrix. **We need to be able to compute the vector-Jacobian product.**

Vector Jacobian Products

For each primitive operation, we must specify the VJPs for each of its arguments

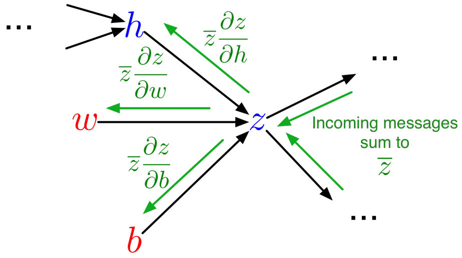
The VJP function should take in the output gradient (i.e. $\bar{y}$), the answer (y), and the arguments (x), and returns the input gradient ($\bar{x}$)

Here are some examples from https://github.com/mattjj/autodidact/blob/master/autograd/numpy/numpy_vjps.py

```
defvjp(anp.negative, lambda g, ans, x: -g)
defvjp(anp.exp,    lambda g, ans, x: ans * g)
defvjp(anp.log,    lambda g, ans, x: g / x)

defvjp(anp.add,         lambda g, ans, x, y : unbroadcast(x, g),
                        lambda g, ans, x, y : unbroadcast(y, g))
defvjp(anp.multiply,    lambda g, ans, x, y : unbroadcast(x, y * g),
                        lambda g, ans, x, y : unbroadcast(y, x * g))
defvjp(anp.subtract,    lambda g, ans, x, y : unbroadcast(x, g),
                        lambda g, ans, x, y : unbroadcast(y, -g))
```

Backprop as Message Passing



...
h
$\overline{z}\frac{\partial z}{\partial h}$
$\overline{z}\frac{\partial z}{\partial w}$
w
z
$\overline{z}\frac{\partial z}{\partial b}$
b
...
Incoming messages sum to $\overline{z}$
...

- Each node in the computation graph receives **messages** from its children, which it aggregates to compute its error signal
- **Messages** then get passed to its parents
- Each message is a VJP

This design provides **modularity!** Each node needs to know how to compute its outgoing messages, i.e. the VJPs corresponding to each of its parents (arguments to the function).

Differentiable Programming

Recall the **key insight** from earlier: For any new deep learning model that we can come up with, if each step of our computation is differentiable, then we can train that model using gradient descent.

Example: Learning to learning by gradient descent by gradient descent `https://arxiv.org/abs/1606.04474`

With AD, any *program* that has differentiable components can be optimized via gradient descent

Autodiff, more generally

This video explains the different ways to automatically compute derivatives:

`https://www.youtube.com/watch?v=wG_nF1awSSY`

- manual
- finite differences
- symbolic differentiation
- autodiff (forward-mode and reverse-mode differentiation)
  - how to avoid computing Jacobians one by one

Section 2

Distributed Representations

Feature Mapping

- Learning good *representations* is an important goal in machine learning
  - These representations are also called *feature mappings*, or *embeddings*
  - The representations we learn are often **reusable** for other tasks
  - We can find good representations through an **unsupervised learning** formulation

Language Modeling

A language model...

- Models the probability distribution of natural language text.
- Determine the **probability** $p(\mathbf{s})$ that a sequence of words (or a *sentence*) $\mathbf{s}$ occurs in text.

A language model gives us a way to compute $p(\mathbf{s})$

Why language models $p(\mathbf{s})$?

- Determine authorship:
  - build a language model $p(\mathbf{s})$ of Shakespeare
  - determine whether a script is written by Shakespeare
- Generate a machine learning paper (given a *corpus* of machine learning papers)
- Use as a *prior* for a speech recognition system $p(\mathbf{s}|\mathbf{a})$, where $\mathbf{a}$ represents the observed speech signal.
  - An **observation model**, or likelihood, represented as $p(\mathbf{a}|\mathbf{s})$, which tells us how likely the sentence $\mathbf{s}$ is to lead to the acoustic signal $\mathbf{a}$.
  - A **prior**, represented as $p(\mathbf{s})$ which tells us how likely a given sentence $\mathbf{s}$ is. For example, "recognize speech" is more likely than "wreck a nice beach"
  - Use Bayes rule to infer a *posterior distribution* over sentences given the speech signal:

$$p(\mathbf{s}|\mathbf{a}) = \frac{p(\mathbf{s})p(\mathbf{a}|\mathbf{s})}{\sum_{\mathbf{s}'} p(s')p(\mathbf{a}|\mathbf{s}')}$$

Training a Language Model

Assume we have a corpus of sentences $\mathbf{s}^{(1)}, \ldots, \mathbf{s}^{(N)}$.

The **maximum likelihood** criterion says we want our model to maximize the probability that our model assigns to the observed sentences. We assume the sentences are independent, so that their probabilities multiply.

In maximum likelihood training, we want to maximize $\prod_{i=1}^{N} p(\mathbf{s}^{(i)})$.

This is equivalent to maximizing $\sum_{i=1}^{N} \log p(\mathbf{s}^{(i)})$, or minimizing

$$-\sum_{i=1}^{N} \log p(\mathbf{s}^{(i)}).$$

Since $p(\mathbf{s})$ is usually small, $-\log p(\mathbf{s})$ is reasonably sized, positive numbers.

Probability of a sentence

A sentence is a sequence of words $w_1, w_2, \ldots, w_T$, so

$$\begin{aligned} p(\mathbf{s}) &= p(w_1, w_2, \ldots, w_T) \\ &= p(w_1)p(w_2|w_1)p(w_3|w_1, w_2) \ldots p(w_T|w_1, w_2, \ldots, w_{T-1}). \end{aligned}$$

We can make a simplifying **Markov assumption** that the distribution over the next word depends on the preceding few words. For example, a context length of 3 means that we approximate

$$p(w_t|w_1, w_2, \ldots, w_{t-1}) \approx p(w_t|w_{t-3}, w_{t-2}, w_{t-1})$$

N-Gram Language Model

A simple way of modeling $p(w_t | w_{t-2}, w_{t-1})$ is by constructing a table of conditional probabilities:

	cat	and	city	$\cdots$
the fat	0.21	0.003	0.01	
four score	0.0001	0.55	0.0001	$\cdots$
New York	0.002	0.0001	0.48	
$\vdots$		$\vdots$		

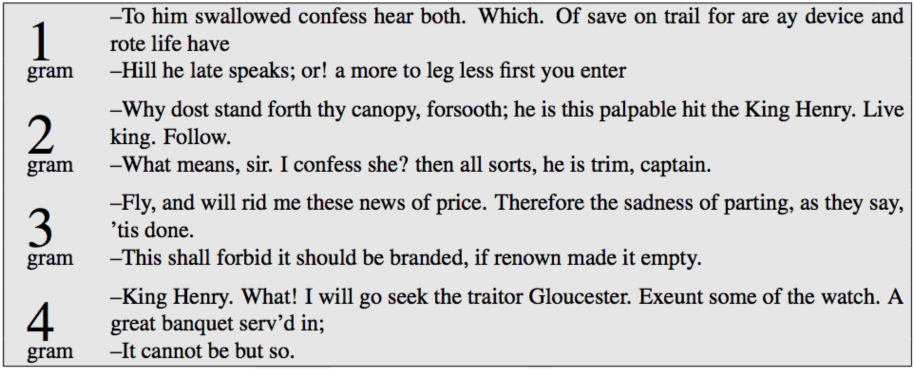
Where the probabilities come from the **empirical distribution**:

$$p(w_3 = \text{cat} | w_1 = \text{the}, w_2 = \text{fat}) = \frac{p(w_1 = \text{the}, w_2 = \text{fat}, w_3 = \text{cat})}{p(w_1 = \text{the}, w_2 = \text{fat})}$$
$$\approx \frac{\text{count(the fat cat)}}{\text{count(the fat)}}.$$

The phrases we're counting are called *n-grams* (where n is the length), so this is an **n-gram language model**. (Note: the above example is considered a 3-gram model, not a 2-gram model!)

Example: Shakespeare N-Gram Language Model

Sentences randomly generated from several n-grams computed from Shakespeare's works.



1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; –It cannot be but so.

Problems with N-Gram Language Model

- The number of entries in the conditional probability table is exponential in the context length.
- **Data sparsity**: most n-grams never appear in the corpus, even if they are possible.

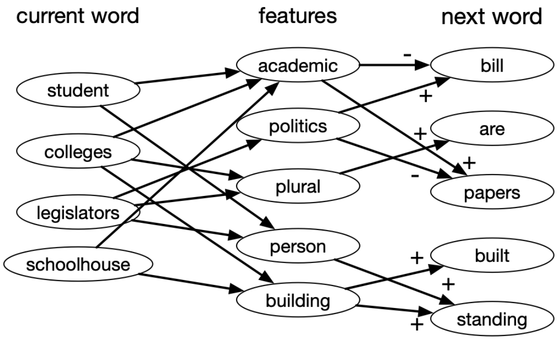
Ways to deal with data sparsity:

- Use a short context (but this means the model is less powerful).
- Smooth the probabilities, e.g. by adding imaginary counts.
- Make predictions using an ensemble of n-gram models with different *n*s.

Localist vs Distributed Representations

Conditional probability tables are a kind of **localist representation**: all the information about a particular word is stored in one place: a column of the table.

But different words are related, so we ought to be able to share information between them.



Distributed Representations: Word Attributes

	academic	**politics**	**plural**	**person**	**building**
students	1	0	1	1	0
colleges	1	0	1	0	1
legislators	0	1	1	1	0
schoolhouse	1	0	0	0	1

Idea:

1. use the **word attributes** to predict the next word.
2. learn the **word attributes** using an MLP with backpropagation

Sharing Information

Distributed representations allows us to share information between related words. E.g., suppose we've seen the sentence

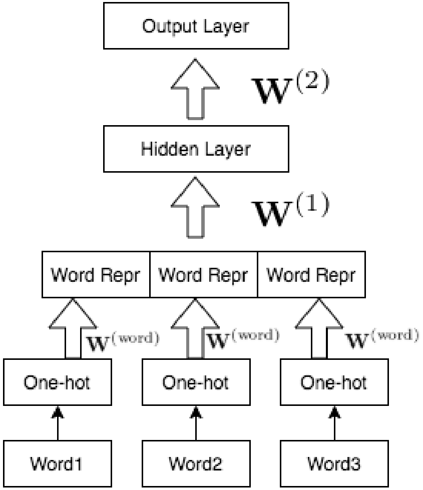
The cat got squashed in the garden on Friday.

This should help us predict the words in the sentence

The dog got flattened in the yard on (???)

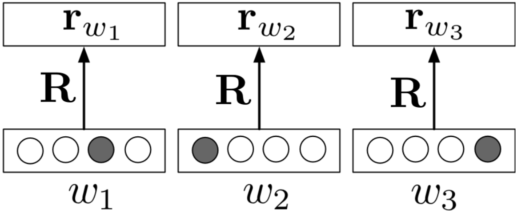
An n-gram model can't generalize this way, but a distributed representation might let us do so.

Neural Language Model (Assignment 1)



Word Representations

Since we are using one-hot encodings for the words, the weight matrix of the word embedding layer acts like a lookup table.

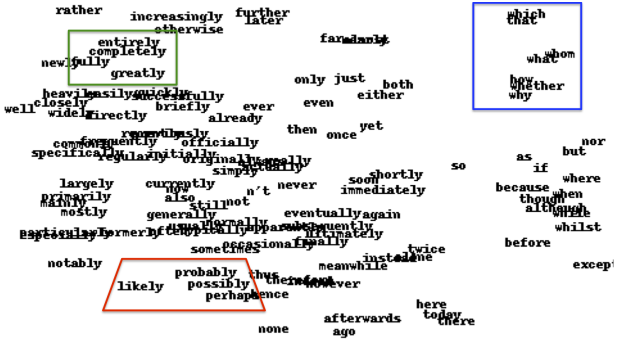


Terminology:

- **Embedding** emphasizes that it's a location in a high-dimensional space; words that are closer together are more semantically similar.
- **Feature vector** emphasizes that it's a vector that can be used for making predictions, just like other feature mappings we've looked at (e.g. polynomials).

What do word embeddings look like?

It's hard to visualize an *n*-dimensional space, but there are algorithms for mapping the embeddings to two dimensions.



In assignment 1, we use algorithm called tSNE, which tries to make distances in the 2-D embedding match the original high-dimensional distances as closely as possible.

A note about these visualizations

- Thinking about high-dimensional embeddings
    - Most vectors are nearly orthogonal (i.e. dot product is close to 0)
    - Most points are far away from each other
    - “In a 30-dimensional grocery store, anchovies can be next to fish and next to pizza toppings” - Geoff Hinton
- The 2D embeddings might be fairly misleading, since they can’t preserve the distance relationship from a higher-dimensional embedding. (Unrelated words might be close together in 2D but far apart in 3D)

Section 3

GloVe Embeddings

GloVe

- Fitting language models is really hard
  - It's really important to make good predictions about relative probabilities of rare words
  - Computing the predictive distribution requires a large softmax
- Maybe this is overkill if all you want is word representations
- Global Vector (GloVe) embeddings are a simpler and faster approach based on a matrix factorization similar to principal component analysis (PCA)

Idea: First fit the distributed word representations using GloVe, then plug these embeddings into a neural net that does some other task (e.g. translation)

Co-occurrence matrix

Consider these sentences: “The cat got squashed in the garden on Friday. The dog got flattened in the yard on Thursday.”

	the	cat	dog	got	squashed
the	0	1	1	2	0
cat	1	0	0	1	1
dog					
got					
squashed					

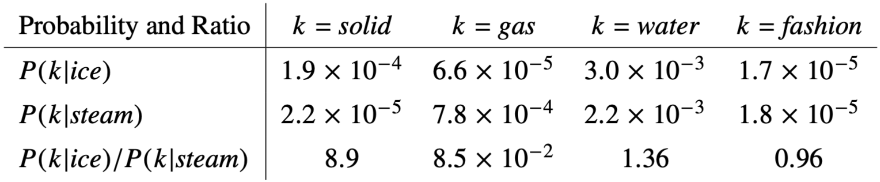
Consider a vocabulary size of V. The **co-occurrence matrix** X is a $V \times V$ matrix that counts the number of times the words appear nearby. Its X_{ij} entry is the number of times word i occurs in the **context** of word j. The context of a word in a sentence is the window of nearby words.

Example: a context of size 5 are two words before and two words after a word. The context of “squashed” is “cat got *squashed* in the”. **Exercise**: Fill the co-occurrence matrix with a context size of 5.

GloVe Embedding Training

Key insight: The co-occurrence matrix of words within the same context (nearby) contain information about the semantic information (meaning) of words

- For example, words “ice” and “water” are more likely to appear closer to each than “ice” and “fashion”.



Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

In particular, the *ratio* of co-occurrences encodes semantic information!

Language Model as Matrix Factorization

Suppose we fit a rank-K approximation

$$\mathbf{X} \approx \mathbf{R}\widehat{\mathbf{R}}^T$$

Where $\mathbf{R}$ and $\widehat{\mathbf{R}}$ are $V \times K$ matrices

- Each row $\mathbf{r}_i$ of $\mathbf{R}$ is the K-dimensional representation of a word.
- Each entry of $\mathbf{X}$ is approximated as $x_{ij} \approx \mathbf{r}_i^\top \widehat{\mathbf{r}}_j$
- Minimizing the squared Frobenius norm of the $||\mathbf{X} - \mathbf{R}\widehat{\mathbf{R}}^T||_F^2$ is basically PCA
- There are some other tricks to make the optimization work

Global Vector (GloVe) Embedding

GloVe is based on this matrix factorization idea, but with some twists. For example:

Problem: Word counts are heavy-tail distributed (some words *very* frequently used, lots of infrequent words). The most common words will dominate the cost function.

- **Solution**: Approximate $\log x_{ij}$ instead of x_{ij}

Problem: $\mathbf{X}$ is extremely large, so fitting the above factorization using least squares is infeasible.

- **Solution**: Reweight the entries so that only nonzero counts matter

GloVe embedding cost function

$$\mathcal{J}(\mathbf{R}) = \sum_{i,j} f(x_{ij})(\mathbf{r}_i^T \widehat{\mathbf{r}}_j + b_i + \hat{b}_j - \log x_{ij})^2$$

- $f(x_{ij}) = (\frac{x_{ij}}{100})^{\frac{3}{4}}$ if $x_{ij} < 100$ and 1 otherwise.
- b_i and $\tilde{b}_j$ are bias parameters.
- We can avoid computing $\log 0$ since $f(0) = 0$.
- We only need to consider the nonzero entries of $\mathbf{X}$. This gives a big computational savings since $\mathbf{X}$ is extremely sparse!

GloVe Embeddings

Pre-trained models are available for download:

`https://nlp.stanford.edu/projects/glove/`

Practitioners often use these embeddings to do other language modeling tasks.

GloVe Embedding Demo

Demo on Google Colab

https://colab.research.google.com/drive/1aNbE6HcawVF67RV0hWi4qK33Um7cKykr?usp=sharing

Key idea from the Demo

- Distances are somewhat meaningful, and are based on **word co-occurrences**
  - the words “black” and “white” will have similar embeddings because they co-occur with similar other words.
  - “cat” and “dog” is more similar to each other than “cat” and “kitten” because the latter two words occur in *different contexts*!
- Word Analogies: Directions in the embedding space can be meaningful
  - “king” - “man” $+$ “woman” $\approx$ “queen”
- Bias in Word Embeddings (and Neural Networks in General)
  - neural networks pick up pattern in the data
  - these patterns can be biased and discriminatory

Bias and Fairness

Word embeddings are inherently biased because there is bias in the training data.

Neural networks learn patterns in the training data, so if the training data contains human biases, then so will the trained model! This effect was seen in:

- criminal sentencing: https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing
- predictive policing: https://www.technologyreview.com/2020/07/17/1005396/predictive-policing-algorithms-racist-dismantled-machine-learning-bias-criminal-justice/
- resume filtering: https://www.reuters.com/article/us-amazon-com-jobs-automation-insight-idUSKCN1MK08G