Neural Sequence Modeling

Part 1: Neural Networks and Embeddings

Aaron Mueller
CAS CS 505: Introduction to Natural Language Processing
Spring 2026
Boston University

Admin

- **HW0** is due **tonight, Feb. 3** at 11:59pm!

- **HW1** has been released! It will be due on **Feb. 19**, in just over 2 weeks.
  - We will have a homework 1 help session on Feb. 17 at your lab section! (Note: this is a Tuesday, but BU will be operating on a Monday schedule.)
  - We are also still available at office hours.

- Thursday's (2/5) lecture will be pre-recorded—no in-person class. I will take questions on Piazza, in class the following Tuesday, and at office hours.

Overview of Concepts

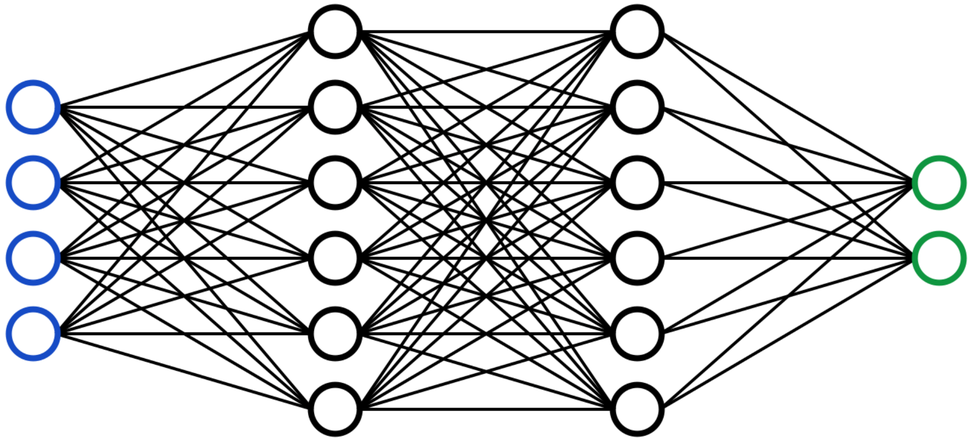
Perceptrons are small linear classifiers.

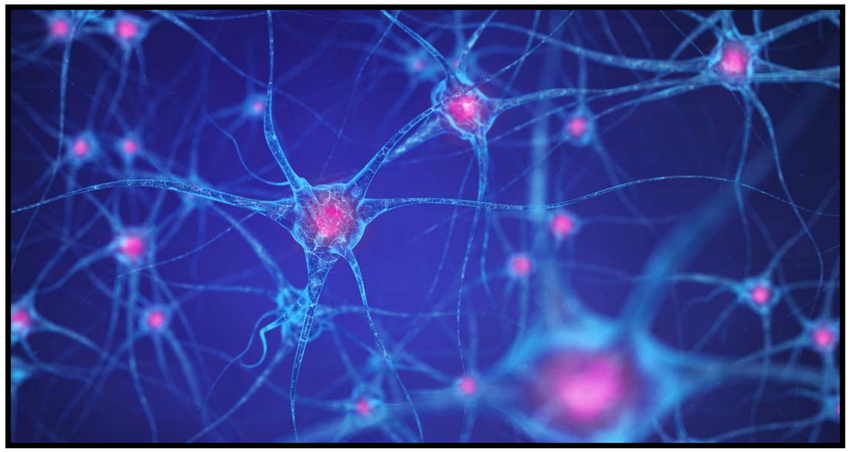
Neural networks are stacks of layers of perceptrons.

Backpropagation is how we compute the gradients of neural network parameters.

Embeddings are *dense* vector representations of tokens.

Skip-grams are embeddings learned via a context classifier.





Problems with n-grams

Sparsity: What if "like to eat w_j" never appeared in the training set?

$$p(w_j \mid \text{like to eat}) = \frac{C(\text{like to eat } w_j)}{C(\text{like to eat})}$$

Storage: Need to store counts of all possible n-grams. Requires $O(\exp(n))$ memory!

Independence: No sharing of information across similar prefixes.

Why neural networks?

- Statistical approaches like n-grams are easy to understand and implement. But they can require a lot of human effort and don't generalize well.
- Logistic regression works well, but it requires us to engineer our own features.

- These days, **neural networks** are all the rage. Here are some reasons why:
  - They learn the "features" by themselves.
  - We have architectures that are great at handling large quantities of data.
  - They generalize well to many tasks when we have a ton of data.

Perceptrons

Recall logistic regression:

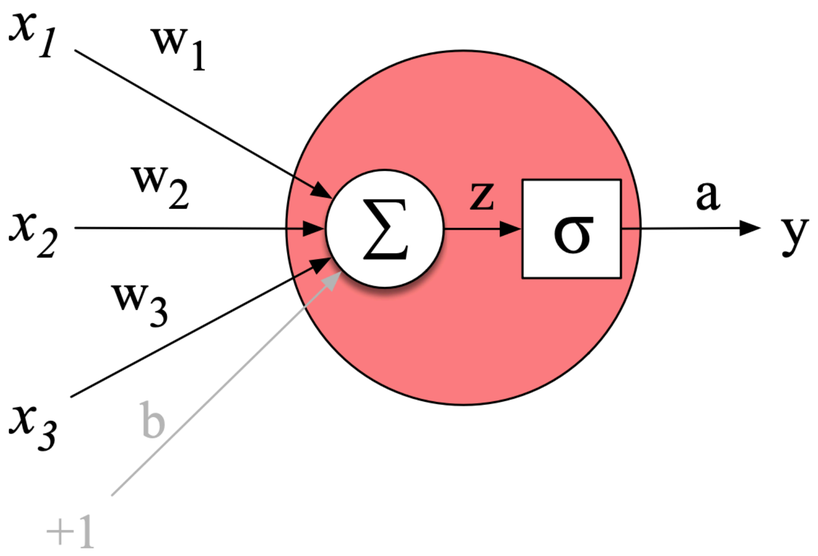
Outputs a probability

$$\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

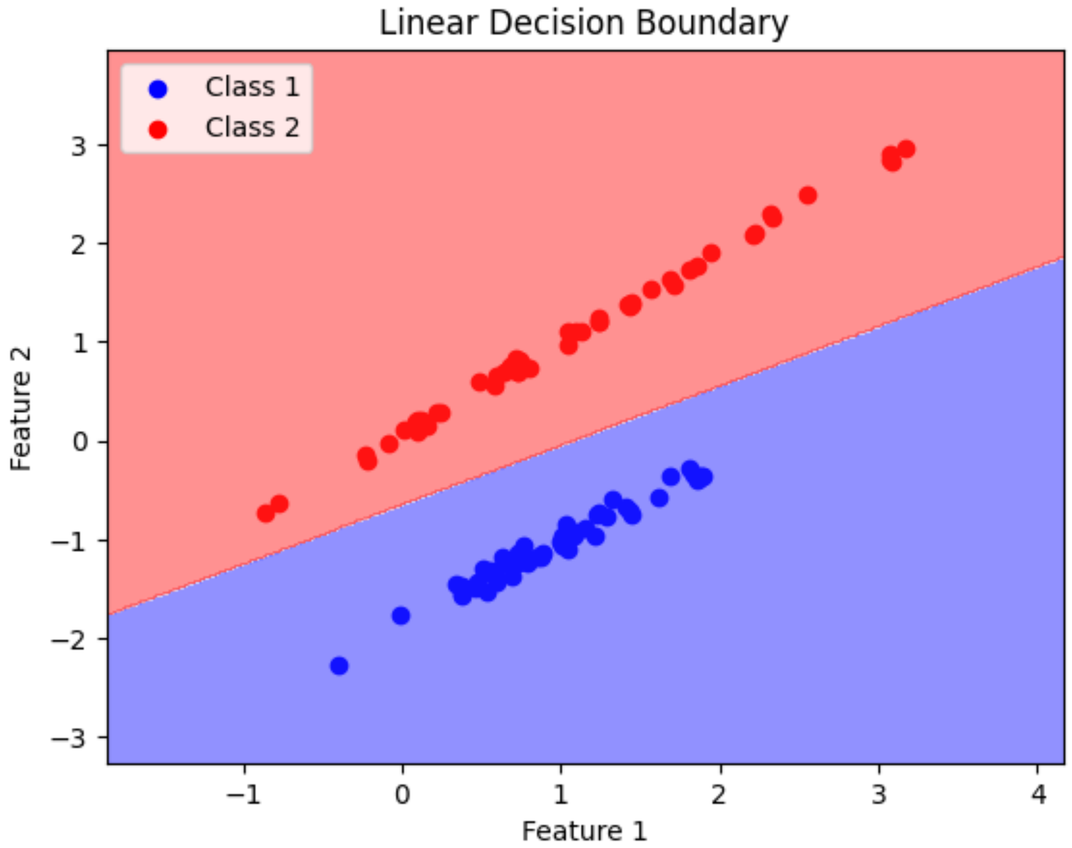
This is similar to the foundational unit of neural networks: a **perceptron**.

$$\hat{y} = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \end{cases}$$

Discrete; hard step function



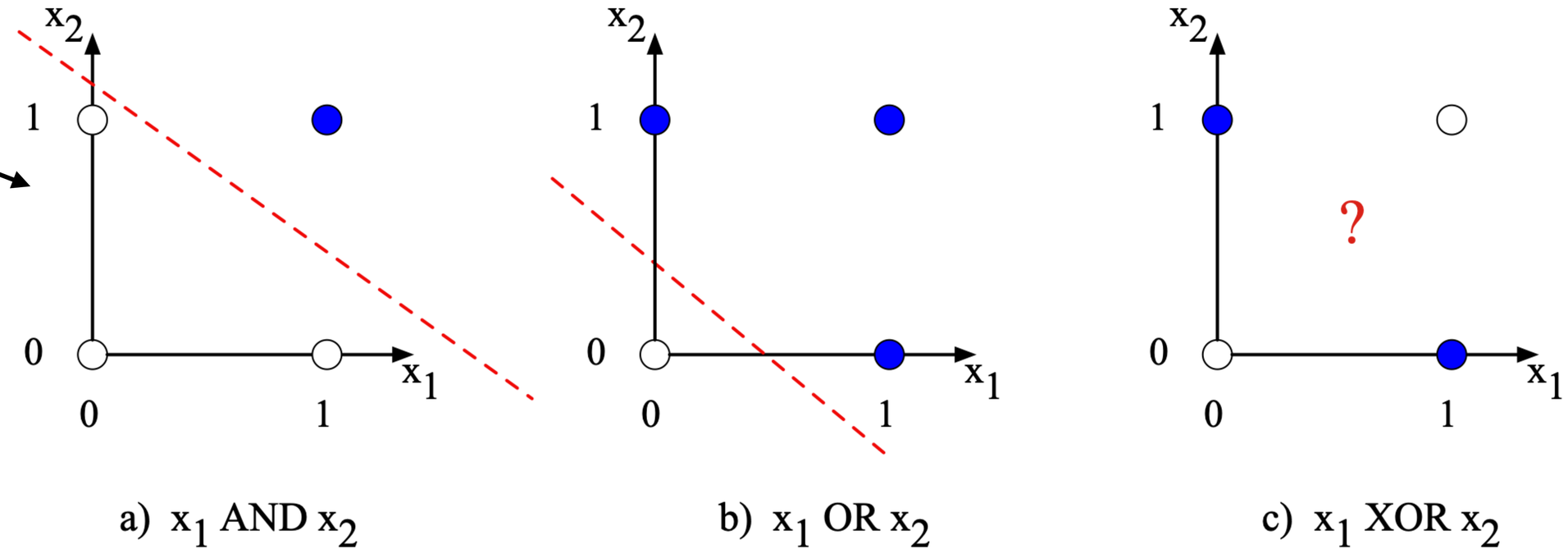
Decision Boundaries



If we plot the feature vector, we can show exactly where the perceptron draws the line between classes.

The Weakness of Perceptrons

XOR function



a) x_1 AND x_2 b) x_1 OR x_2 c) x_1 XOR x_2

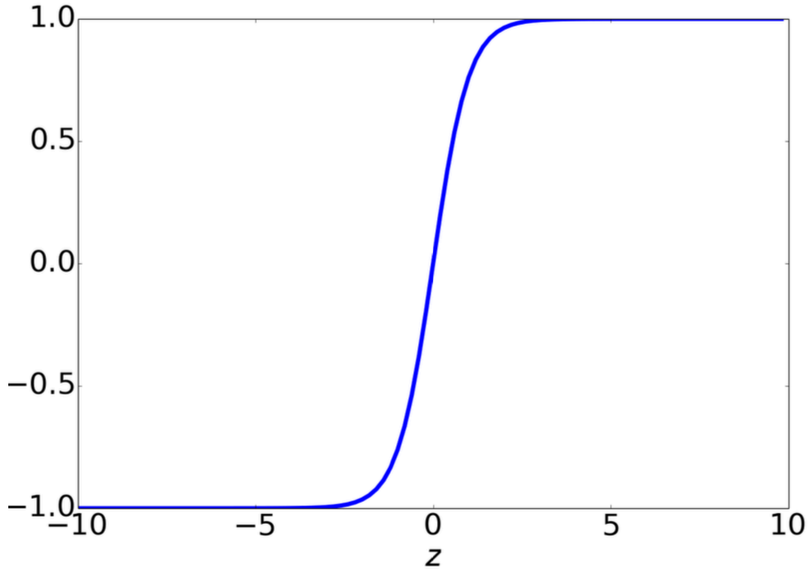
- Perceptrons just draw a line through the feature space between classes.
- But what if we can't draw a clean line between them?
- This is a *fundamental weakness* of perceptrons: they can't handle situations like this.

Non-linearities

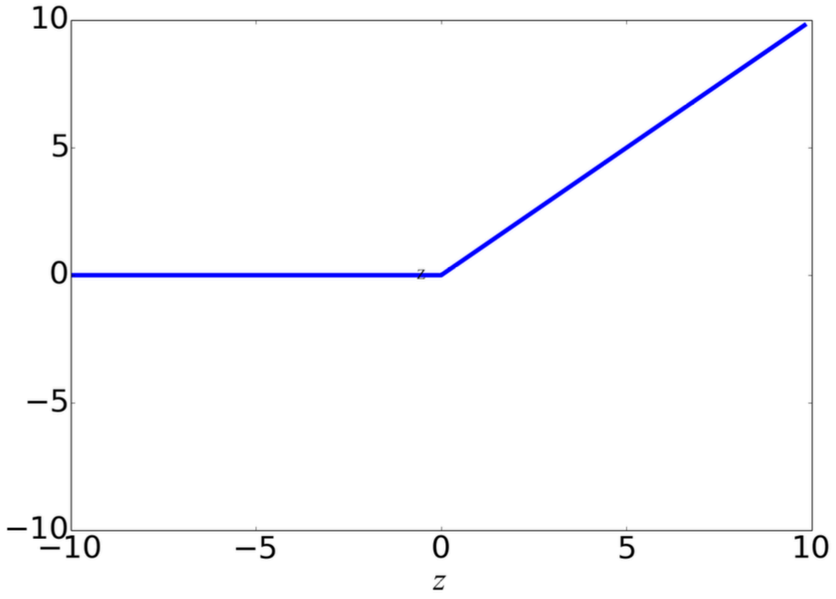
$$\hat{y} = \sigma(\mathbf{w}\mathbf{x} + b)$$

- Non-linearities are **essential**. We apply them after every layer of perceptrons.
- A sigmoid is one kind of non-linearity. In practice, they aren't super common. More common and better functions include:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



$$\text{ReLU}(z) = \max(0, z)$$



1. Why do we need non-linearities?

2. Why are these better than the sigmoid?

We'll come back to these questions.

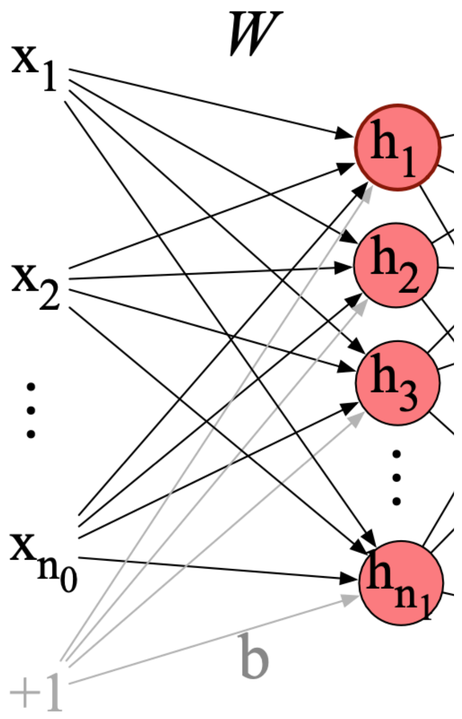
Stacking Perceptrons

We can use multiple perceptrons in parallel, each with their own weights:

$$h_1 = \sigma(\mathbf{w}_1\mathbf{x} + b_1)$$
$$h_2 = \sigma(\mathbf{w}_2\mathbf{h} + b_2)$$
$$\dots$$

We'll refer to the vector of concatenated h's as $\mathbf{h}$.

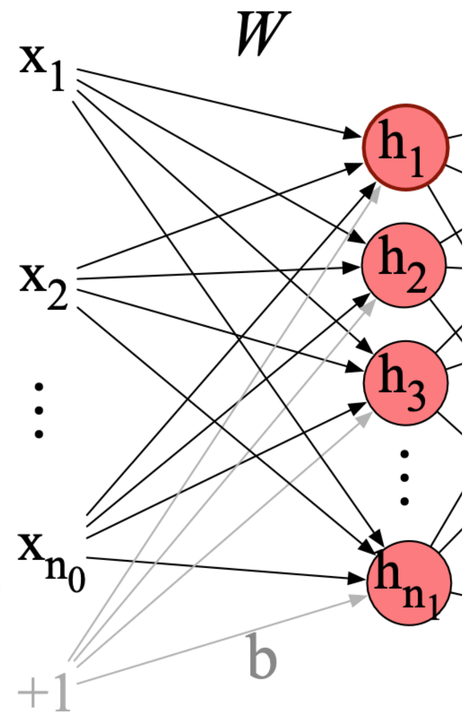
We'll refer to the stacked $\mathbf{w}$'s as the weight matrix W.



Stacking Perceptrons

We can use multiple perceptrons in parallel, each with their own weights:

$$\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$$



We'll call this a **layer** of perceptrons.

...

We'll refer to the vector of concatenated h's as $\mathbf{h}$.

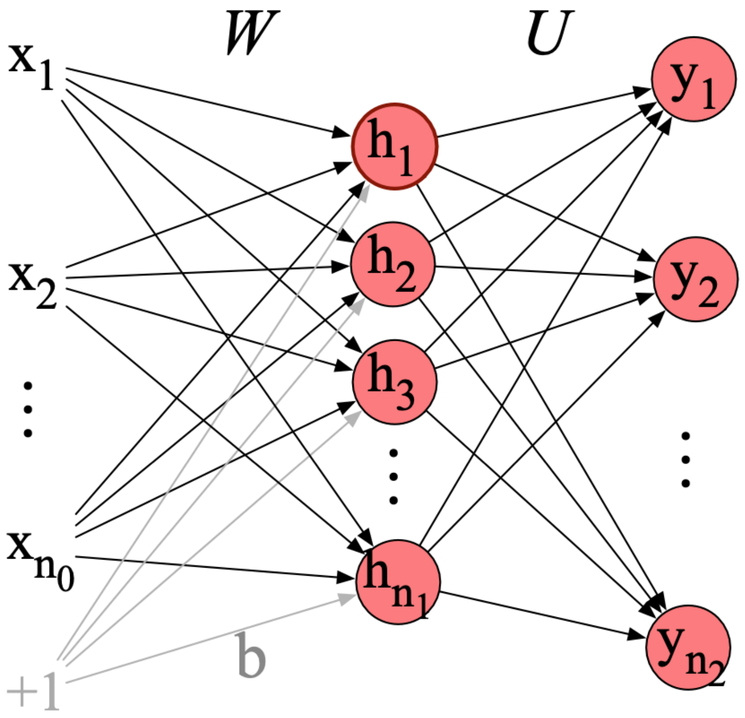
We'll refer to the stacked $\mathbf{w}$'s as the weight matrix W.

Stacking Perceptrons

We can use multiple perceptrons in parallel, each with their own weights:

$$\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$$

This is the **hidden layer**. We do not directly see it, because it is not the input nor the output; it is *hidden* inside the network of perceptrons.



We can then apply another layer of perceptrons on top of $\mathbf{h}$!

$$\mathbf{z} = U\mathbf{h}$$

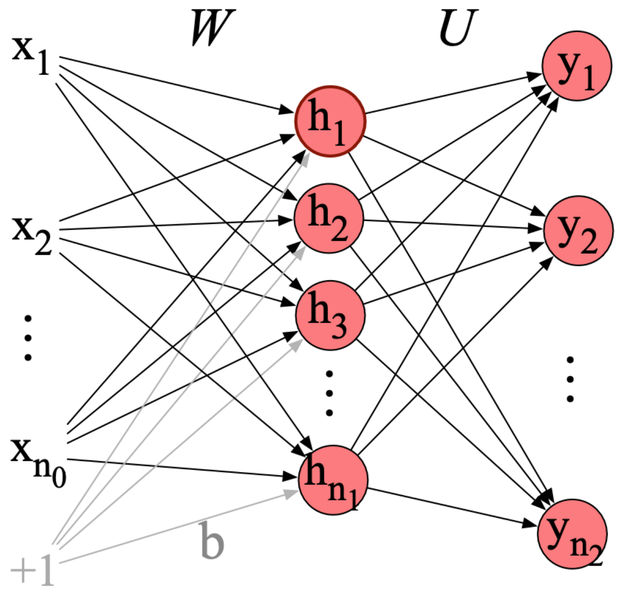
This is the **output layer**.

By analogy, $\mathbf{x}$ is the **input layer**.

Return of the Softmax

Recall the softmax: $\text{softmax}(z_i) = \dfrac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}$

We needed this to turn unnormalized logits into probabilities. We'll apply this to the output layer to get probabilities over outputs.



Thus, the full definition of our multi-layer neural network is:

$$\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$$

$$\mathbf{z} = U\mathbf{h}$$

$$\mathbf{y} = \text{softmax}(\mathbf{z})$$

Why non-linearities?

Imagine we did not have any non-linearities:

$$\mathbf{h} = W\mathbf{x} + \mathbf{b}_1$$

$$\mathbf{y} = U\mathbf{h} + \mathbf{b}_2$$

Thus, any *n*-layer *linear* neural network has equal expressive power to a 1-layer neural network.

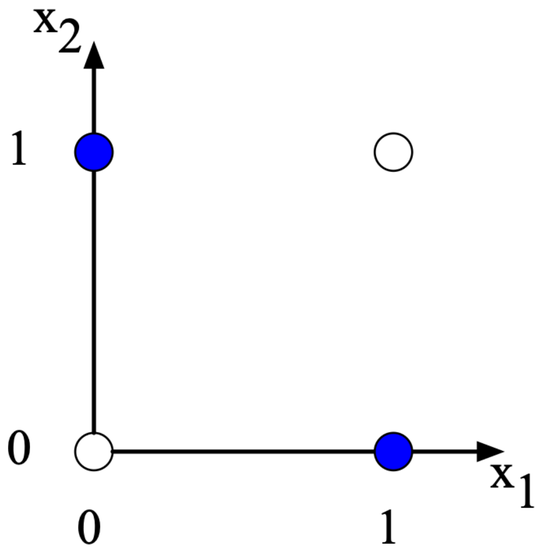
We can rewrite this as:

$$\begin{aligned}\mathbf{y} &= U(W\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \\ &= UW\mathbf{x} + U\mathbf{b}_1 + \mathbf{b}_2 \\ &= W'\mathbf{x} + \mathbf{b}'\end{aligned}$$

This is *not* true if we include non-linearities: expressive power increases with >1 layer!

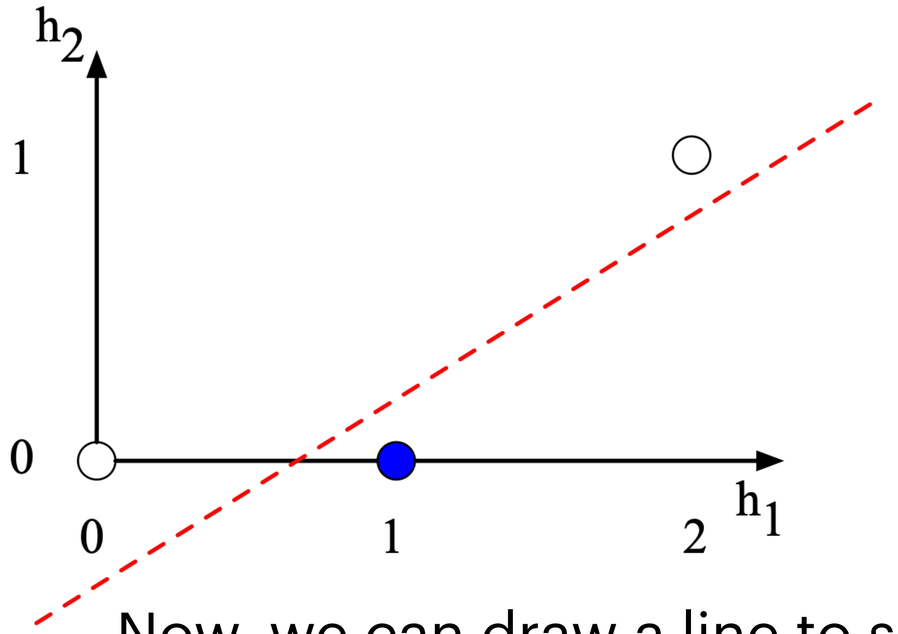
Why non-linearities?

$\mathbf{x}$ before non-linearity:



Can’t draw a line to separate classes.

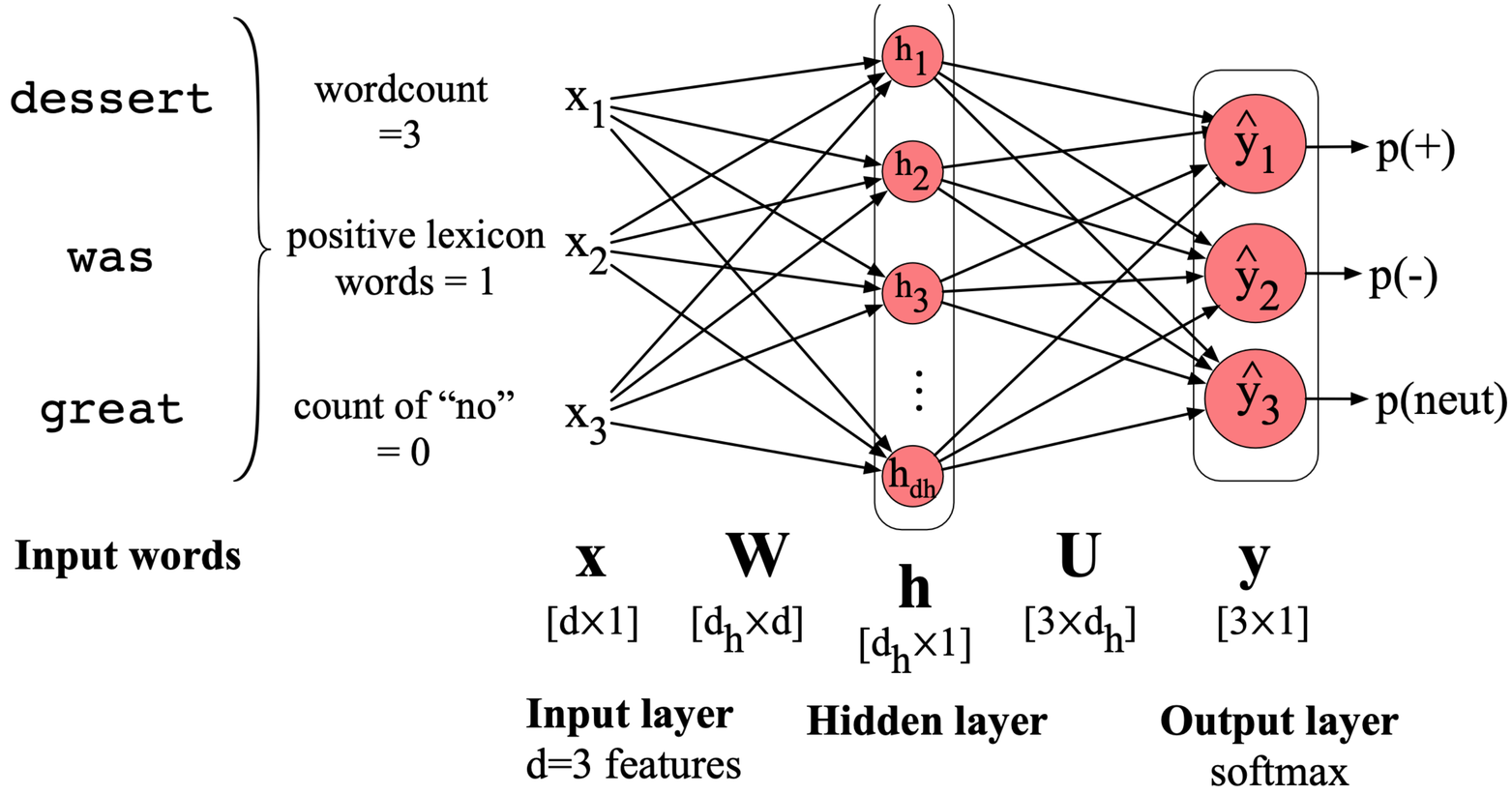
$\mathbf{h}$ (after a non-linearity):



Now, we can draw a line to separate them!

Text Classification with MLPs

Let's return to the sentiment classification task using hand-crafted features:



Training a Neural Network

Recall how we trained our logistic regression classifiers with **gradient descent**. We train neural networks in an intuitively similar way:

1. Make a prediction $\hat{y}$ using the neural network.
2. Use the **cross-entropy loss** to compare the prediction to the correct answer y.
3. Compute the **gradient** of the weights, and use these to update the weights.

$$\frac{\partial L_{\text{CE}}}{\partial w_j} = -(y - \hat{y})x_j$$

This only works for the *last* layer!

How do we compute the gradients for earlier layers?

Backpropagation

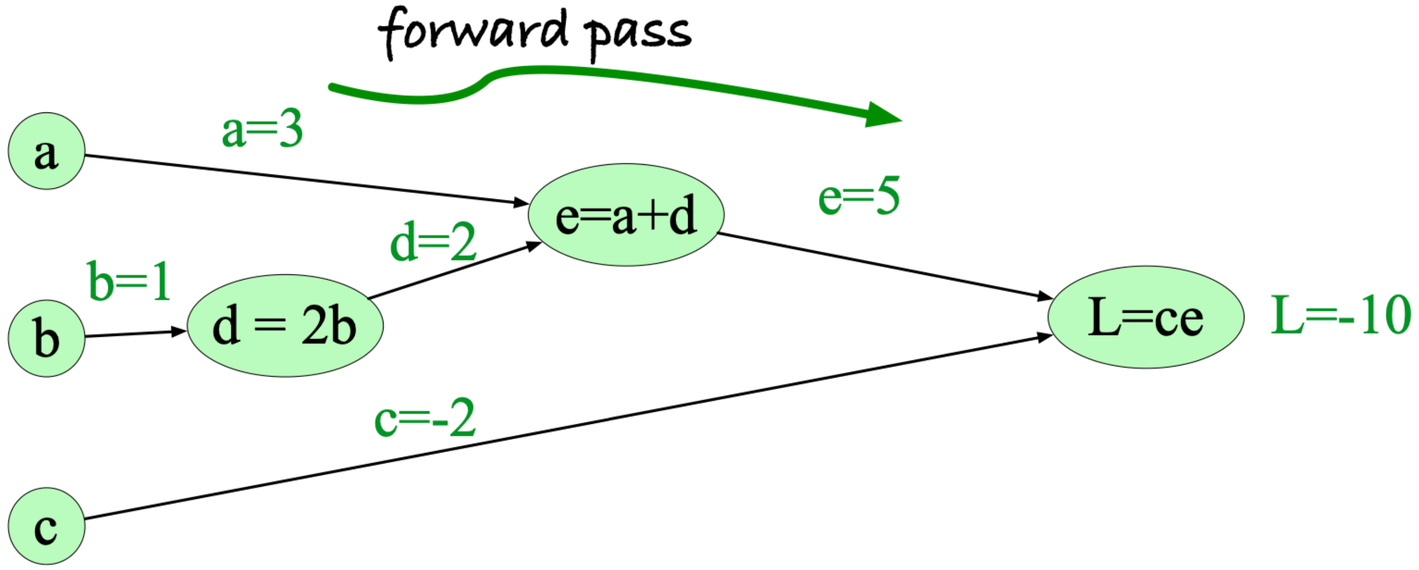
Forward Pass

- To update the weights, we must do a **forward pass** and a **backward pass**.
- The forward pass just means getting a prediction from the network given the inputs:

$$d = 2b$$

$$e = a + d$$

$$L = ce$$



This structure is called a **computation graph.**

Backpropagation

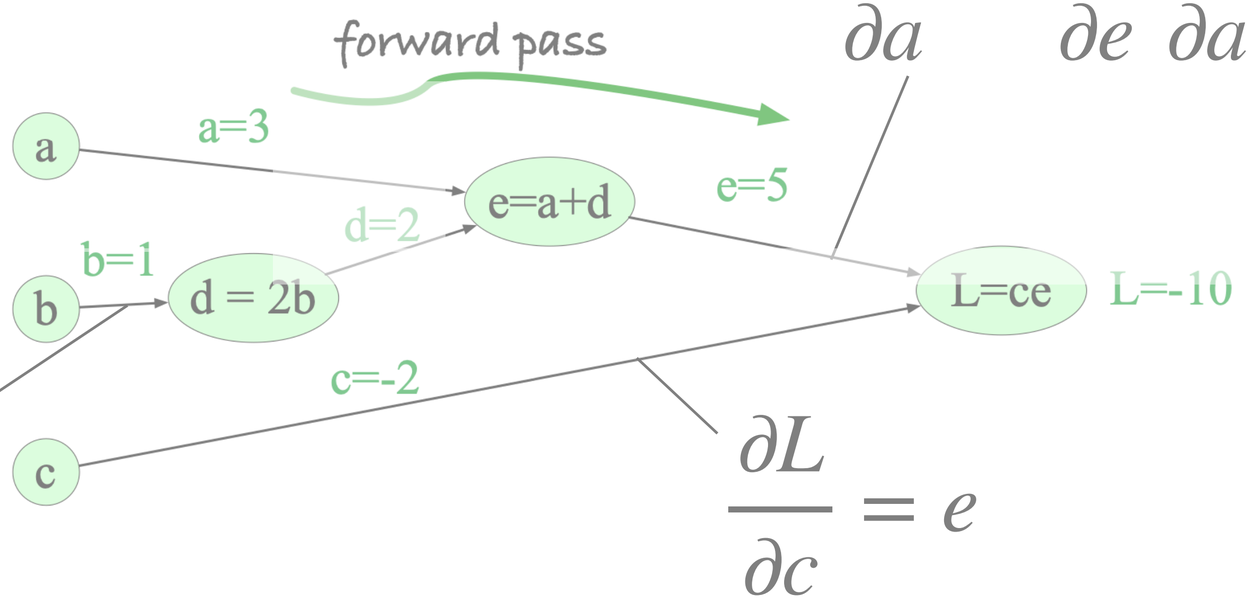
Backward Pass

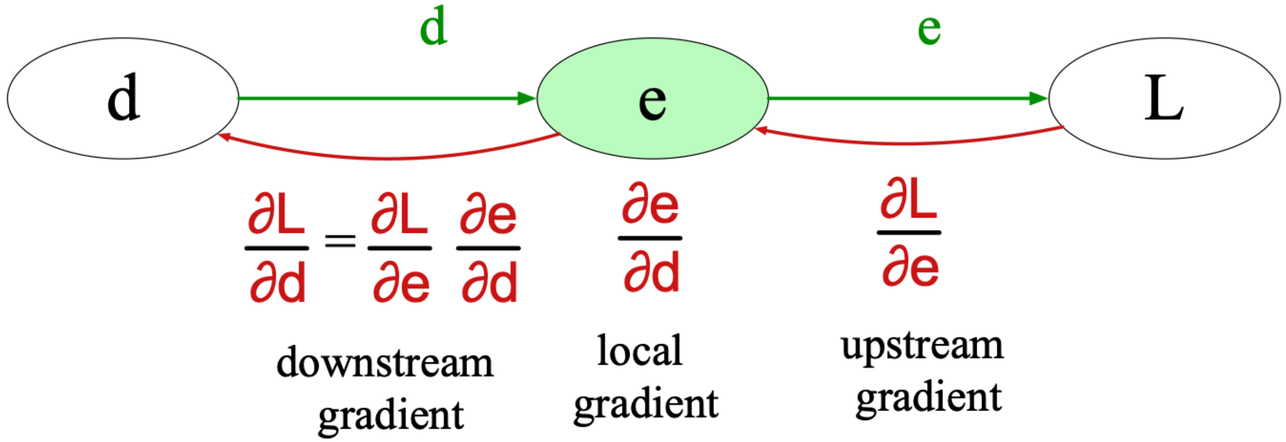
- We compute the gradients using the **chain rule of derivatives.**

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

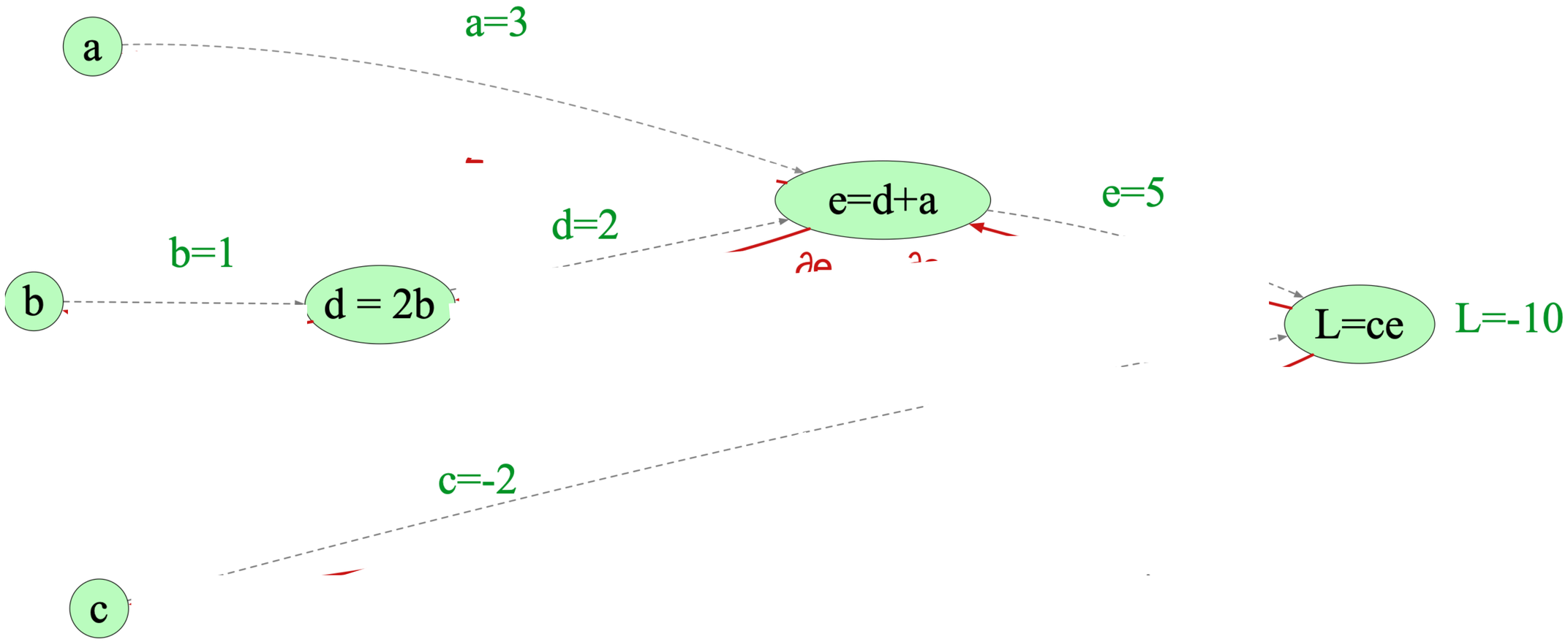
$$\frac{\partial L}{\partial c} = e$$





Backpropagation

Example



Backpropagation in a Neural Network

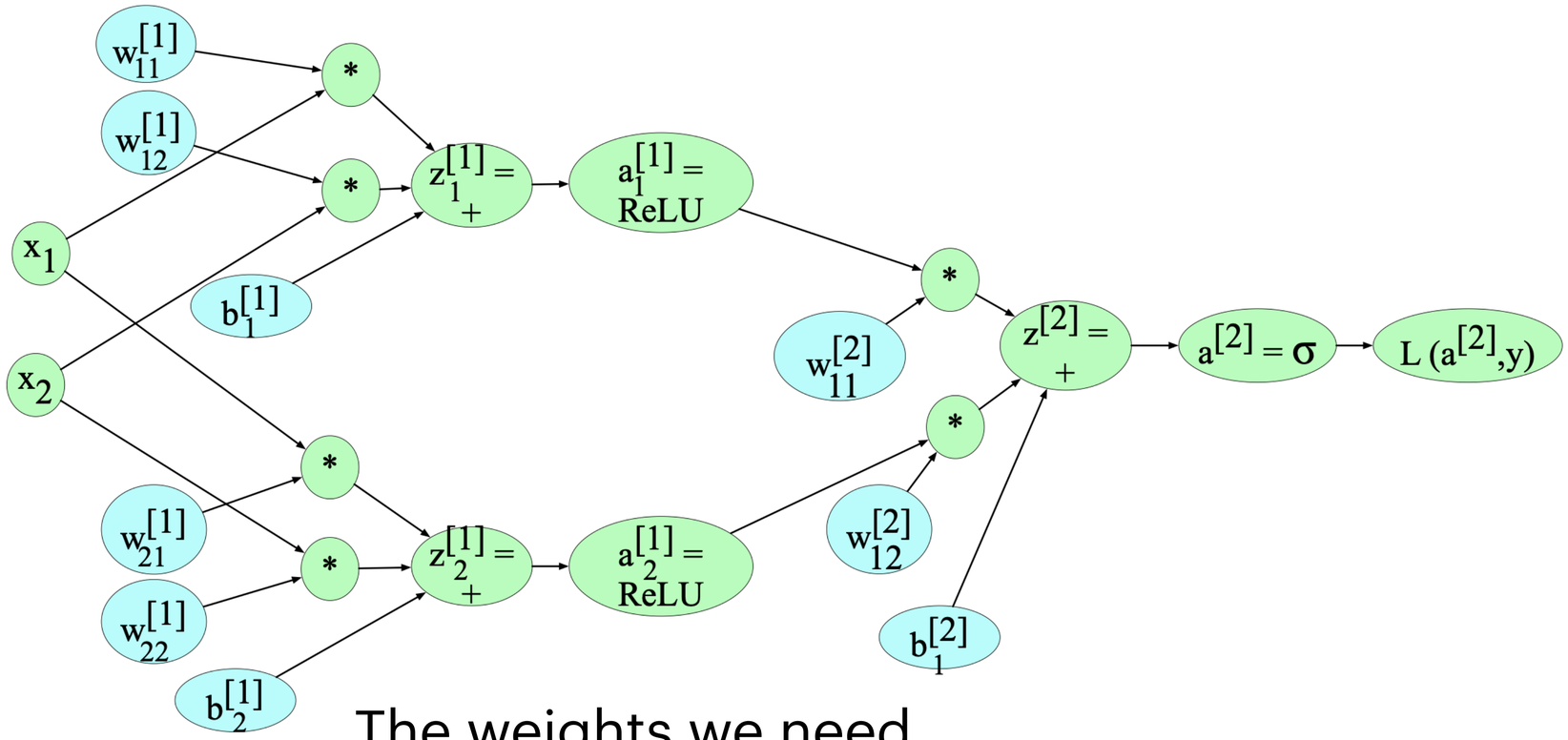
Forward Pass

Forward pass:

$$\mathbf{h} = \text{ReLU}(W_1\mathbf{x} + \mathbf{b}_1)$$

$$z = W_2\mathbf{h} + b_2$$

$$y = \sigma(z)$$



The weights we need to update are shown in **teal**.

Backpropagation in a Neural Network

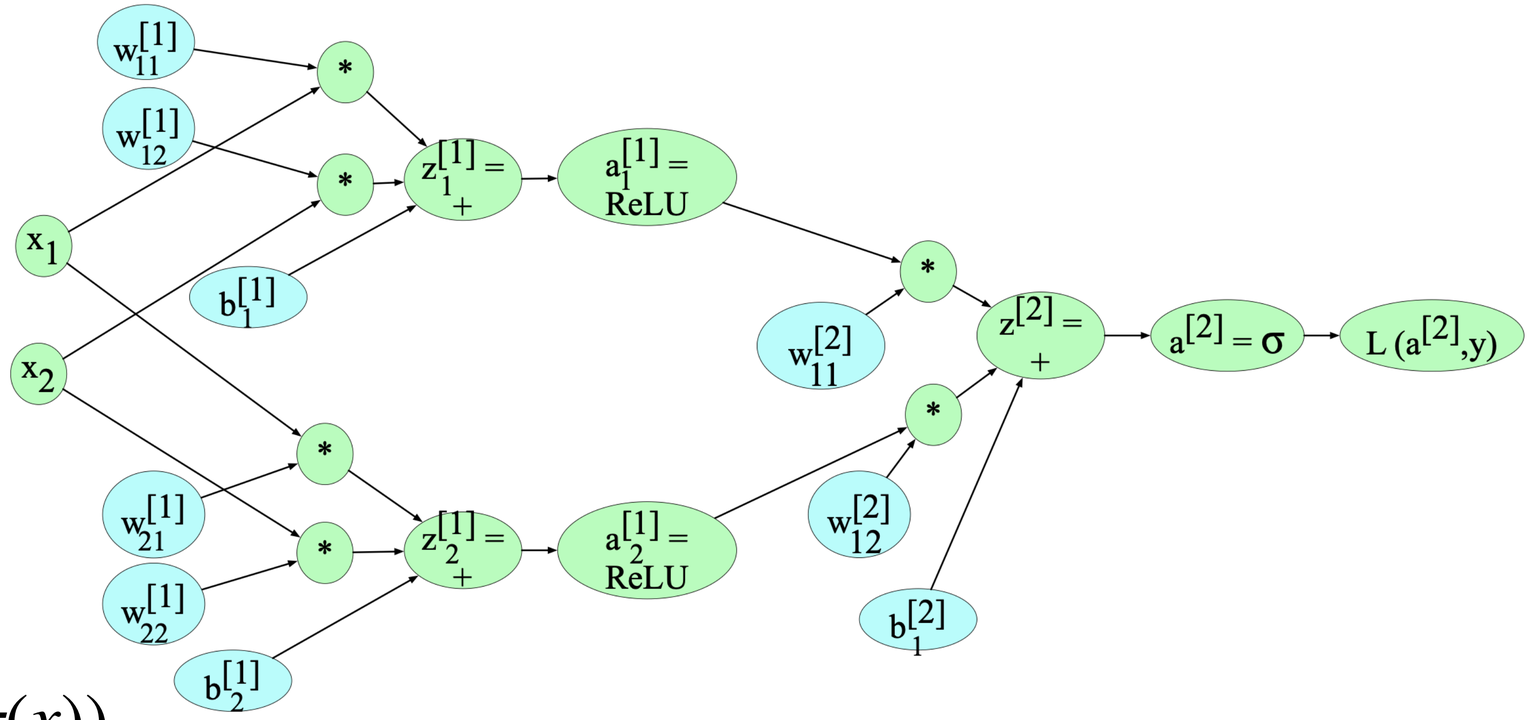
Backward Pass

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}$$

$$L_{CE} = -\left[y \log a + (1-y)\log(1-a)\right]$$

$$\frac{\partial L}{\partial a} = -\left(\left(y\frac{\partial \log a}{\partial a}\right) + (1-y)\left(\frac{\partial \log(1-a)}{\partial a}\right)\right)$$

$$\frac{\partial L}{\partial a} = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)$$



The derivative of $\sigma(x)$ is $\sigma(x)(1-\sigma(x))$.

$$\frac{\partial a}{\partial z} = a(1-a)$$

We repeat this, taking all partial derivatives for each edge in this graph, until we have the gradients for all **teal** nodes w.r.t. L.

Derivatives for Common Non-linear Functions

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$$

$$\frac{d\tanh}{dx} = 1 - \tanh^2(x)$$

$$\frac{d\text{ReLU}}{dx} = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

Think about what will happen if you keep multiplying these functions by themselves over and over, like you would in a neural network.

Vanishing Gradients

Why are sigmoids considered not as good as ReLUs?

Imagine you stack a bunch of sigmoids in a multi-layer perceptron:

$$\mathbf{h}^{(1)} = \sigma(W^{(1)}\mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{h}^{(2)} = \sigma(W^{(2)}\mathbf{h}^{(1)} + \mathbf{b}_2)$$

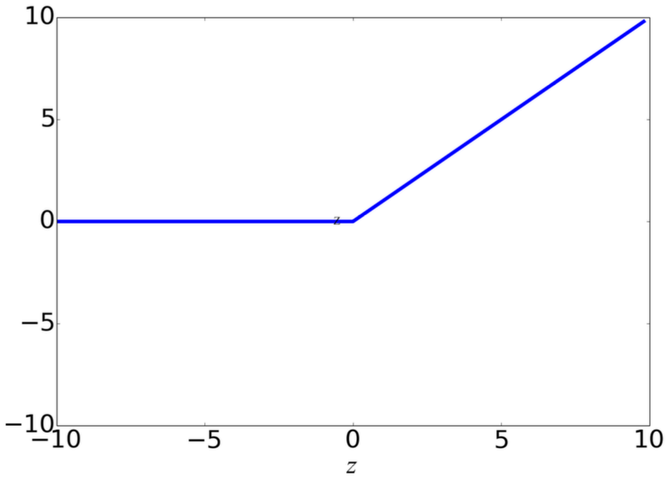
$$\mathbf{y} = \text{softmax}(U\mathbf{h}^{(2)})$$

The gradient for $\mathbf{h}^{(1)}$ requires multiplying the derivatives of sigmoids to each other.

- These derivatives are usually small, so their products get smaller and smaller with more and more layers, until the gradient just... becomes 0. Learning stops.

Vanishing Gradients

$\text{ReLU}(z) = \max(0, z)$



Imagine instead we had used tanh or ReLU:

$$\mathbf{h}^{(1)} = \text{ReLU}(W^{(1)}\mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{h}^{(2)} = \text{ReLU}(W^{(2)}\mathbf{h}^{(1)} + \mathbf{b}_2)$$

$$\mathbf{y} = \text{softmax}(U\mathbf{h}^{(2)})$$

The gradient for $\mathbf{h}^{(1)}$ requires multiplying the derivatives of sigmoids to each other.

- These derivatives will always be 0 or 1!

Practical Considerations

- Neural networks have a lot of **hyperparameters** to tune, like learning rates and number of training steps. Tuning these is essential for good performance.
    - High LR/Low training steps: the network never converges to a good solution (underfitting), or even diverges
    - Low LR: the network takes too long to converge

- As networks get more powerful, we need to worry more about preventing overfitting.
    - You will often encounter **regularization** methods that help prevent overfitting.

Implementation Details

PyTorch makes doing backpropagation super easy.

```python
class SimpleNN(nn.Module):
    def __init__(self, input_size, num_classes):
        super(SimpleNN, self).__init__()
        # Fully connected layers
        self.fc1 = nn.Linear(input_size, 128)
        self.relu = nn.ReLU() # Activation function
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

```python
model = SimpleNN()

# 2. Define Loss function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Sample dummy data (input X, target y)
inputs = torch.randn(1, 10)
targets = torch.tensor([1])

# 1. Forward pass: compute predicted y
outputs = model(inputs)

# 2. Compute loss
loss = criterion(outputs, targets)
print(f'Loss: {loss.item()}')

# 3. Backward pass: compute gradients
loss.backward()

# 4. Optimizer step: update weights
optimizer.step()
```

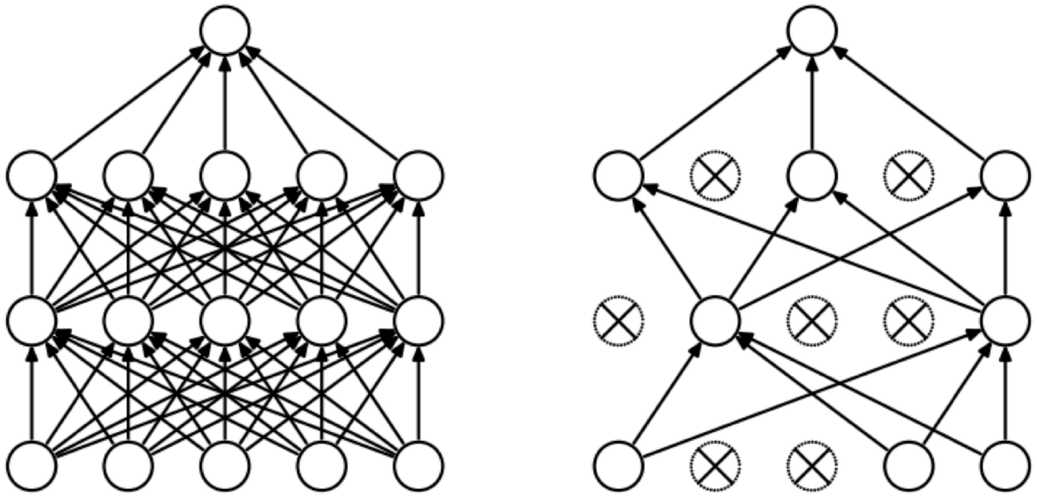
Dropout

- With some probability, we can zero out parts of the network during training to prevent overfitting.
- (Use full network at test time.)

- One line in PyTorch:

```
nn.Dropout(0.2)
```

"During every forward pass, randomly zero out a component with probability 0.2."



(a) Standard Neural Net

(b) After applying dropout.

Summary

- Neural networks are a bunch of perceptrons stacked next to and on top of each other, with non-linearities between layers.
    - Way more powerful than logistic regression or perceptrons on their own.

- Training neural networks requires **backpropagation**, which is just the chain rule of derivatives.
    - This is a way of computing the gradients in multi-layer neural networks.
    - Otherwise, we still use gradient descent, like before.

Embeddings and Word Meanings

Beyond Hand-crafted Features

- So far, we've manually computed feature vectors or hand-crafted some features for our models to use.
    - This is kind of annoying, and doesn't scale well to large datasets.

- Can we have the neural network learn representations of tokens by itself? What would these look like?

- Answer: We can! These token representations are called **embeddings.**

What does a word mean?

Odor (noun): A strong smell.

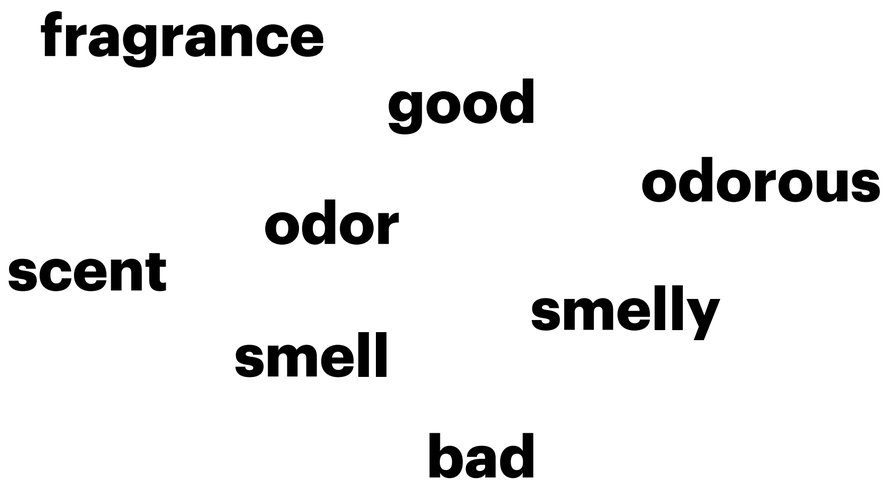
Part of speech

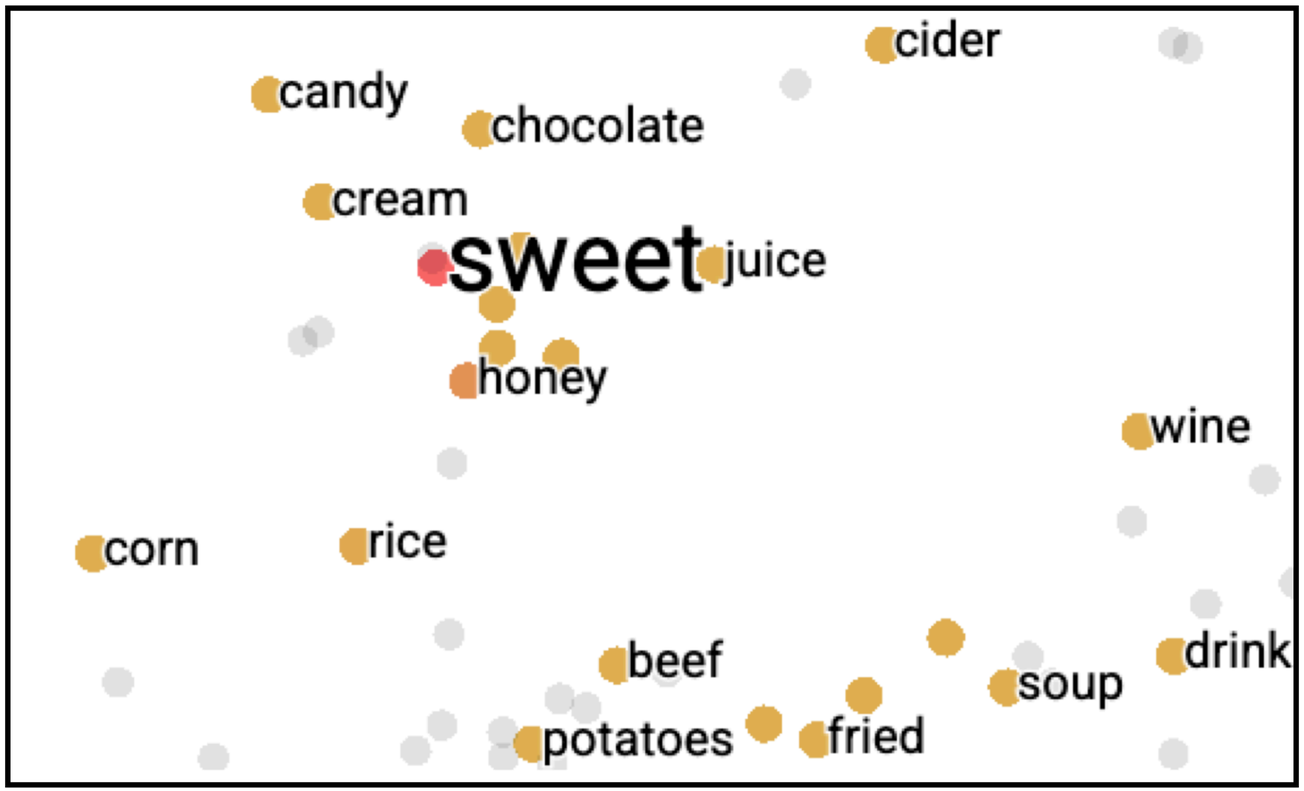
Definition

Usually means a *bad* smell.

Connotation

A word can often be defined in terms of other words:





We want our token representations to be such that more similar tokens are closer to each other.

The Distributional Hypothesis

- **The distributional hypothesis**: *"You shall know a word by the company it keeps."*
  - Similar words will appear in similar contexts
  - Thus, a word's **distribution** tells you a lot about its relationship to other words.

In my band, we have a guitarist, drummer, and *blorpfa*.

She was a *blorpfa* until last year, when she hurt her foot.

Do you know any *blorpfa*?

Dan Firth

Count-based Embeddings

is traditionally followed by	**cherry**	pie, a traditional dessert
often mixed, such as	**strawberry**	rhubarb pie. Apple pie
computer peripherals and personal	**digital**	assistants. These devices usually
a computer. This includes	**information**	available on the internet

This is our token representation for “cherry”

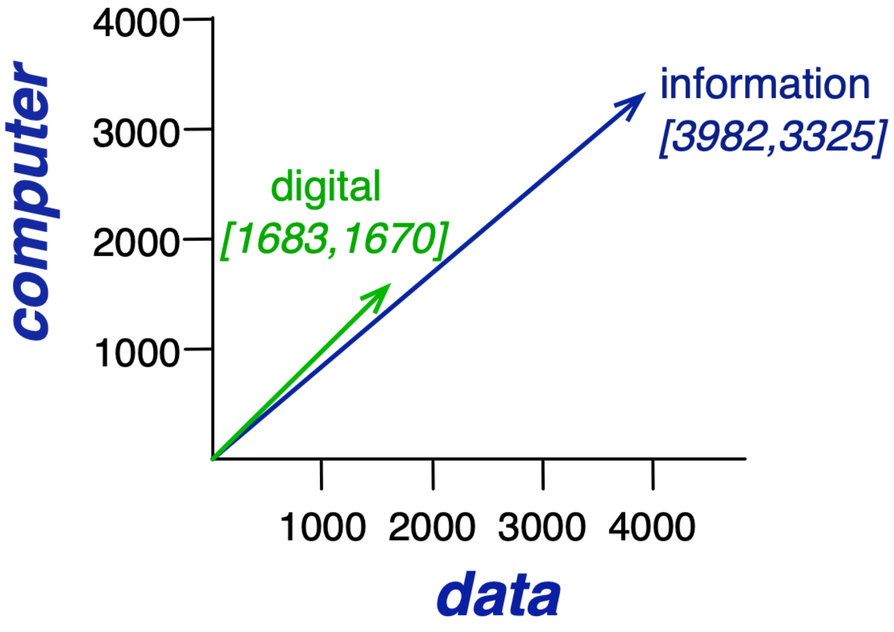
	a	**computer**	**pie**
cherry	1	0	1
strawberry	0	0	2
digital	0	1	0
information	1	1	0

Count-based Embeddings

Here's an example from Wikipedia:

	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...

More similar words
end up pointing
in similar directions!



Most of these numbers
will be zero, so this is
a **sparse** representation
of word meaning.

word2vec

- *Idea*: instead of counting co-occurrences, we'll train a classifier that computes how likely word c will be to co-occur with another word.
- The task doesn't matter; the weights of this classifier will be our word representations.

Here, the context window is of size 2:

```
... lemon,  a [tablespoon of apricot jam,      a] pinch ...
              c1          c2    w     c3       c4
```

Word

Classifier computes: $p(+|w,c)$ Context

Probability that c will co-occur with w

$$p(+|w,c) = \sigma(\mathbf{w} \cdot \mathbf{c})$$

Learn $\mathbf{w}$ *and* $\mathbf{c}$ *s.t. the likelihood of the training data is maximized.*

Learning word2vec

Data

positive examples +

w	c_{pos}
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

w	c_{neg}	w	c_{neg}
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

Non-co-occurring words, randomly sampled from lexicon

- This way of training embeddings is called **skip-gram with negative sampling** (SGNS).
  - The "negative sampling" part refers to the above.
- We sample k negative words per positive word. Here, $k = 2$.

Learning word2vec

Loss

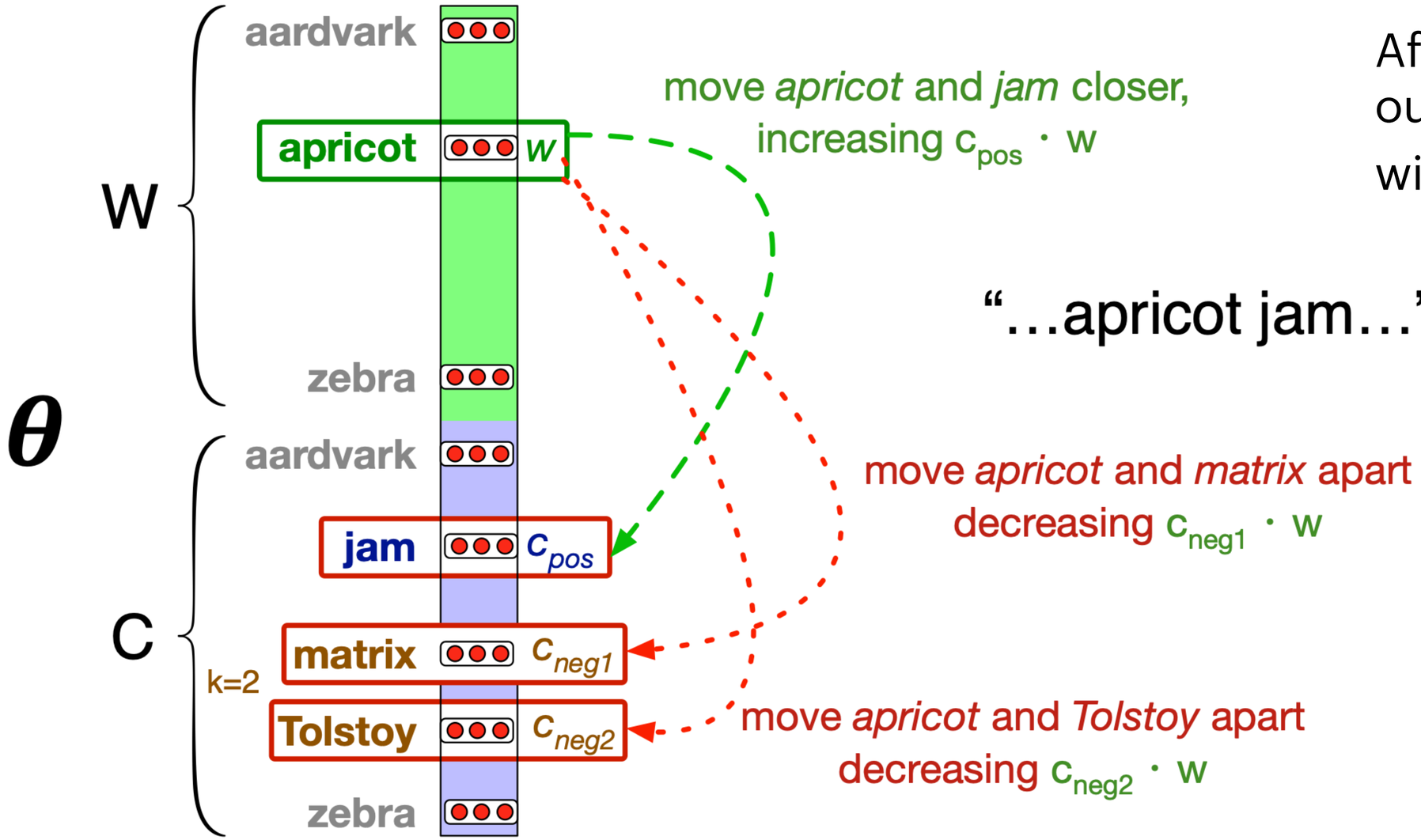
k : Number of negative examples per positive example

$$L(w, c_+, c_-) = -\log\left[p(+|w, c_+)\prod_{i=1}^{k} p(-|w, c_-)\right]$$

$$= -\left[\log p(+|w, c_+) + \sum_{i=1}^{k} p(-|w, c_-)\right]$$

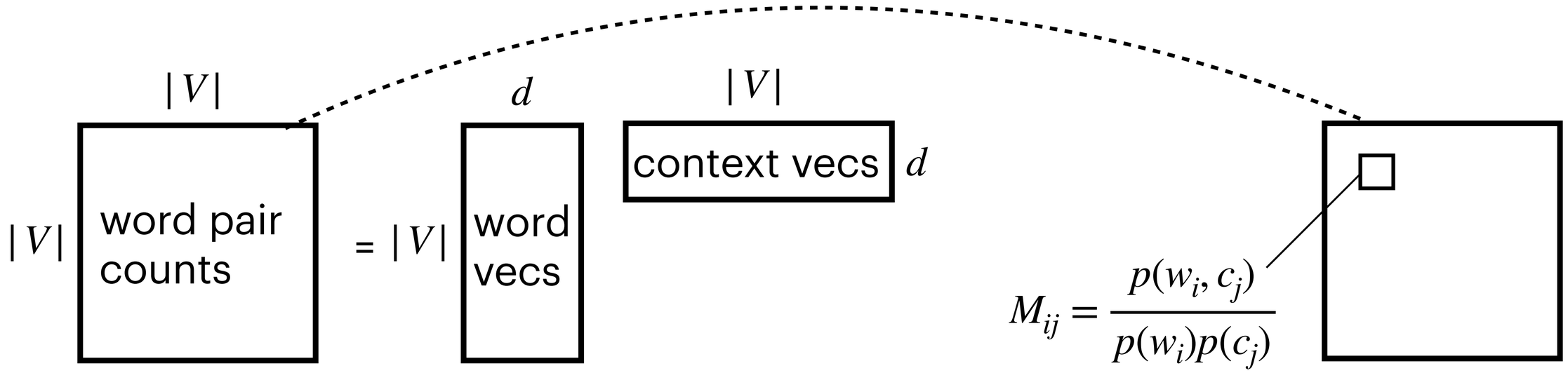
$$= -\left[\log \sigma(c_+ \cdot w) + \sum_{i=1}^{k} \log \sigma(-c_- \cdot w)\right]$$

So basically, we want to maximize the dot product of words with context words they occur with, and minimize the dot product of words with context words they do *not* occur with.



After learning,
our word embedding
will be $\mathbf{w}_i + \mathbf{c}_i$.

Connections to Matrix Factorization



- Skip-gram objective corresponds nearly exactly to factoring this matrix!
  - Only if we use negative sampling

Other Word Vectors

- **GloVe** (Global Vectors) is based on global corpus statistics
  - Based on probability ratios from the word-word co-occurrence matrix

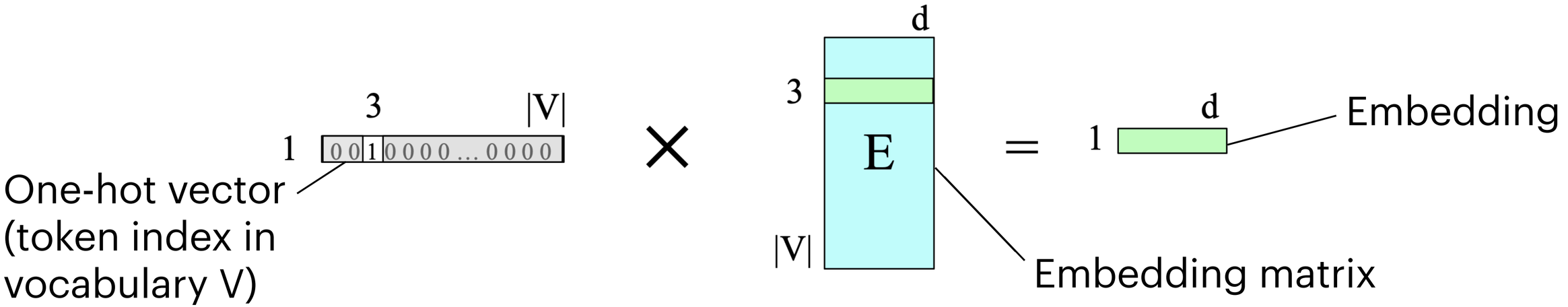
- **fasttext** is an extension of word2vec that better handles unknown words and sparsity
  - Uses subword modeling. E.g., “where” is represented as <where>, plus

    <wh, whe, her, ere, re>

  - Where we have an embedding for each of the above, and “where”’s embedding is the sum of these subword embeddings.

Using Embeddings in Neural Networks

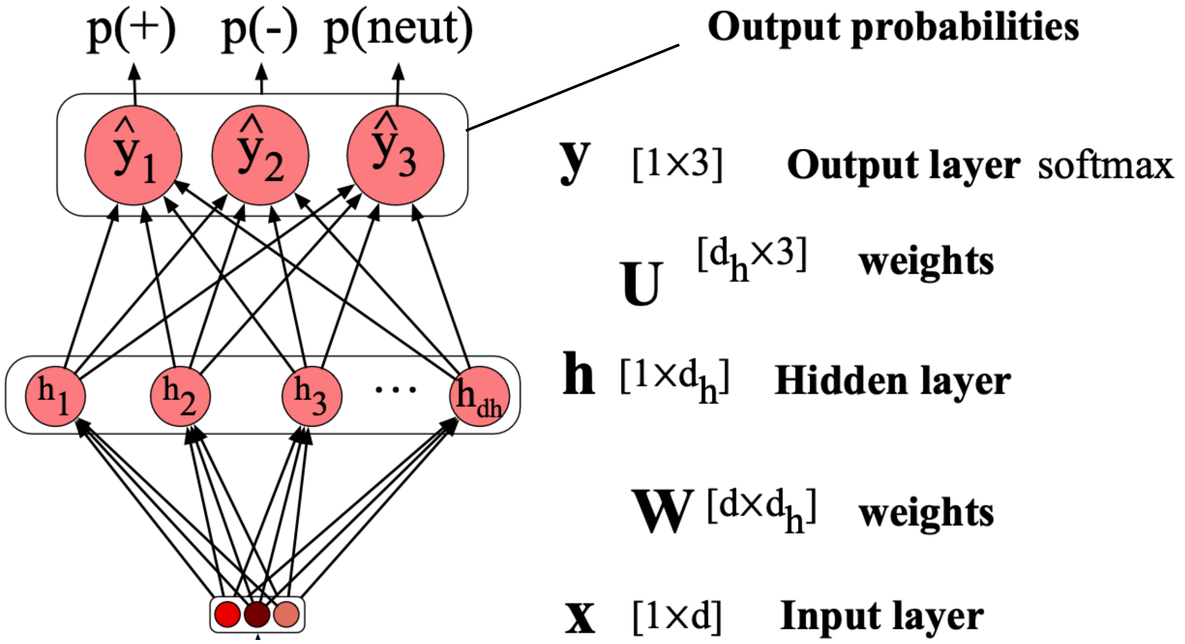
Given our pre-trained word embedding matrix, we can retrieve each embedding using a series of one-hot vector multiplications:



Where E usually comes from word2vec, GloVe, or some pre-trained embedding.

- You can also learn E alongside the rest of the neural network. *Tends to require a lot of data.*

Back to sentiment classification, but without hand-crafted features this time:



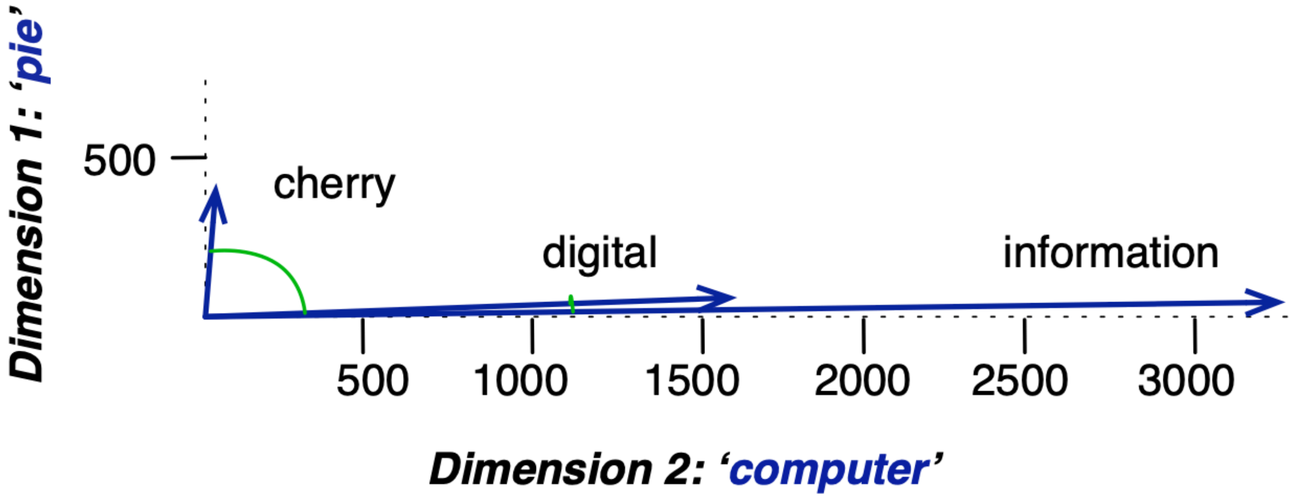
Evaluating Embeddings

- We want similar words to have similar embeddings to each other.

- How do we compute word similarity? We use the **cosine similarity:**

$$\cos(\mathbf{v}, \mathbf{w}) = \frac{\mathbf{v} \cdot \mathbf{w}}{|\mathbf{v}||\mathbf{w}|}$$

Basically the angle between the two word vectors.



[Levy et al., 2015]

Method	WordSim Similarity	WordSim Relatedness	Bruni et al. MEN	Radinsky et al. M. Turk	Luong et al. Rare Words	Hill et al. SimLex
PPMI	.755	**.697**	.745	.686	.462	.393
SVD	**.793**	.691	**.778**	.666	**.514**	.432
SGNS	**.793**	.685	.774	**.693**	.470	**.438**
GloVe	.725	.604	.729	.632	.403	.398

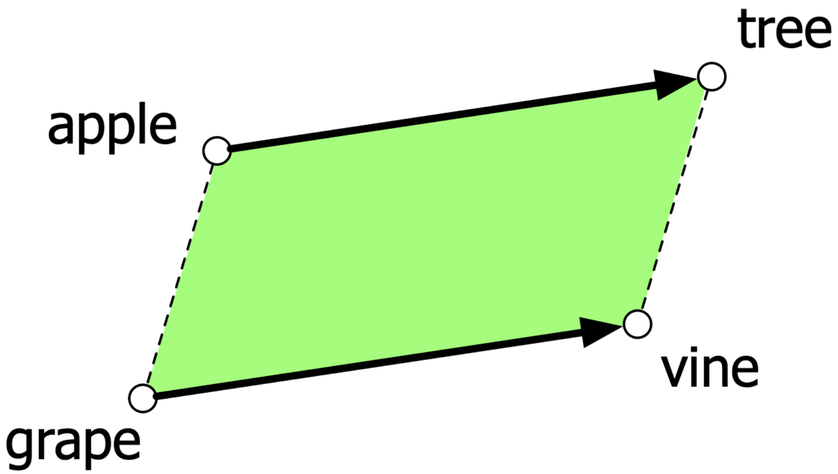
- GloVe is best in very controlled settings, but depends on the hyperparameters.
- (In practice, these distinctions are small enough to not matter much in reality.)

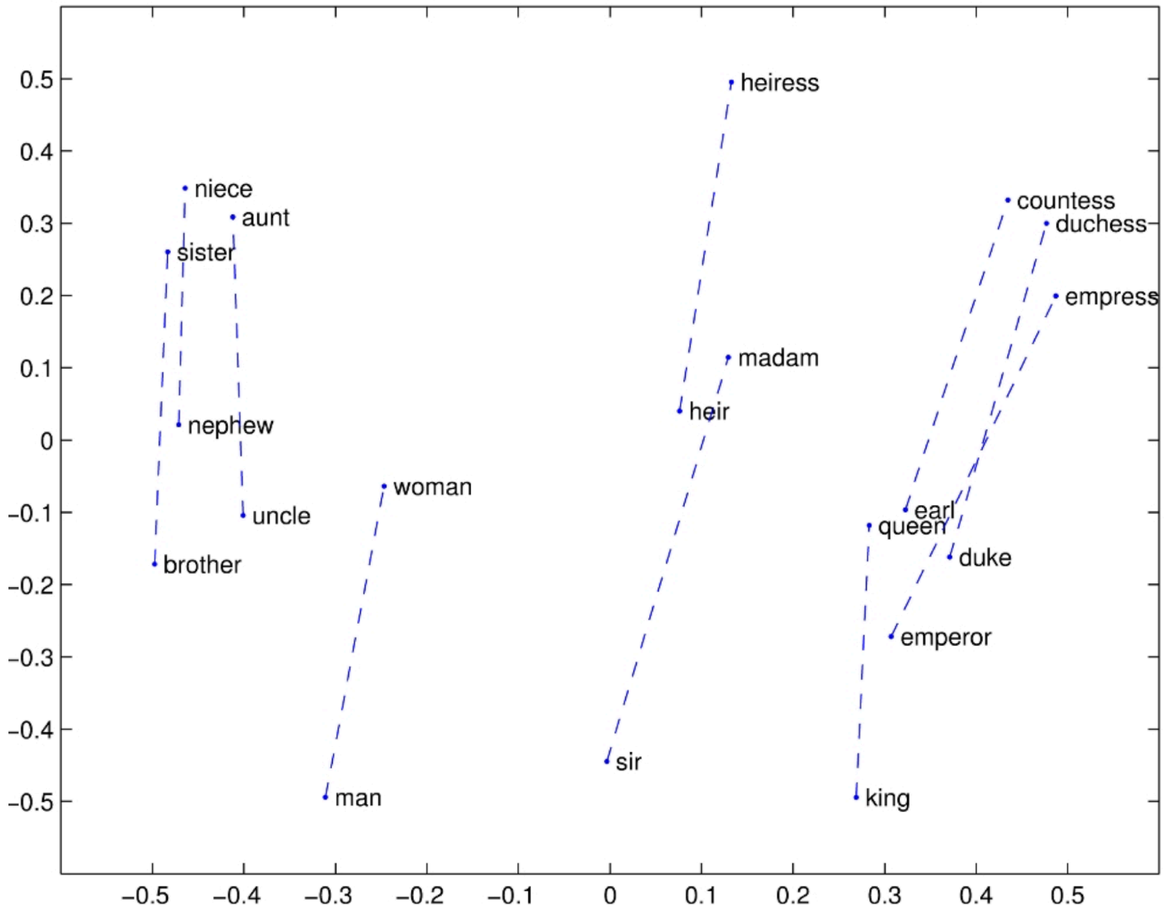
Word Vector Analogies

- Apple is to **tree** as grape is to _____

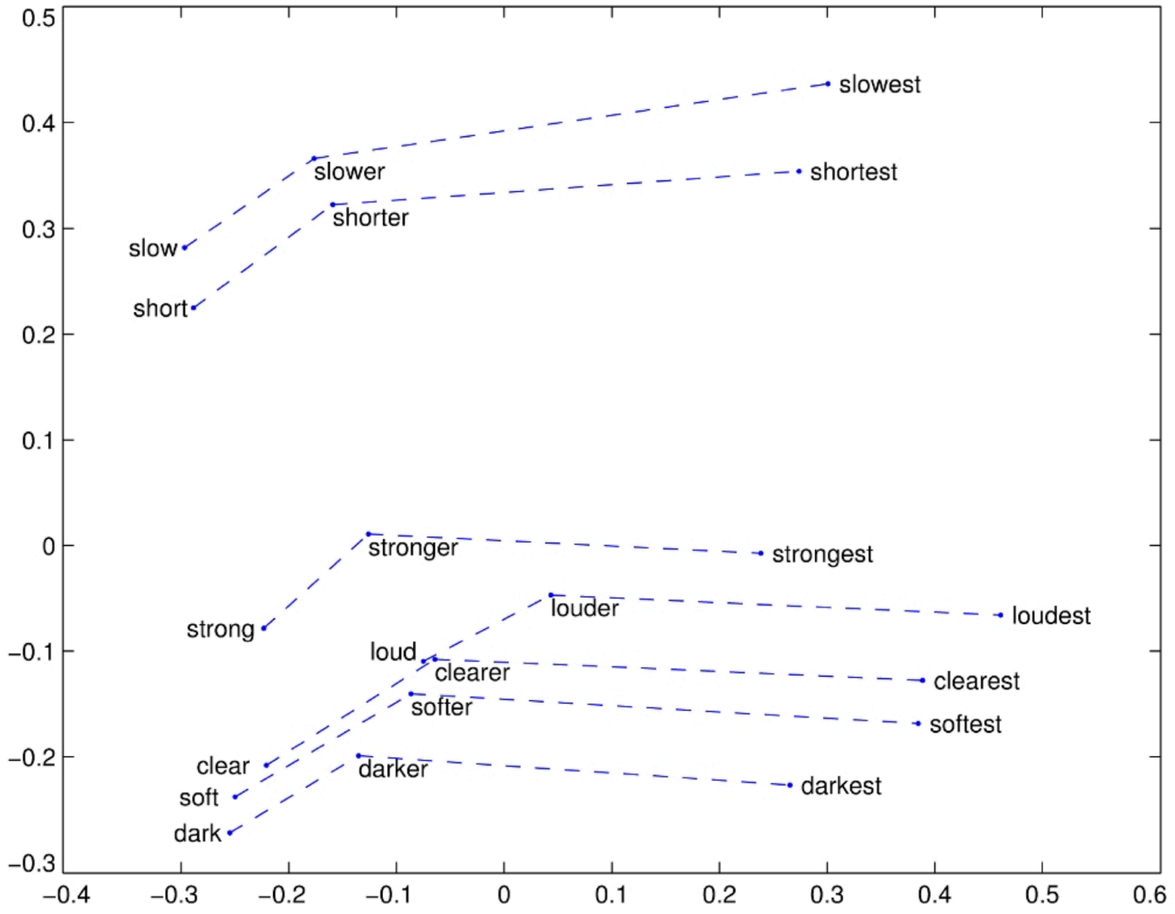
_____ = **tree** + (grape - apple)

- Why does this work?
- (grape - apple) captures differences in context, while tree specifies the type of object in the analogy

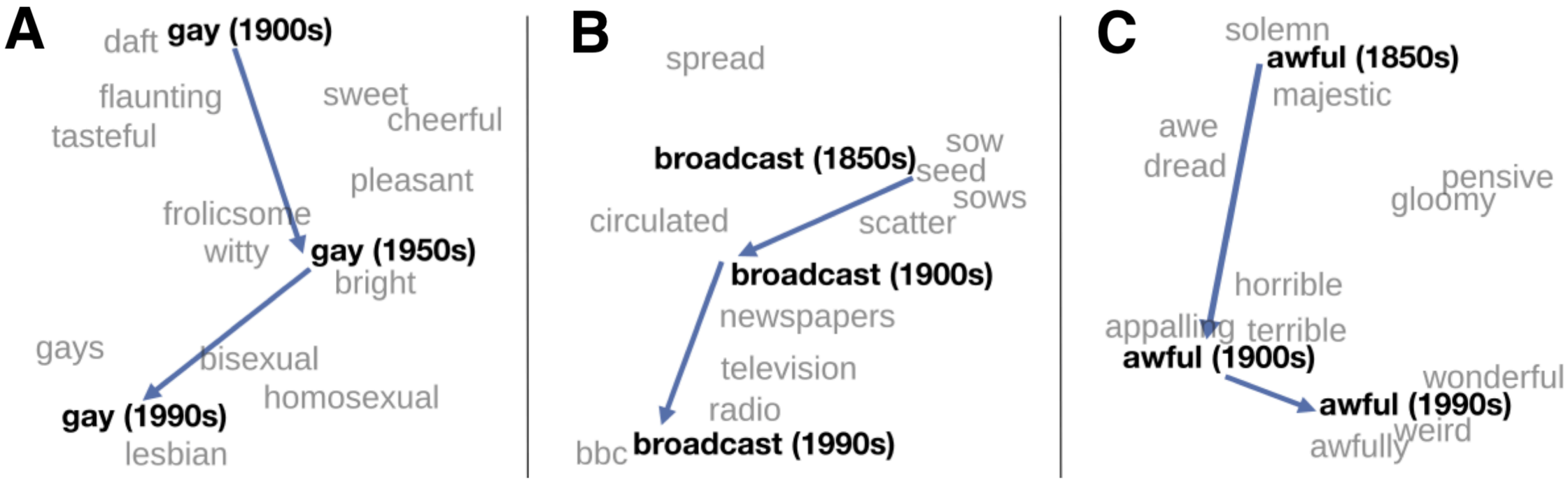




Consistent gender direction in GloVe



Captures comparative/superlative morphology, too!



If we compute embeddings from three different time periods, we can view the change in the meanings of particular words!

Social Considerations

- Word embeddings are known to capture gender and racial biases.
  - **Bolukbasi et al. [2016]**: the closest occupation to "computer programmer - man + woman" is homemaker.
  - Embeddings are often *more* biased than the actual text statistics **[Zhao et al., 2017; Etharayajh et al., 2019]** and actual labor statistics **[Garg et al., 2018]**
  - The cosine similarity of traditionally African American names with negative words is higher than that of traditionally European American names with negative words

- A lot of research tries to remove these properties from embeddings by applying some transformation to the learned embeddings **[**e.g., **Zhao et al., 2017; 2018]**. These generally do not fully remove bias **[Gonen & Goldberg, 2019]**.

Preview: Contextual Embeddings

- All of the embeddings discussed today assume that we can use the exact same embedding for a given word in *any* context.
- But words often have many meanings depending on context:

I went to the **park**

I will **park** my car

The industrial **park** was abandoned

You can **park** your bag by the door

- Later, we'll discuss ways of taking this context into account in our embeddings.

Summary

- Many methods now exist that allow us to automatically embed a token into a vector representation.
- Lots of pretrained embeddings work well in practice, and they often capture interesting high-level trends and analogies.

- Next time: overcoming short-context limitations with **recurrent neural networks** (RNNs)