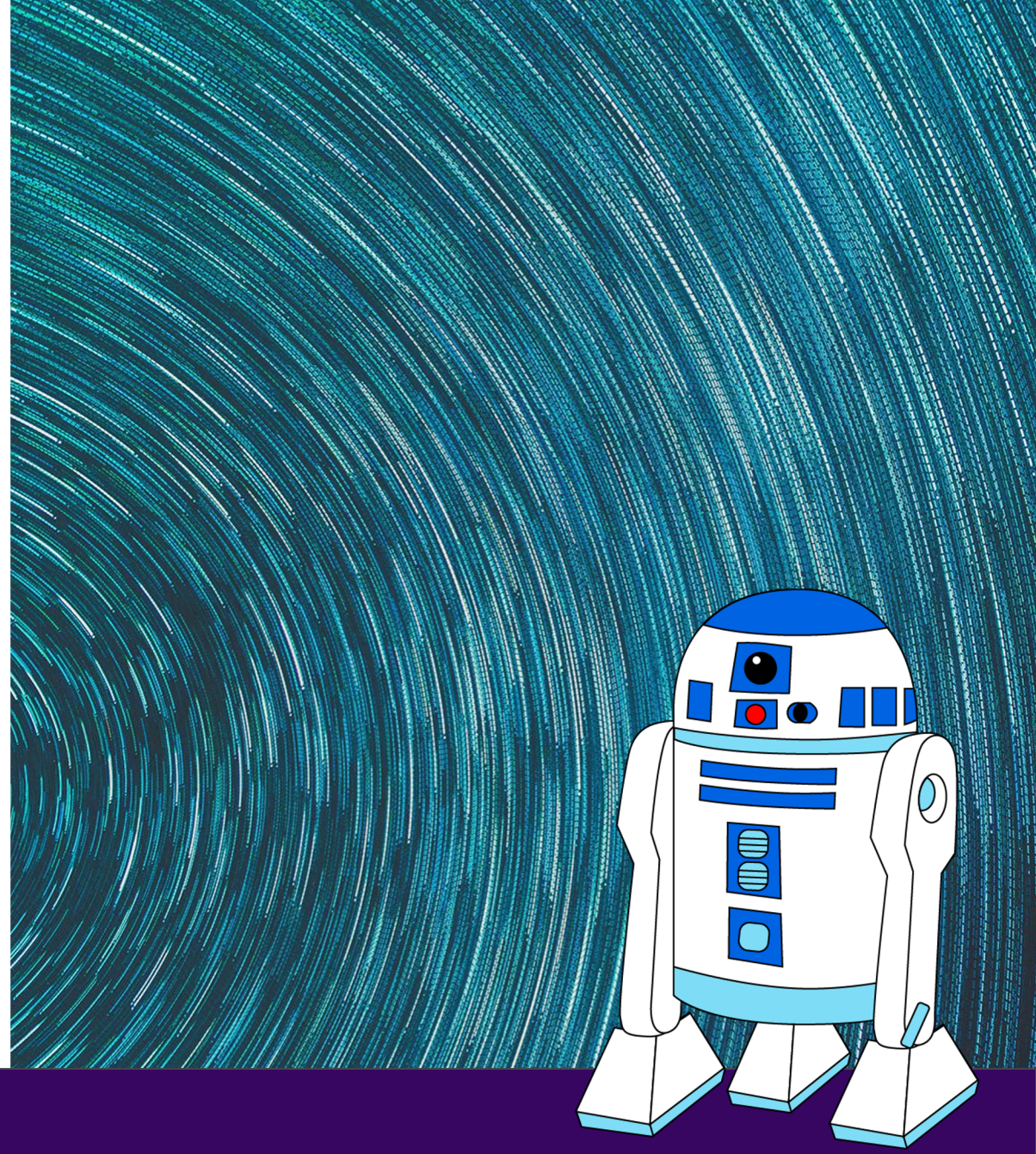
CIS 421/521:
ARTIFICIAL INTELLIGENCE

Neural Networks and Neural LMs

Jurafsky and Martin Chapters 7 and 9



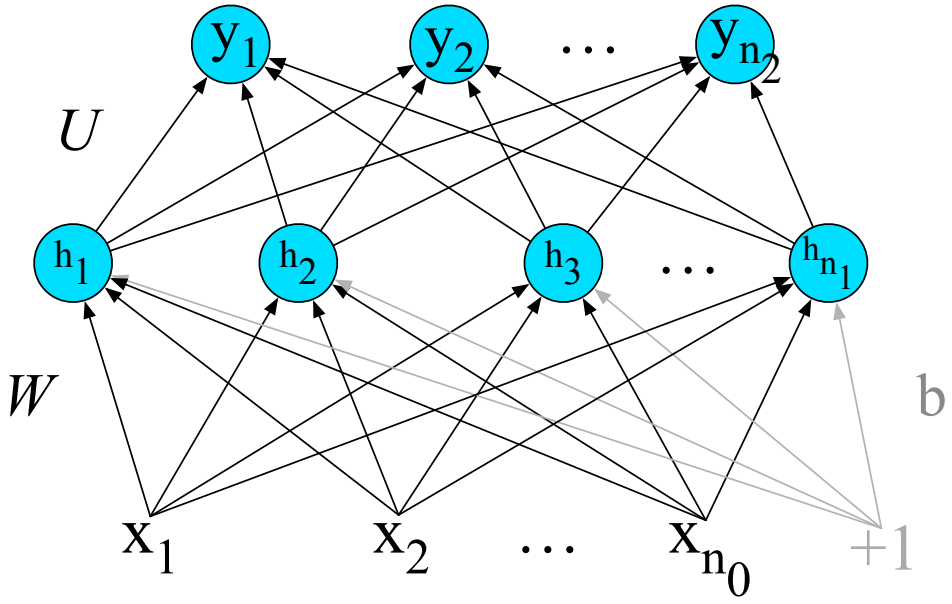
Penn Engineering
University of Pennsylvania

Review: Feed-Forward Neural Network

The simplest kind of is the **Feed-Forward Neural Network**

Multilayer network, all units are usually **fully-connected**, and **no cycles**.

The outputs from each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.



Layer 2 (output layer)

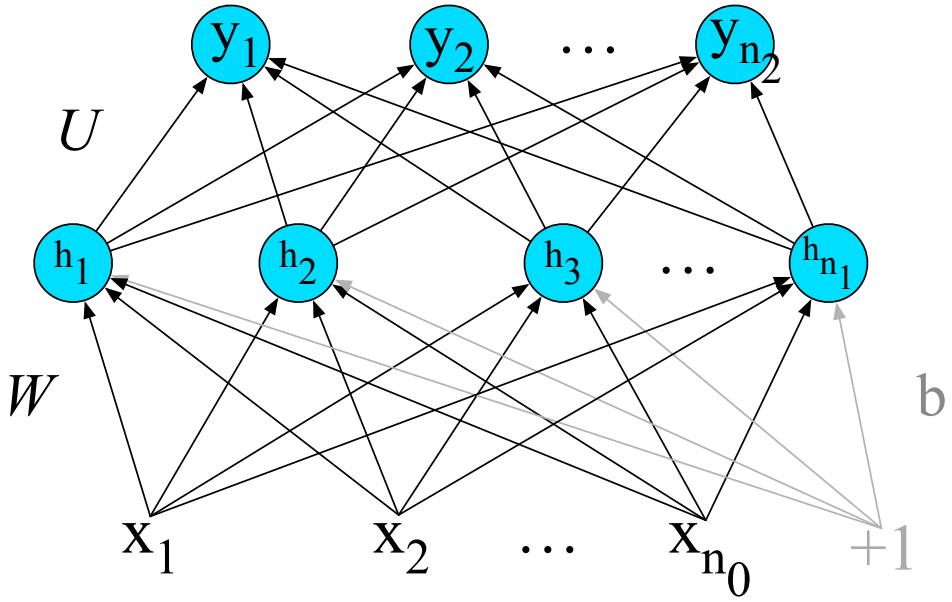
Layer 1 (hidden layer)

Layer 0 (input layer)

Review: Feed-Forward Neural Network

A single hidden unit has parameters ***w*** (the weight vector) and ***b*** (the bias scalar).

We represent the parameters for the **entire hidden layer** by combining the weight vector $\mathbf{w}_i$ and bias $\boldsymbol{b}_i$ for each unit ***i*** into a single weight matrix ***W*** and a single bias vector **b** for the whole layer.



Layer 2 (output layer)

Layer 1 (hidden layer)

Layer 0 (input layer)

Review: Feed-Forward Neural Network

The advantage of using a single matrix ***W*** for the weights of the entire layer is the hidden layer computation can be done efficiently with simple matrix operations.

The computation has three steps:

1. multiplying the weight matrix by the input vector *x*,
2. adding the bias vector *b*, and
3. applying the activation function *g* (such as Sigmoid)

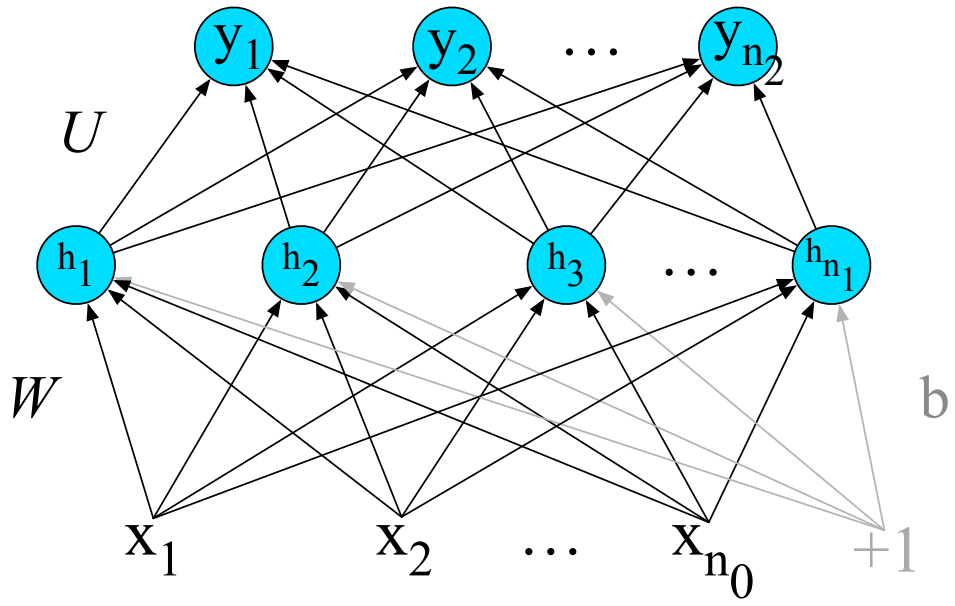
The output of the hidden layer, the vector *h*, is thus the following, using the sigmoid function σ:

$$h = \sigma(Wx+b)$$

Review: Feed-Forward Neural Network

Like the hidden layer, the output layer has a weight matrix ***U***.
Its weight matrix is multiplied by its input vector (***h***) to produce the intermediate output ***z***.

$$z=Uh$$



Layer 2 (output layer)

Layer 1 (hidden layer)

Layer 0 (input layer)

Review: Feed-Forward Neural Network

Here are the final equations for a feedforward network with a single hidden layer, which takes an input vector x, outputs a probability distribution y, and is parameterized by weight matrices W and U and a bias vector b:

$h = \sigma(Wx+b)$
$z = Uh$
$y = \text{softmax}(z)$

Like with logistic regression, softmax normalizes the output and turns it into a probability distribution.

Training Neural Nets

Like logistic regression, we want to learn the best parameters for the neural net to make its predictions $\hat{y}$ as close to possible as the gold standard labels in our training data *y*.

What do we need?

A loss function – cross-entropy loss

An optimization algorithm – gradient descent

A way of computing the gradient of the loss function – error propagation

Cross-Entropy Loss

If the neural network is a binary classifier with a sigmoid at the final layer, the loss function is exactly the same as we saw in logistic regression:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

Cross-Entropy Loss

If the neural network is a binary classifier with a sigmoid at the final layer, the loss function is exactly the same as we saw in logistic regression:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

For multinomial classification

$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

If there is only one correct answer, where the truth is $y_i=1$, then this simplifies to be

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i$$

Plugging into softmax:

$$L_{CE}(\hat{y}, y) = -\log \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Computing the gradient

Logistic regression can be thought of as a network with just one weight layer and a sigmoid output. In that case the gradient is:

$$\frac{\partial LCE(w,b)}{\partial wj} = (\hat{y} - y)\, xj$$

$$= (\sigma(w \cdot x + b) - y)x_j$$

But these derivatives only give correct updates for the last weight layer!

For deeper networks, computing the gradients requires looking back through all the earlier layers in the network, even though the loss is only computed with respect to the output of the network.

Solution: **error backpropagation algorithm**

Computation Graphs

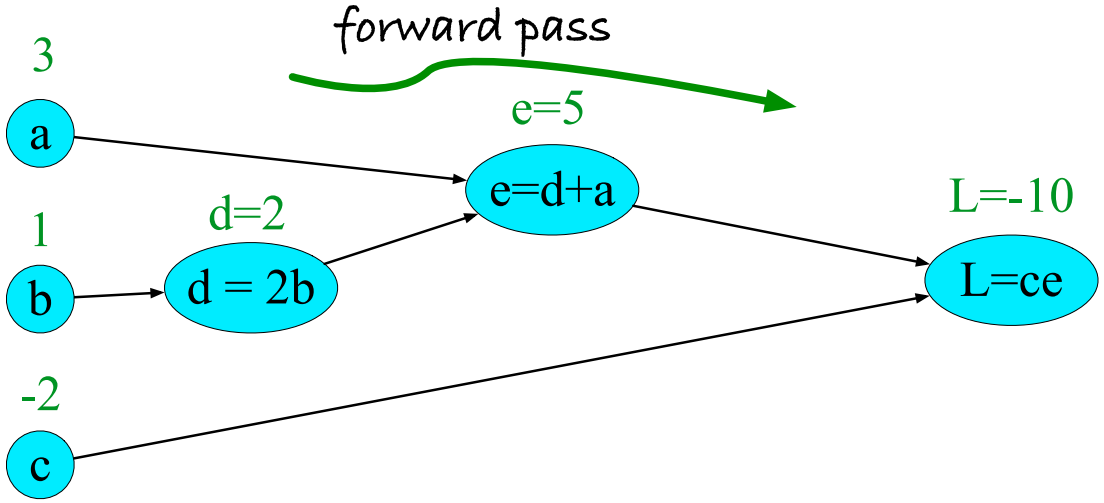
Although backpropagation was invented for neural nets, it is related to general procedure called **backward differentiation**, which depends on the notion of **computation graphs**.

A computation graph represents the process of computing a mathematical expression. The computation is broken down into separate operations. Each operation is a node in a graph.

$$L(a, b, c) = c(a + 2b)$$

$$d = 2 * b$$
$$e = a + d$$
$$L = c * e$$

Forward pass



$L(a, b, c) = c(a + 2b)$

$d = 2*b$
$e = a+d$
$L = c*e$

inputs $a = 3$, $b = 1$, $c = -2$,

Backward differentiation

The importance of the computation graph comes from the backward pass, which is used to compute the derivatives that we'll need for the weight update.

How do we compute the derivative of our output function L with respect to the input variables a, b, and c?

Backwards differentiation uses the **chain rule** from calculus.

$$\frac{\partial L}{\partial a}, \frac{\partial L}{\partial b}, \text{ and } \frac{\partial L}{\partial c}$$

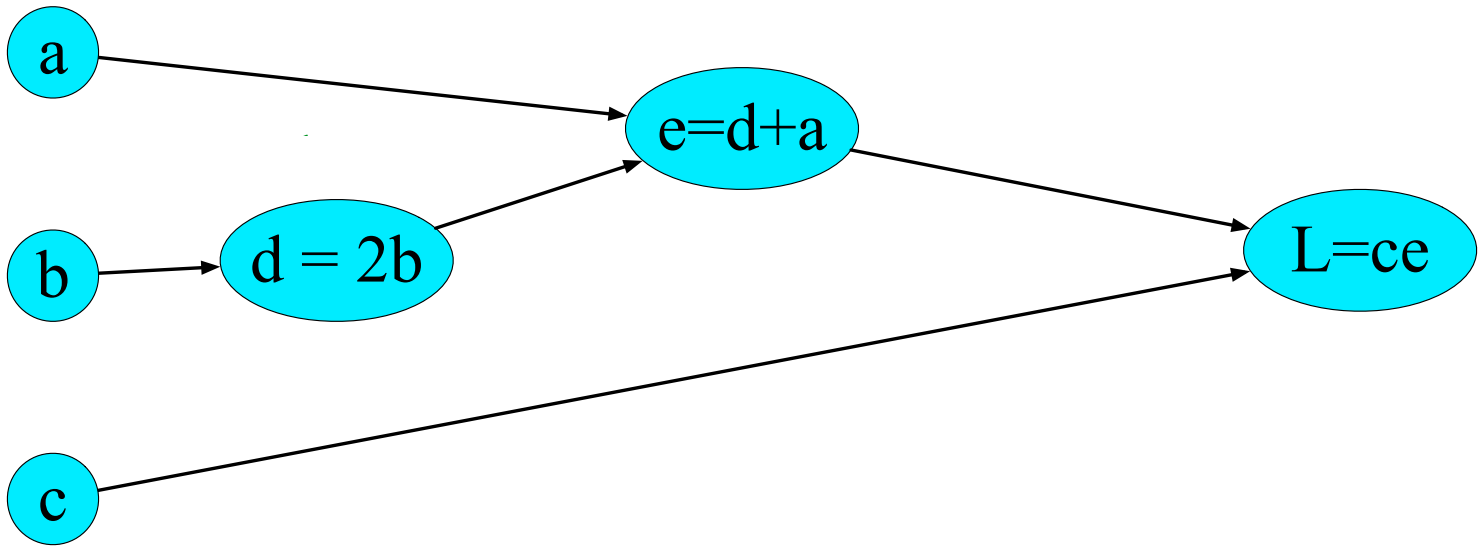
Chain rule

For a composite function $f(x) = u(v(x))$, the derivative of $f(x)$ is:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

Similarly for, $f(x) = u(v(w(x)))$, the derivative of $f(x)$ is:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$

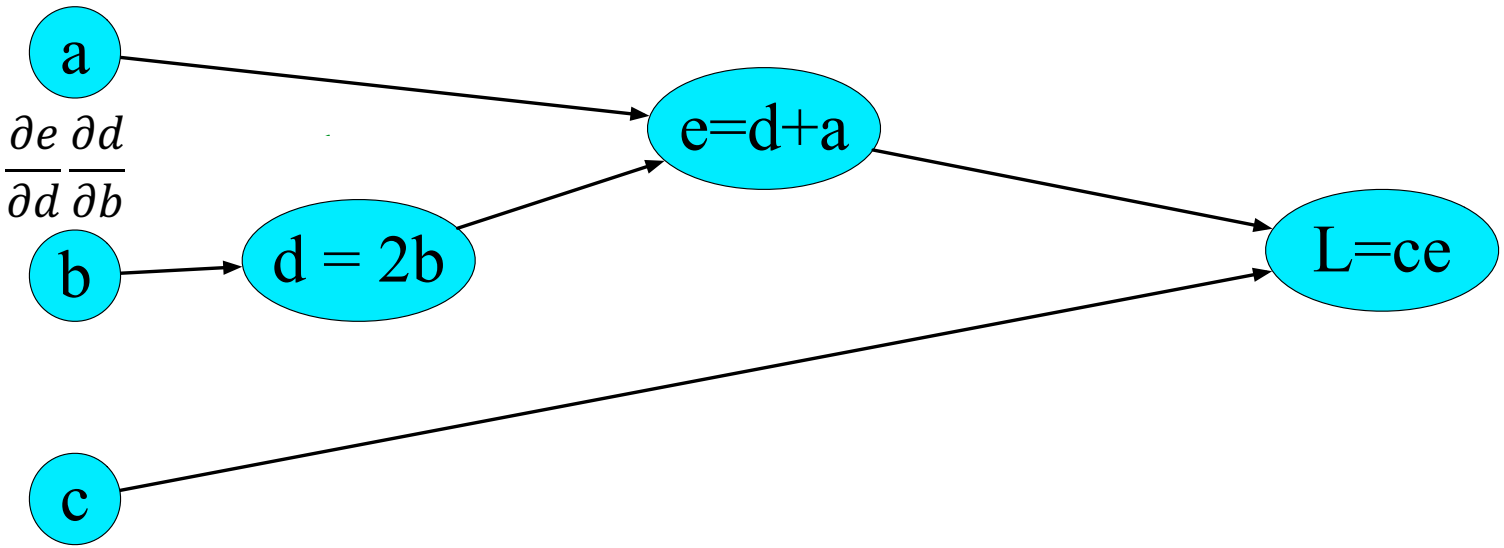


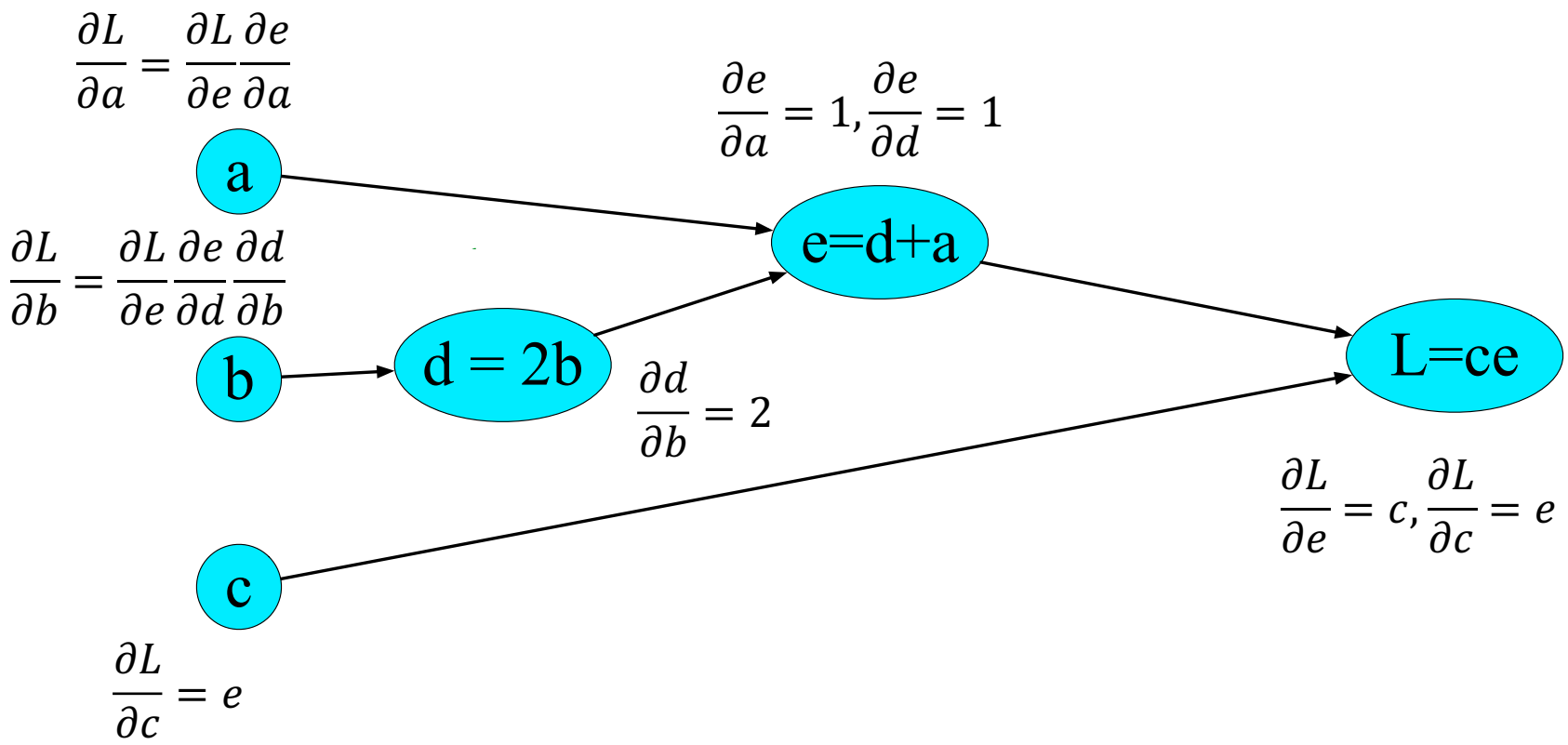
$$\frac{\partial L}{\partial c} = e$$

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

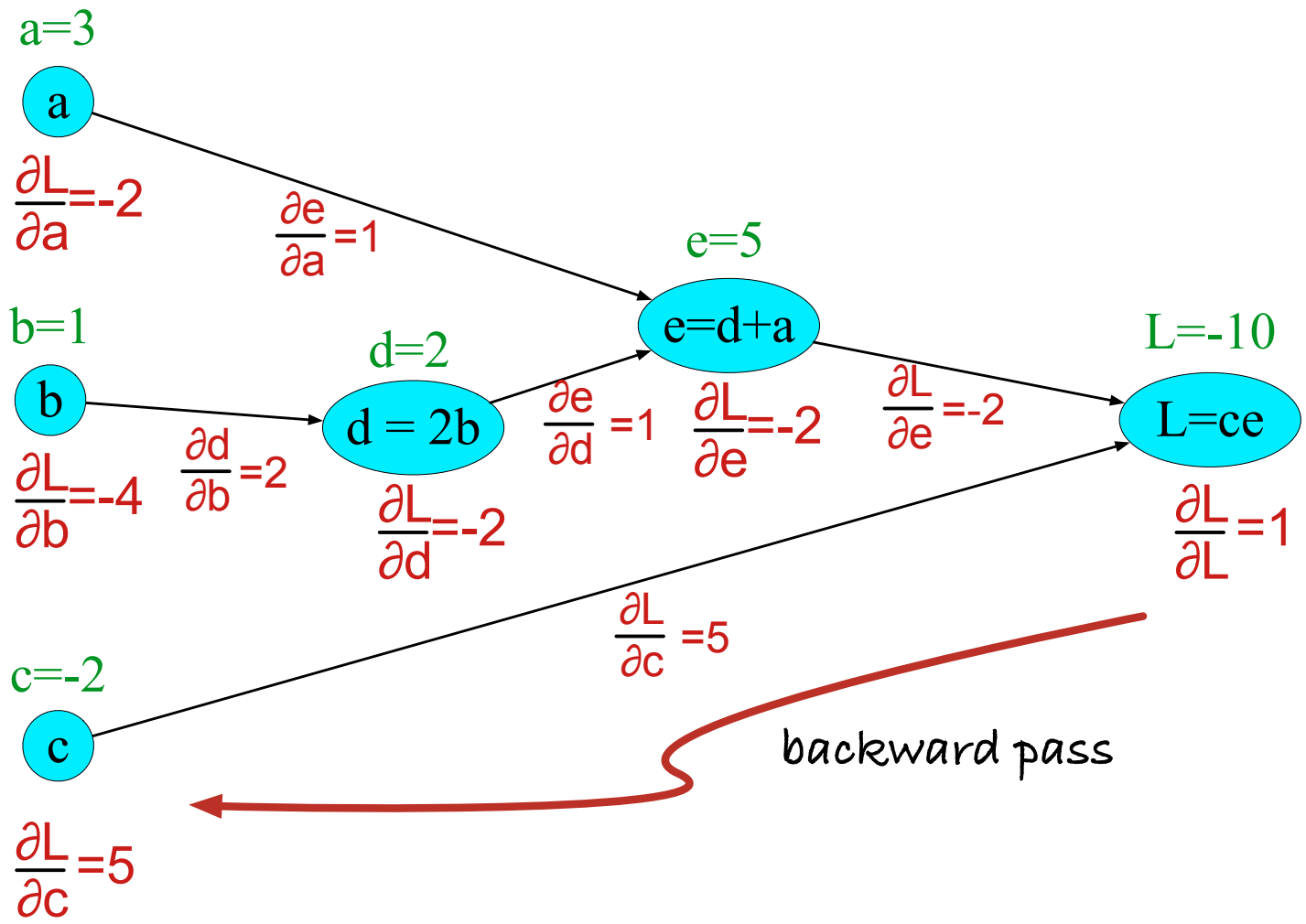
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

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

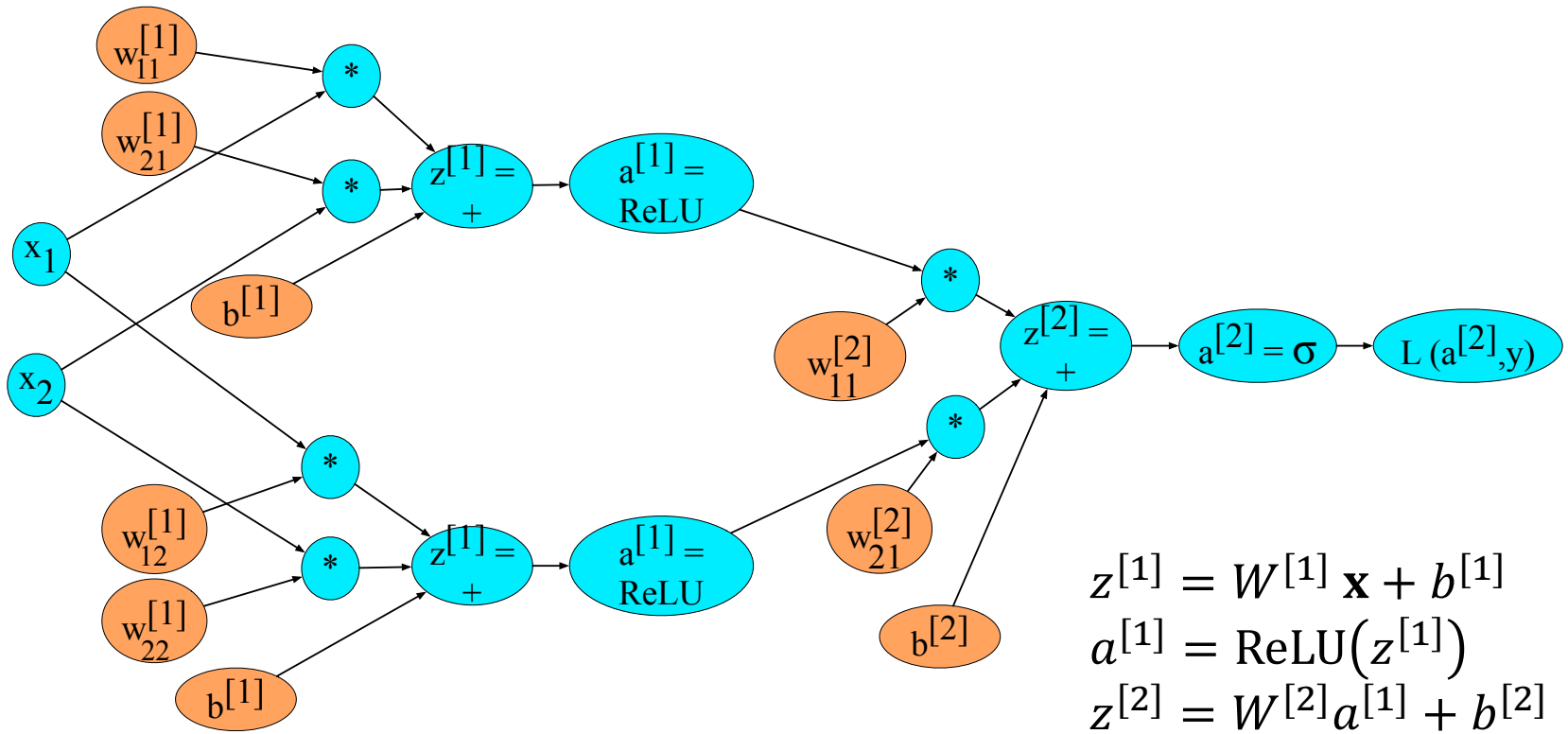



$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$
$\frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$
a
e=d+a
$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$
b
d = 2b
$\frac{\partial d}{\partial b} = 2$
L=ce
$\frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$
c
$\frac{\partial L}{\partial c} = e$

Backward pass



Computation Graph for a NN



$$z^{[1]} = W^{[1]}\,\mathbf{x} + b^{[1]}$$
$$a^{[1]} = \text{ReLU}\left(z^{[1]}\right)$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma\left(z^{[2]}\right)$$
$$\hat{y} = a^{[2]}$$

Neural Language Models

Language Models

Estimate the probability of a sentence consisting of word sequence $w_{1:n}$

$$P(w_{1:n}) \approx \prod_{i=1}^{n} P(w_i \mid w_{i-k:i-1})$$

We need to estimate the probability of $P(w_{i+1} \mid w_{k-i:i})$ from a large corpus.

$$\hat{p}_{MLE}(w_{i+1} = m \mid w_{i-k:i}) = \frac{\#(w_{i-k:i+1})}{\#(w_{i-k:i})}$$

$$\hat{p}_{add-\alpha}(w_{i+1} = m \mid w_{i-k:i}) = \frac{\#(w_{i-k:i+1}) + \alpha}{\#(w_{i-k:i}) + \alpha|V|}$$

Neural Language Models

Neural Language models have several advantages over n-gram LMs:

1. They don’t need smoothing
2. They can handle much longer histories.
3. They can generalize over contexts of similar words.
4. Neural LMs tend to have much higher predictive accuracy than n-gram LMs.

Disadvantage: slower to train than traditional n-gram LMs

Neural LMs (Bengio et al 2003)

1. Associate each word in the vocabulary with a vector-representation, thereby creating a notion of similarity between words.
2. Express the joint probability *function* of a word sequence in terms of the word vectors for the words in that sequence.
3. Simultaneously learn the word vectors and the parameters of the *function.*

The word vectors are low-dimensional (d=30 to d=100) dense vectors, like we've seen before.

The probability function is expressed the product of conditional probabilities of the next word given the previous word, *using a multi-layer, feed forward neural network.*

Neural LMs

The input to the neural network is a k-gram of words $w_{1:k}$.

The output is a probability distribution over the next word.

The *k* context words are treated as a word window. Each word is associated with an **embedding** vector:

The input vector **x** just concatenates v(w) for each of the k words:

$$v(w) \in \mathbb{R}^{d_w}$$

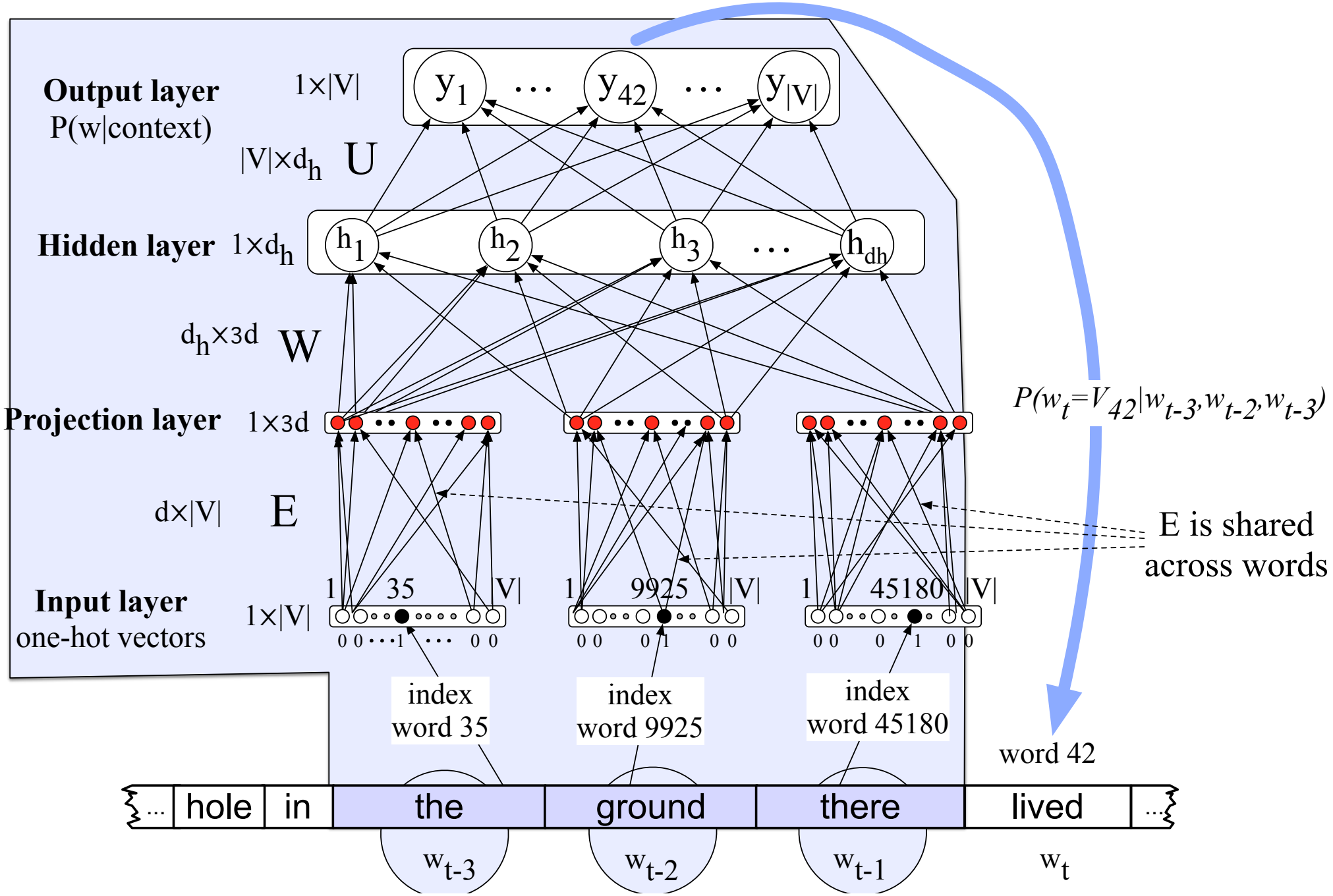
$$\boldsymbol{x} = [v(w_1); v(w_2); \ldots; v(w_k)]$$

Neural LMs

The input **x** is fed into a neural network with 1 or more hidden layers:

$$\hat{\boldsymbol{y}} = P(w_i|w_{1:k}) = LM(w_{1:k}) = \text{softmax}(\boldsymbol{h}\boldsymbol{W}^{\boldsymbol{2}} + \boldsymbol{b}^{\boldsymbol{2}})$$
$$\boldsymbol{h} = g(\boldsymbol{x}\boldsymbol{W}^{\boldsymbol{1}} + \boldsymbol{b}^{\boldsymbol{1}})$$
$$\boldsymbol{x} = [v(w_1); v(w_2); \ldots; v(w_k)]$$
$$v(w) = \boldsymbol{E}_{[w]}$$

$$w_i \in V \quad \boldsymbol{E} \in \mathbb{R}^{|V| \times d_w} \quad \boldsymbol{W}^{\boldsymbol{1}} \in \mathbb{R}^{k \cdot d_w \times d_{\text{hid}}} \quad \boldsymbol{b}^{\boldsymbol{1}} \in \mathbb{R}^{d_{\text{hid}}} \quad \boldsymbol{W}^{\boldsymbol{2}} \in \mathbb{R}^{d_{\text{hid}} \times |V|} \quad \boldsymbol{b}^{\boldsymbol{2}} \in \mathbb{R}^{|V|}$$



Training

The training examples are simply word k-grams from the corpus

The identities of the first k+1 words are used as features, and the last word is used as the target label for the classification.

Conceptually, the model is trained using cross-entropy loss.

Advantages of NN LMs

Better results. They achieve better preplexity scores than SOTA n-gram LMs.

Larger N. NN LMs can scale to much larger orders of n. This is achievable because parameters are associated only with individual words, and not with n-grams.

They generalize across contexts. For example, by observing that the words *blue, green, red, black*, etc. appear in similar contexts, the model will be able to assign a reasonable score to the *green car* even though it never observed in training, because it did observe *blue car* and *red car*.

A by-product of training are **word embeddings**

Language Modeling

Goal: Learn a **function** that returns the joint probability

Primary difficulty:

1. There are too many parameters to accurately estimate. This is sometimes called the “curse of dimensionality”
2. In n-gram-based models we fail to generalize to related words / word sequences that we have observed.

Curse of dimensionality / sparse statistics

Suppose we want a joint distribution over 10 words.
Suppose we have a vocabulary of size 100,000.

$$100{,}000^{10} = 10^{50} \text{ parameters}$$

This is too high to estimate from data.

Chain rule

In LMs we user chain rule to get the conditional probability of the next word in the sequence given all of the previous words:

$$P(w_1 w_2 w_3 \ldots w_t) = \prod_{t=1}^{T} P(w_t | w_1 \ldots w_{t-1})$$

What assumption do we make in n-gram LMs to simplify this?

The probability of the next word only depends on the previous *n*-1 words.

A small *n* makes it easier for us to get an estimate of the probability from data.

Probability tables

We construct tables to look up the probability of seeing a word given a history.

curse of	$P(w_t \mid w_{t-n} \ldots w_{t-1})$
dimensionality	
azure	
knowledge	
oak	

The tables only store observed sequences.

What happens when we have a new (unseen) combination of n words?

Unseen sequences

What happens when we have a new (unseen) combination of n words?

1. Back-off
2. Smoothing / interpolation

We are basically just stitching together short sequences of observed words.

Alternate idea

Let's try **generalizing**.

Intuition: Take a sentence like

The cat is walking in the bedroom

And use it when we assign probabilities to similar sentences like

The dog is running around the room

A Neural Probabilistic LM

Bengio et al NIPS 2003

1. Use a vector space model where the words are vectors with real values $\mathbb{R}^m$. m=30, 60, 100. This gives a way to compute word similarity.
2. Define a function that returns a joint probability of words in a sequence based on a sequence of these vectors.
3. Simultaneously learn the word representations **and** the probability function from data.

Seeing one of the cat/dog sentences allows them to increase the probability for that sentence **and** its combinatorial # of **"neighbor" sentences** in vector space.

A Neural Probabilistic LM

Given:

A training set $w_1 \ldots w_t$ where $w_t \in V$

Learn:

$f(w_1 \ldots w_t) = P(w_t | w_1 \ldots w_{t-1})$

Subject to giving a high probability to an unseen text/dev set (e.g. minimizing the perplexity)

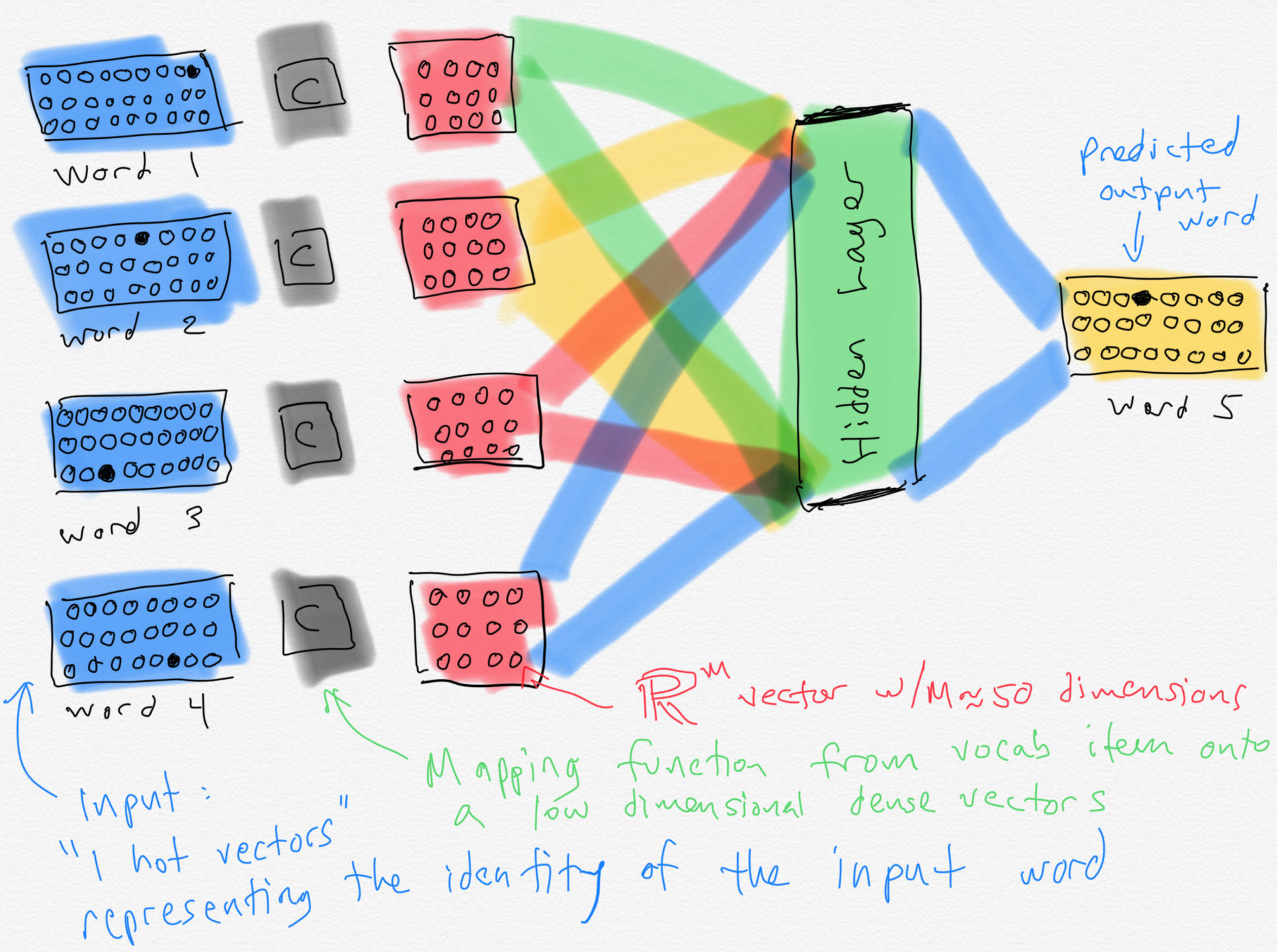
Constraint:

Create a proper probability distribution (e.g. sums to 1) so that we can take the product of conditional probabilities to get the joint probability of a sentence

A Neural Probabilistic LM

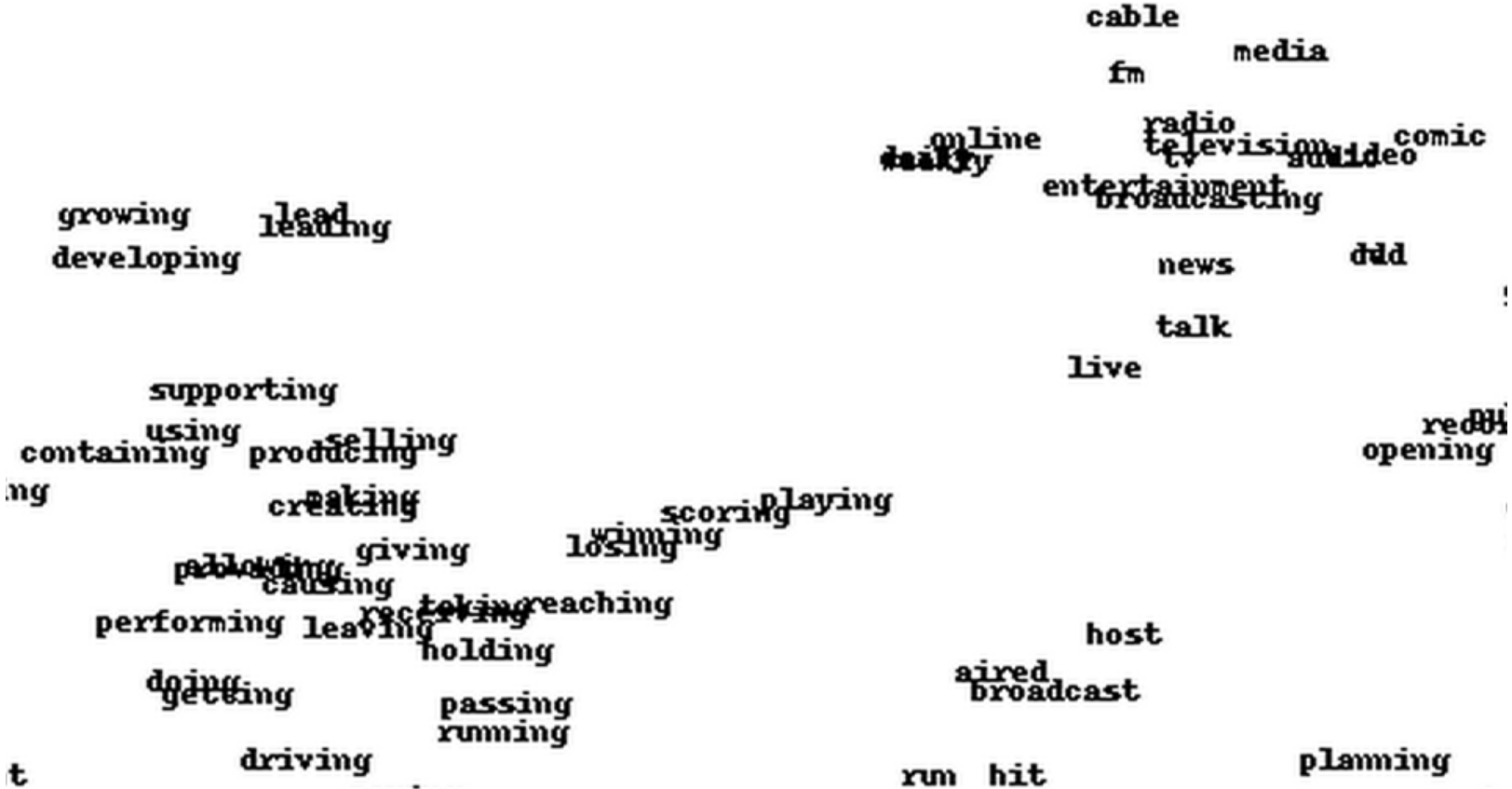
1. Create a mapping function C from any word in V onto $\mathbb{R}^M$. Store this in a V-by-M matrix. Initialize it with singular value decomposition (SVD).
2. The neural architecture: a function *g* maps sequence of word vectors onto a probability distribution over the vocabulary V

$$g(C(w_{t-n}) \ldots C(w_{t-1})) = P(w_t | w_{t-n} \ldots w_{t-1})$$



Word embeddings

When the ~50 dimensional vectors that result from training a neural LM are projected down to 2-dimensions, we see a lot of words that are intuitively similar to each other are close together.



Current state of the art neural LMs

ELMo

GPT

BERT

GPT-2