Neural Networks and Neural Language Models

Reference:
- D. Jurafsky and J. Martin, “Speech and Language Processing”

Introduction

- Neural networks are a fundamental computational tool for language processing, and a very old one.
- They are called neural because their origins lie in the McCulloch Pitts neuron, a simplified model of the human neuron as a kind of computing element that could be described in terms of propositional logic.
- But the modern use in language processing no longer draws on these early biological inspirations.

Introduction

- Instead, a modern neural network is a network of small computing units, each of which takes a vector of input values and produces a single output value.
- We introduce the neural net applied to classification. The architecture we introduce is called a feed-forward network because the computation proceeds iteratively from one layer of units to the next.
- The use of modern neural nets is often called deep learning, because modern networks are often deep (have many layers).

Introduction

- Neural networks share much of the same mathematics as logistic regression.
- But neural networks are a more powerful classifier than logistic regression.
  - Indeed a minimal neural network (technically one with a single ‘hidden layer’) can be shown to learn any function.
- When working with neural networks, it is more common to avoid most uses of rich hand-derived features, instead building neural networks that take raw words as inputs and learn to induce features as part of the process of learning to classify.

Introduction

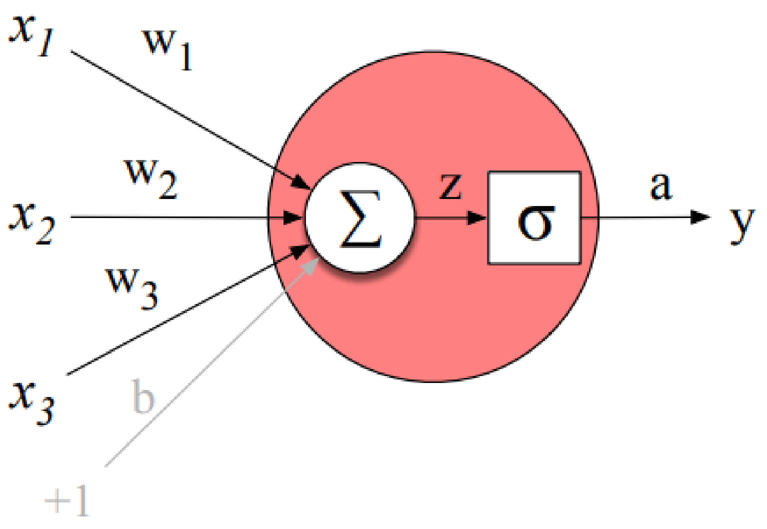
- Nets that are very deep are particularly good at representation learning
  - Deep neural nets are the right tool for large scale problems that offer sufficient data to learn features automatically.
- We will introduce feedforward networks as classifiers, and apply them to the simple task of language modeling: assigning probabilities to word sequences and predicting upcoming words.

Units

- The building block of a neural network is a single computational unit. A unit takes a set of real valued numbers as input, performs some computation on them, and produces an output.
- At its heart, a neural unit is taking a weighted sum of its inputs, with one additional term in the sum called a bias term.

Units

Below is a final schematic of a basic neural unit. In this example the unit takes 3 input values x_1,x_2, and x_3, and computes a weighted sum, multiplying each value by a weight (w_1, w_2, and w_3, respectively), adds them to a bias term b, and then passes the resulting sum through a sigmoid function to result in a number between 0 and 1.



- A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y.
- Some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a.
- In this case the output of the unit y is the same as a, but in deeper networks we'll reserve *y* to mean the final output of the entire network, leaving a as the activation of an individual node.

Units

- Given a set of inputs $x_1 \dots x_n$, a unit has a set of corresponding weights $w_1 \dots w_n$ and a bias b, so the weighted sum z can be represented as:

$$z = b + \sum_i w_i x_i$$

- Often it's more convenient to express this weighted sum using vector notation.
- Thus we'll talk about z in terms of a weight vector w, a scalar bias b, and an input vector x, and we'll replace the sum with the convenient **dot product**:

$$z = w \cdot x + b$$

z is just a real valued number.

Units

- Finally, instead of using z, a linear function of x, as the output, neural units apply a non-linear function f to z.
- We will refer to the output of this function as the activation value for the unit, a.
- Since we are just modeling a single unit, the activation for the node is in fact the final output of the network, which we'll generally call y. So the value y is defined as:

$$y = a = f(z)$$

Units

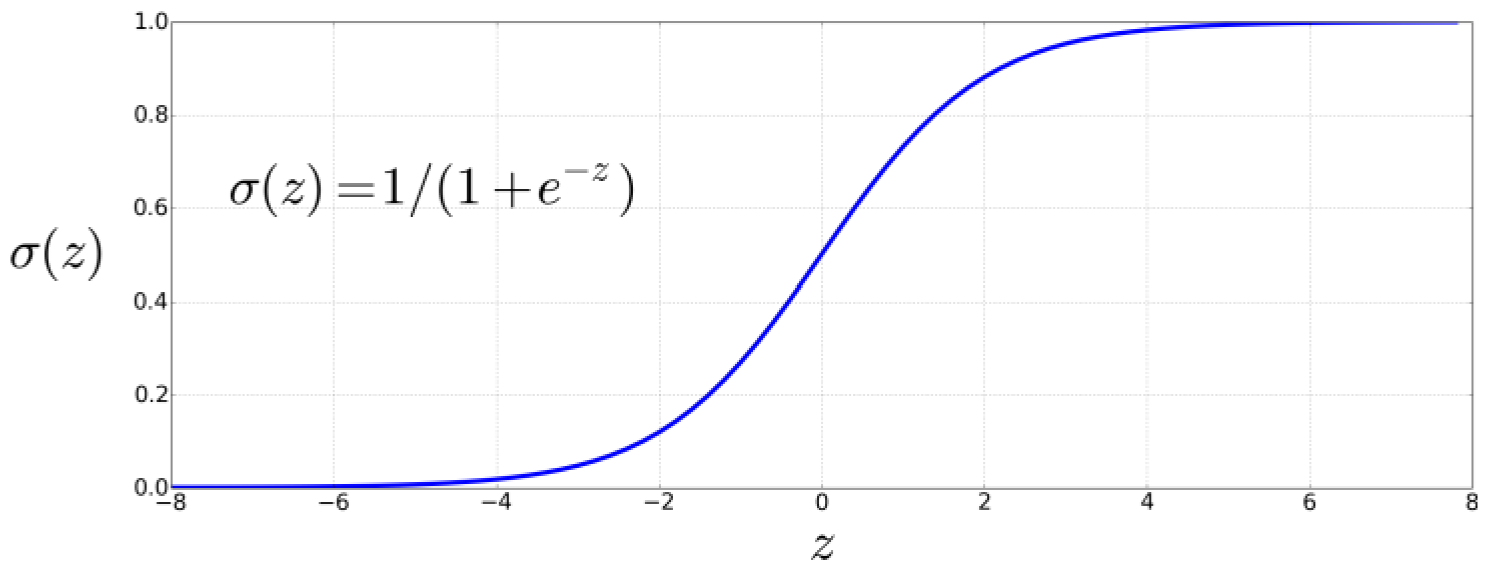
- We'll discuss three popular non-linear functions $f()$ below (the **sigmoid**, the **tanh**, and the rectified linear **ReLU**) but it's pedagogically convenient to start with the **sigmoid** function :

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

- The sigmoid has a number of advantages; it maps the output into the range [0,1], which is useful in squashing outliers toward 0 or 1. And it's differentiable, which will be handy for learning.
- Substituting the sigmoid equation gives us the output of a neural unit:

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))}$$

Units



The sigmoid function takes a real value and maps it to the range [0, 1]. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

Units

Suppose we have a unit with the following weight vector and bias:

$$w = [0.2, 0.3, 0.9]$$
$$b = 0.5$$

What would this unit do with the following input vector:

$$x = [0.5, 0.6, 0.1]$$

The resulting output y would be:

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}}$$
$$= \frac{1}{1 + e^{-(.5 * .2 + .6 * .3 + .1 * .9 + .5)}}$$
$$= \frac{1}{1+e^{-0.87}} = .70$$

Units

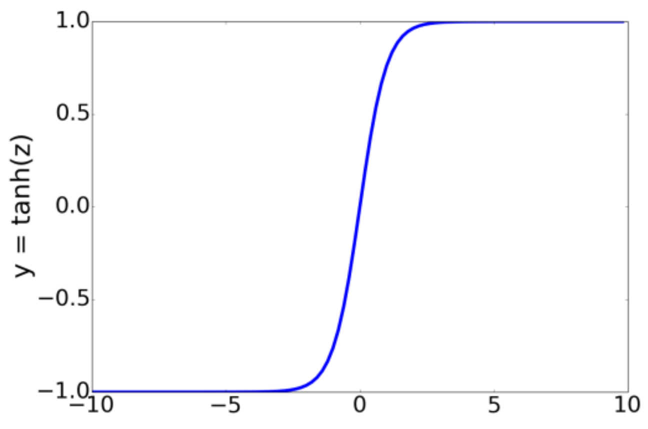
- In practice, the sigmoid is not commonly used as an activation function.
- A function that is very similar but almost always better is the **tanh** function, which is a variant of the sigmoid that ranges from -1 to +1:

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

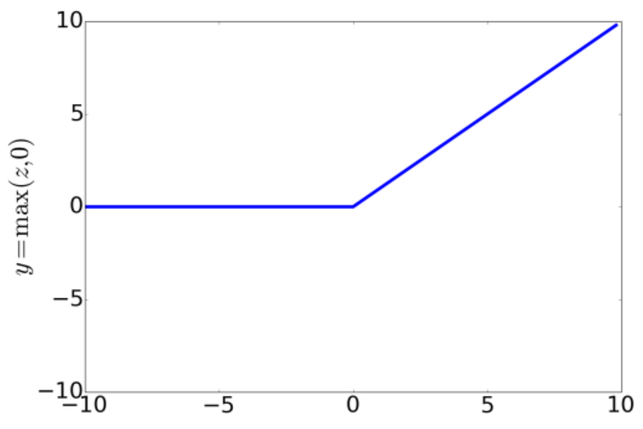
- The simplest activation function, and perhaps the most commonly used, is the rectified linear unit, also called the **ReLU**. It's just the same as z, when z is positive, and 0 otherwise:

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

Units



(a)



(b)

The tanh and ReLU activation functions are shown above.

- The tanh function has the nice properties of being smoothly differentiable and mapping outlier values toward the mean.
- The rectifier function has nice properties that result from it being very close to linear.
- In the sigmoid or tanh functions, very high values of z result in values of y that are saturated, i.e., extremely close to 1, and have derivatives very close to 0, which cause problems for learning because, as we'll see later,
    - we'll train networks by propagating an error signal backwards, multiplying gradients (partial derivatives) from each layer of the network; gradients that are almost 0 cause the error signal to get smaller and smaller until it is too small to be used for training, a problem called the vanishing gradient problem.
- Rectifiers don't have this problem, since the derivative of ReLU for high values of z is 1 rather than very close to 0.

The XOR Problems

- A single neural unit cannot compute some very simple functions of its input.
- Consider the task of computing elementary logical functions of two inputs, like AND, OR, and XOR. As a reminder, here are the truth tables for those functions:

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

The XOR Problems

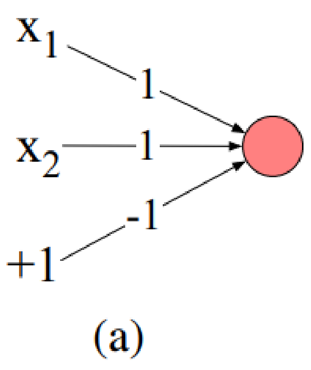
- This example was first shown for the perceptron, which is a very simple neural unit that has a binary output and does not have a non-linear activation function.
- The output y of a perceptron is 0 or 1, and is computed as follows:

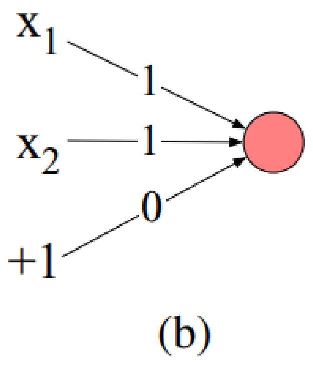
$$y = \begin{cases} 0, & \text{if } w \cdot x + b \le 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

The XOR Problems

- It's very easy to build a perceptron that can compute the logical AND and OR functions of its binary inputs.
- The weights *w* and bias *b* for perceptrons for computing logical functions are shown below. The inputs are shown as $x_1$and x_2 and the bias as a special node with value +1 which is multiplied with the bias weight *b*.

(a) logical AND, showing weights w_1= 1 and w_2= 1 and bias weight $b = -1$.





(b) logical OR, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = 0$.

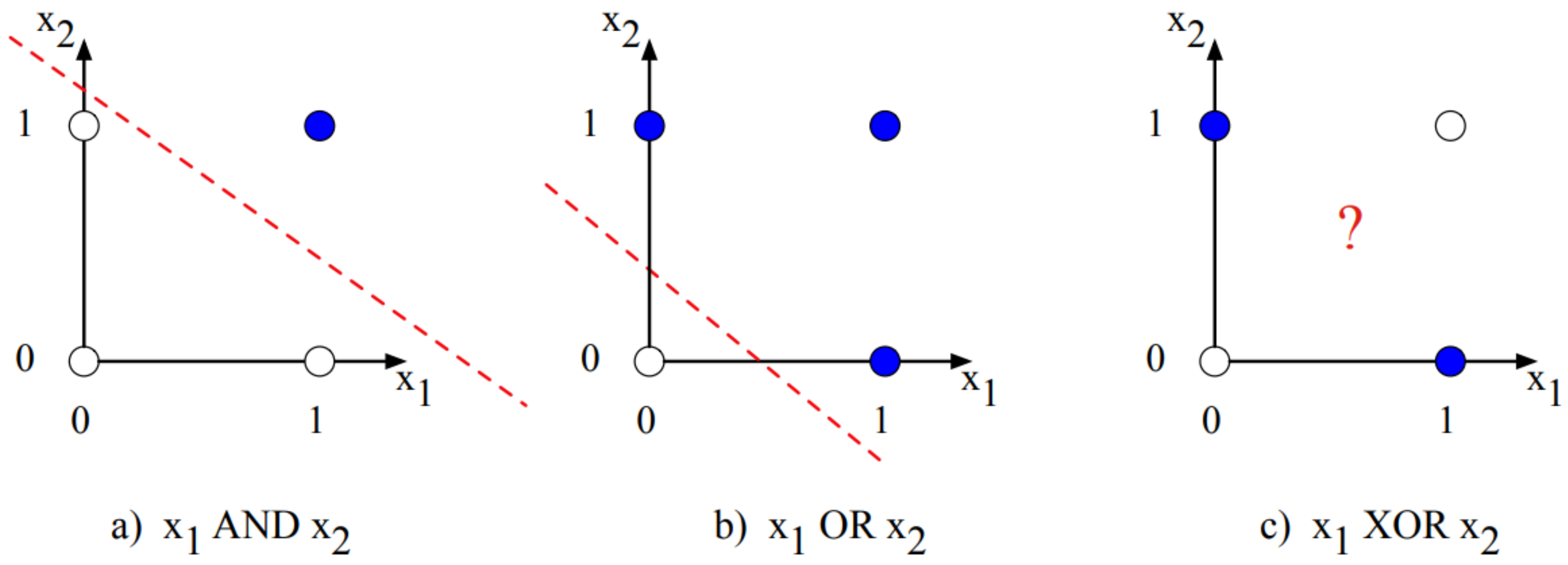
These weights/biases are just one from an infinite number of possible sets of weights and biases that would implement the functions.

The XOR Problems

- However, it’s not possible to build a perceptron to compute logical XOR!
- The intuition behind this important result relies on understanding that a perceptron is a linear classifier.
- For a two-dimensional input x_1 and x_2, the perception equation, $w_1x_1 + w_2x_2 + \ b \ = \ 0$ is the equation of a line
- This line acts as a decision boundary in two-dimensional space in which the output 0 is assigned to all inputs lying on one side of the line, and the output 1 to all input points lying on the other side of the line.
- If we had more than 2 inputs, the decision boundary becomes a hyperplane instead of a line, but the idea is the same, separating the space into two categories.

The XOR Problems

- Shown below are the possible logical inputs $(00, 01, 10, \text{and } 11)$ and the line drawn by one possible set of parameters for an AND and an OR classifier.
- Notice that there is simply no way to draw a line that separates the positive cases of XOR $(01 \text{ and } 10)$ from the negative cases $(00 \text{ and } 11)$. We say that XOR is not a linearly separable function.
  - Of course we could draw a boundary with a curve, or some other function, but not a single line.

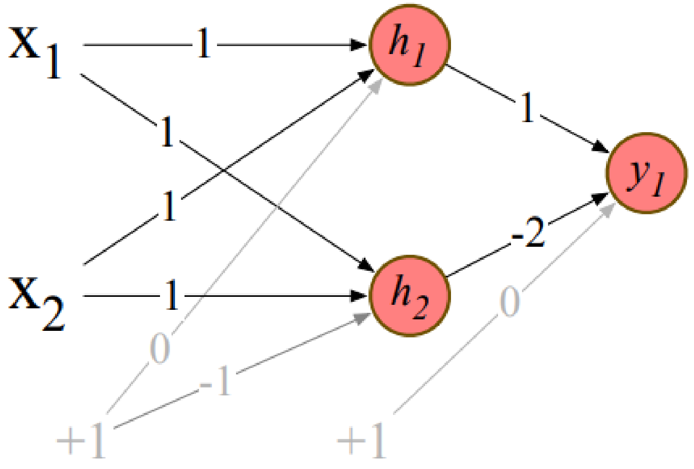


The functions AND, OR, and XOR, represented with input x_1 on the x-axis and input x_2 on the y axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR.

The XOR Problems

The solution: neural networks

- While the XOR function cannot be calculated by a single perceptron, it can be calculated by a layered network of perceptron units.
- Let’s see how to compute XOR using two layers of ReLU-based units instead. Shown below is the input being processed by two layers of neural units. The middle layer (called *h*) has two units, and the output layer (called *y*) has one unit. A set of weights and biases are shown for each ReLU that correctly computes the XOR function.



There are three ReLU units, in two layers; we’ve called them h_1, h_2 (*h* for “hidden layer”) and y_1. As before, the numbers on the arrows represent the weights *w* for each unit, and we represent the bias *b* as a weight on a unit clamped to +1, with the bias weights/units in gray.

The XOR Problems

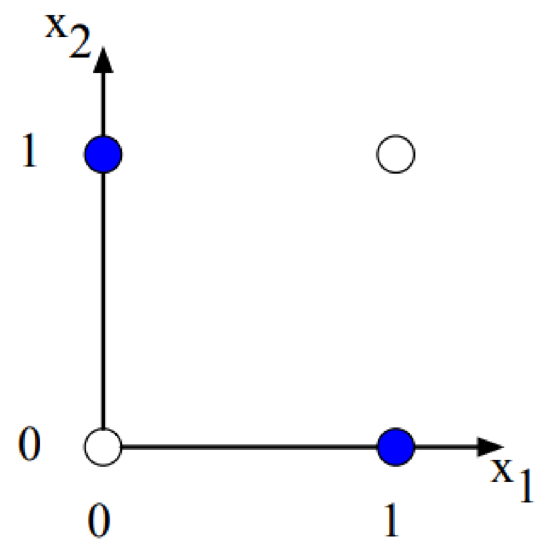
The solution: neural networks

- Let's walk through what happens with the input $x = [0,0]$. If we multiply each input value by the appropriate weight, sum, and then add the bias b, we get the vector $[0,-1]$, and we then apply the rectified linear transformation to give the output of the h layer as $[0,0]$.
- Now we once again multiply by the weights, sum, and add the bias (0 in this case) resulting in the value 0.
- The reader should work through the computation of the remaining 3 possible input pairs to see that the resulting y values correctly are 1 for the inputs $[0,1]$ and $[1,0]$ and 0 for $[0,0]$ and $[1,1]$.

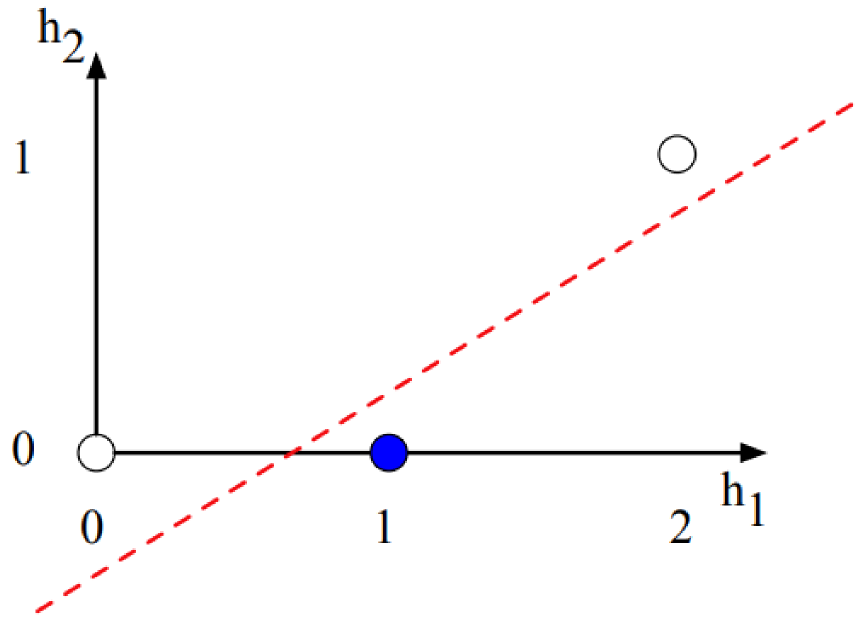
The XOR Problems

The solution: neural networks

- In the previous slide, the *h* vector for the inputs *x* = [0, 0] was [0, 0]. Shown in (b) below are the values of the *h* layer for all 4 inputs.



a) The original x space



b) The new (linearly separable) h space

The hidden layer forming a new representation of the input. (b) is the representation of the hidden layer h, compared to the original input representation x in (a). Notice that the input point $[0, 1]$ has been collapsed with the input point $[1, 0]$, making it possible to linearly separate the positive and negative cases of XOR.

The XOR Problems

The solution: neural networks

- Notice that hidden representations of the two input points $x = [0, 1]$ and $x = [1, 0]$ (the two cases with XOR output = 1) are merged to the single point h = $[1, 0]$. The merger makes it easy to linearly separate the positive and negative cases of XOR.
- In other words, we can view the hidden layer of the network is forming a *representation* for the input.

The XOR Problems

The solution: neural networks

- In the previous example, we just set the weights. For real situations, the weights are learned automatically using the error backpropagation algorithm.
- That means the hidden layers will learn to form useful representations.
- This intuition, that neural networks can automatically learn useful representations of the input, is one of their key advantages, and one that we will return to again and again later.

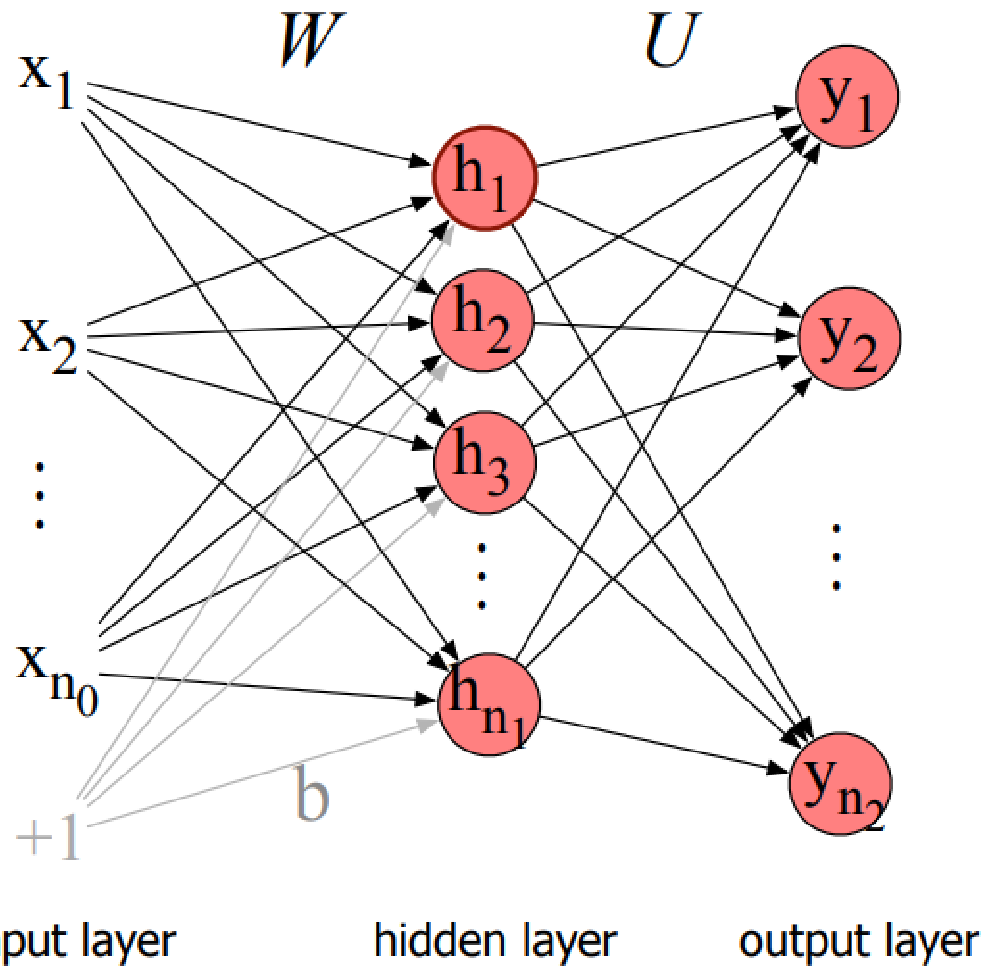
Feed-Forward Neural Networks

- A feed-forward network is a multilayer network in which the units are connected with no cycles.
- The outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.
- For historical reasons multilayer networks, especially feed-forward networks, are sometimes called **multi-layer perceptron** (or MLPs); this is a technical misnomer, since the units in modern multilayer networks aren't perceptrons (perceptrons are purely linear, but modern networks are made up of units with non-linearities like sigmoids), but at some point the name stuck.

Feed-Forward Neural Networks

- Simple feed-forward networks have three kinds of nodes: **input units, hidden units, and output units.**
- The input layer x is a vector of simple scalar values.
- The core of the neural network is the hidden layer h formed of hidden units h_i, each of which is a neural unit taking a weighted sum of its inputs and then applying a non-linearity.
- In the standard architecture, each layer is fully-connected, meaning that each unit in each layer takes as input the outputs from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers. Thus each hidden unit sums over all the input units.

Feed-Forward Neural Networks



A simple 2-layer feed-forward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Feed-Forward Neural Networks

- Recall that a single hidden unit has a weight vector and a bias as parameters.
- We represent the parameters for the entire hidden layer by combining the weight vector and bias for each unit i into a single weight matrix W and a single bias vector b for the whole layer (see the previous figure).
- Each element W_{ji} of the weight matrix W represents the weight of the connection from the ith input unit x_i to the jth hidden unit h_j.

Feed-Forward Neural Networks

- The advantage of using a single matrix W for the weights of the entire layer is that now the hidden layer computation for a feedforward network can be done very efficiently with simple matrix operations.
- In fact, the computation only has three steps:
  - multiplying the weight matrix by the input vector x,
  - adding the bias vector b,
  - applying the activation function g (such as the sigmoid, tanh, or ReLU activation function defined above).

Feed-Forward Neural Networks

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

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

- Notice that we're applying the σ function here to a vector.
- We're thus allowing $\sigma(\cdot)$, and indeed any activation function $g(\cdot)$, to apply to a vector element-wise, so

$$g[z1, z2, z3] \;=\; [g(z1), g(z2), g(z3)]$$

Feed-Forward Neural Networks

- Let's introduce some constants to represent the dimensionalities of these vectors and matrices.
- We'll refer to the input layer as layer 0 of the network, and have n_0 represent the number of inputs, so x is a vector of real numbers of dimension n_0, or more formally $x \in \mathbb{R}^{n_0}$, a column vector of dimensionality $[n_0, 1]$.
- Let's call the hidden layer layer 1 and the output layer layer 2. The hidden layer has dimensionality n_1, so $h \in \mathbb{R}^{n_1}$ and also $b \in \mathbb{R}^{n_1}$
- The weight matrix W has dimensionality $W \in \mathbb{R}^{n_1 \times n_0}$, i.e. $[n_1, n_0]$.

Feed-Forward Neural Networks

- The resulting value h (for *hidden* but also for *hypothesis*) forms a *representation* of the input.
- The role of the output layer is to take this new representation h and compute a final output.
- This output could be a real-valued number, but in many cases the goal of the network is to make some sort of classification decision, and so we will focus on the case of classification.

Feed-Forward Neural Networks

- If we are doing a binary task like sentiment classification, we might have a single output node, and its value y is the probability of positive versus negative sentiment.
- If we are doing multinomial classification, such as assigning a part-of-speech tag, we might have one output node for each potential part-of-speech, whose output value is the probability of that part-of-speech, and the values of all the output nodes must sum to one.
- The output layer is thus a vector y that gives a probability distribution across the output nodes.

Feed-Forward Neural Networks

- Like the hidden layer, the output layer has a weight matrix U , but some models don't include a bias vector b in the output layer, so we'll simplify by eliminating the bias vector in this example.
- The weight matrix is multiplied by its input vector (h) to produce the intermediate output z.

$$z = Uh$$

- There are n_2 output nodes, so $z \in \mathbb{R}^{n_2}$, weight matrix U has dimensionality $U \in \mathbb{R}^{n_2 \times n_1}$, and element U_{ij} is the weight from unit j in the hidden layer to unit i in the output layer.

Feed-Forward Neural Networks

- However, z can't be the output of the classifier, since it's a vector of real-valued numbers, while what we need for classification is a vector of probabilities.
- There is a convenient function for normalizing a vector of real values, by which we mean converting it to a vector that encodes a probability distribution (all the numbers lie between 0 and 1 and sum to 1).
- For a vector z of dimensionality d, the **softmax** is defined as:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=i}^{d} \exp(z_j)}, 1 \leq i \leq d$$

- Thus for example given a vector

$$z = [0.6,\ 1.1,\ -1.5,\ 1.2,\ 3.2,\ -1.1],$$

$\text{softmax}(z)$ is $[0.055,\ 0.090,\ 0.0067,\ \ 0.10,\ 0.74,\ 0.010]$.

Feed-Forward Neural Networks

- Softmax was exactly what is used to create a probability distribution from a vector of real-valued numbers in the multinomial version of logistic regression.
- A neural network classifier with one hidden layer as building a vector h which is a hidden layer representation of the input, and then running standard multinomial logistic regression on the features that the network develops in h.
  - By contrast, in multinomial logistic regression, the features were mainly designed by hand via feature templates.
- A neural network is like multinomial logistic regression, but
  - (a) with many layers, since a deep neural network is like layer after layer of logistic regression classifiers,
  - (b) with those intermediate layers having many possible activation functions (tanh, ReLU, sigmoid) instead of just sigmoid (although we'll continue to use σ for convenience to mean any activation function)
  - (c) rather than forming the features by feature templates, the prior layers of the network induce the feature representations themselves.

Feed-Forward Neural Networks

- Here are the final equations for a feed-forward 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:

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

- $x \in \mathbb{R}^{n_0}$, $h \in \mathbb{R}^{n_1}$, $b \in \mathbb{R}^{n_1}$, $W \in \mathbb{R}^{n_1 \times n_0}$, $U \in \mathbb{R}^{n_2 \times n_1}$
- We'll call this network a 2-layer network (we traditionally don't count the input layer when numbering layers,
- but do count the output layer). So by this terminology logistic regression is a 1-layer network.

Feed-Forward Neural Networks

- For the deeper networks of depth more than 2, we'll use superscripts in square brackets to mean layer numbers, starting at 0 for the input layer.
- So $W^{[1]}$ will mean the weight matrix for the (first) hidden layer, and $b^{[1]}$ will mean the bias vector for the (first) hidden layer. n_j will mean the number of units at layer j.
- We'll use $g(\cdot)$ to stand for the activation function, which will tend to be ReLU or tanh for intermediate layers and softmax for output layers.

Feed-Forward Neural Networks

- We'll use $a^{[i]}$ to mean the output from layer i, and $z^{[i]}$ to mean the combination of weights and biases $W^{[i]}a^{[i-1]} + b^{[i]}$.
- The 0th layer is for inputs, so we'll refer to the inputs x more generally as $a^{[0]}$.
- Thus we'll re-represent our 2-layer net as follows:

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]})$$
$$\hat{y} = a^{[2]}$$

Feed-Forward Neural Networks

- Note that with this notation, the equations for the computation done at each layer are the same.
- The algorithm for computing the forward step in an n-layer feedforward network, given the input vector $a^{[0]}$ is thus simply:

```
for $i$ in $1..n$
        $z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}$
        $a^{[i]} = g^{[i]}\left(z^{[i]}\right)$
$\hat{y} = a^{[n]}$
```

- The activation functions $g(\cdot)$ are generally different at the final layer. Thus $g^{[2]}$ might be softmax for multinomial classification or sigmoid for binary classification
- ReLU or tanh might be the activation function $g(\cdot)$ at the internal layers.

Feed-Forward Neural Networks

- One of the reasons we use nonlinear activation functions for each layer in a neural network is that if we did not, the resulting network is exactly equivalent to a single-layer network.
- Imagine the first two layers of such a network of purely linear layers:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$z^{[2]} = W^{[2]}z^{[1]} + b^{[2]}$$

- We can rewrite the function that the network is computing as:

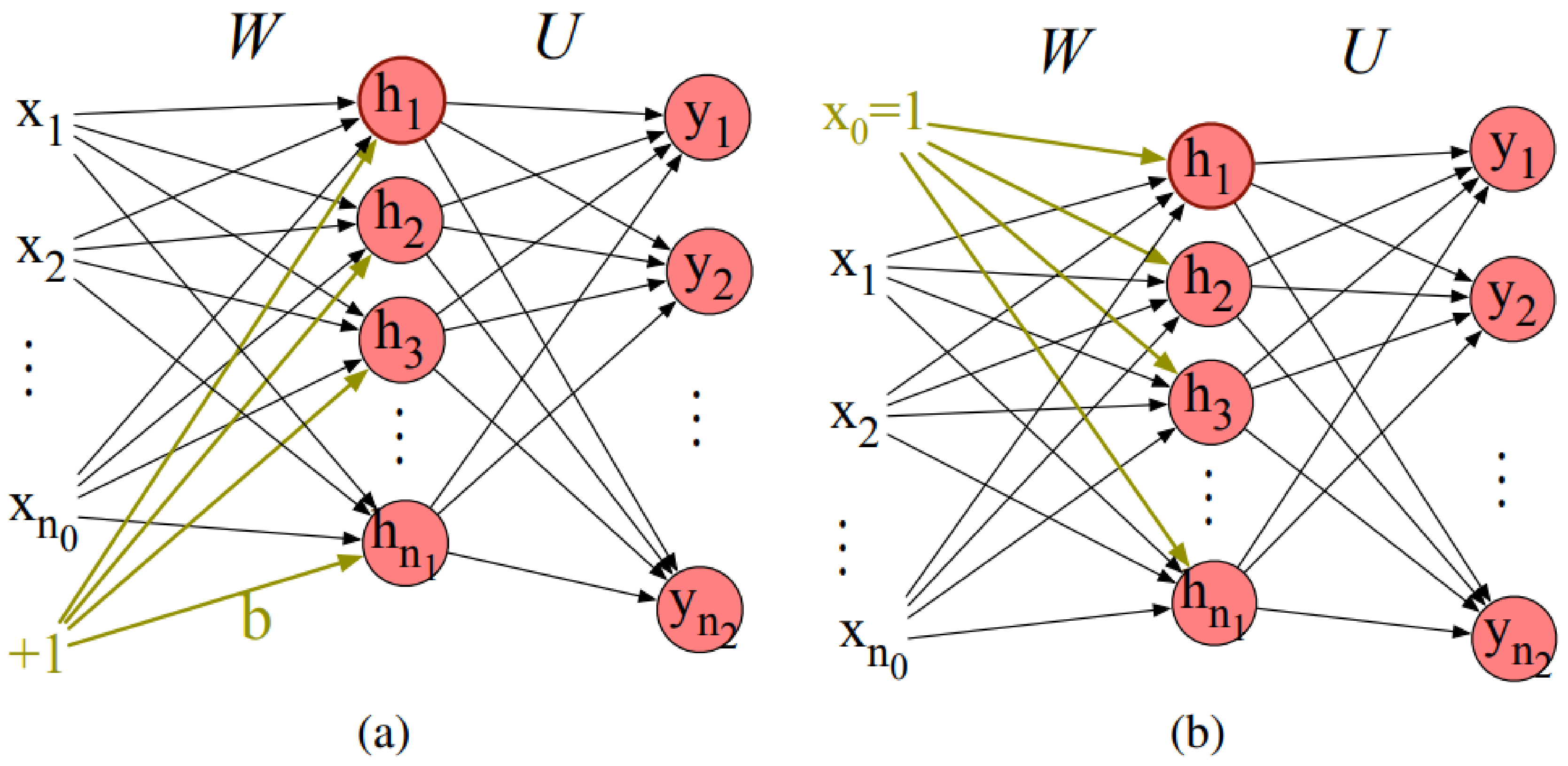
$$\begin{aligned} z^{[2]} &= W^{[2]}z^{[1]} + b^{[2]} \\ &= W^{[2]}\left(W^{[1]}x + b^{[1]}\right) + b^{[2]} \\ &= W^{[2]}W^{[1]}x + W^{[2]}b^{[1]} + b^{[2]} \\ &= W'x + b' \end{aligned}$$

- This generalizes to any number of layers. So without non-linear activation functions, a multilayer network is just a notational variant of a single layer network with a different set of weights, and we lose all the representational power of multilayer networks.

Feed-Forward Neural Networks

- In describing networks, we will often use a slightly simplified notation. Instead, we add a dummy node a_0 to each layer whose value will always be 1. Thus layer 0, the input layer, will have a dummy node $a_0^{[0]} = 1$, layer 1 will have $a_0^{[1]} = 1$, and so on.
- Instead of an equation like $h = \sigma(Wx + b)$, we'll use: $h = \sigma(Wx)$
- Instead of x having n_0 values: $x = x_1, \cdots, x_{n_0}$, it will have $n_0 + 1$ values, with a new 0th dummy value $x_0 = 1$: $x = x_0, \cdots, x_{n_0}$
- Instead of computing h_j as $h_j = \sigma\left(\sum_{i=1}^{n_0} W_{ji}x_i + b_j\right)$, we will have $h_j = \sigma\left(\sum_{i=0}^{n_0} W_{ji}x_i\right)$ where W_{j0} replaces what had been b_j

Feed-Forward Neural Networks



Replacing the bias node (shown in a) with x_0 (b).

Feedforward networks for NLP: Classification

- Let's see how to apply feedforward networks to NLP tasks! We'll first look at classification tasks like sentiment analysis.
- Let's begin with a simple 2-layer sentiment classifier.
- Imagine taking our logistic regression classifier, which corresponds to a 1-layer network, and just adding a hidden layer.
- The input element x_i could be scalar features, e.g., x_1 = count(words ∈ doc), x_2 = count(positive lexicon words ∈ doc), x_3 = 1 if "no" ∈ doc, and so on.
- And the output layer $\hat{y}$ could have two nodes (one each for positive and negative), or 3 nodes (positive, negative, neutral), in which case $\hat{y}_1$ would be the estimated probability of positive sentiment, $\hat{y}_2$ the probability of negative and $\hat{y}_3$ the probability of neutral.

Feedforward networks for NLP: Classification

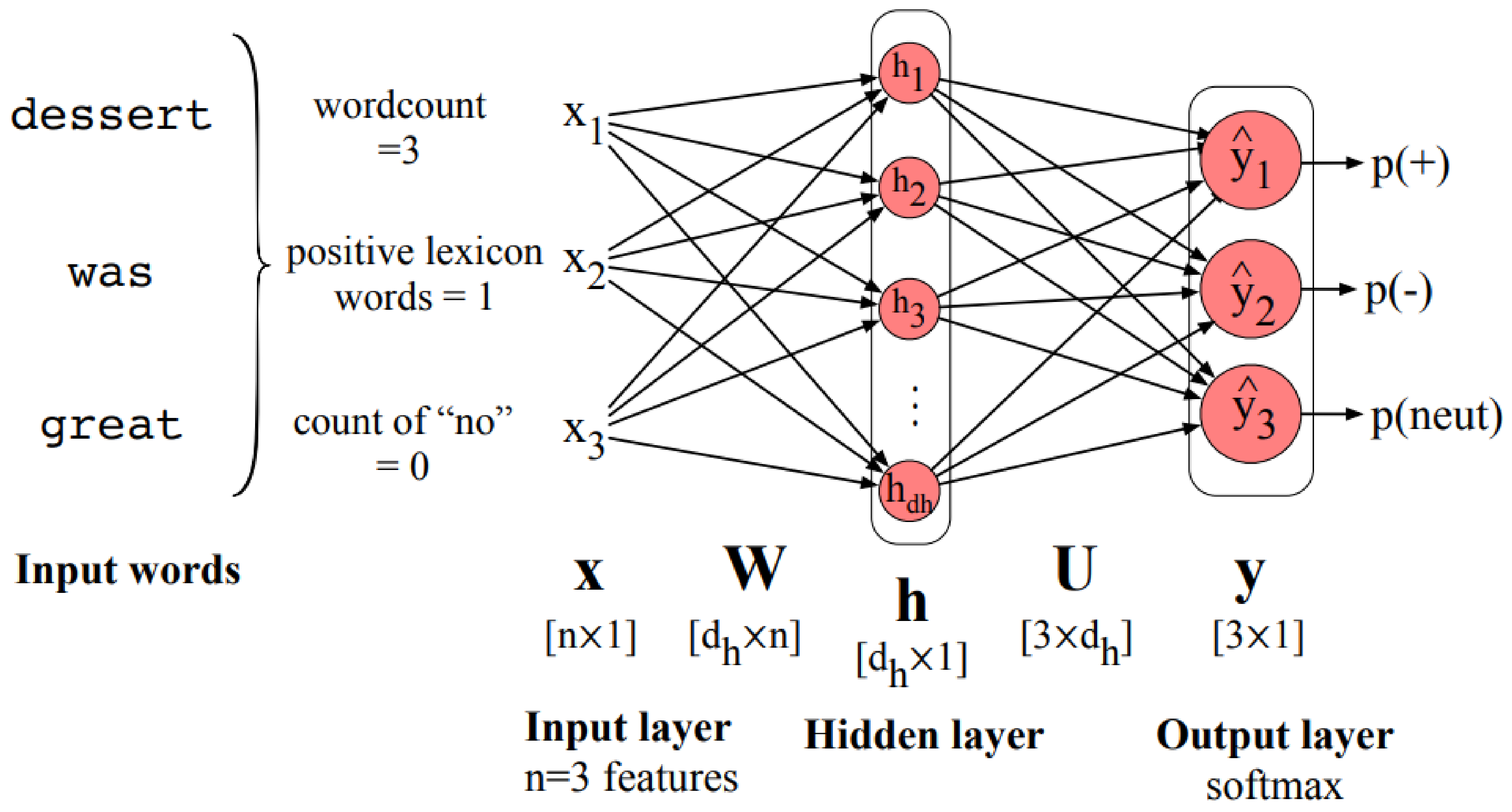
- The resulting equations would be just what we saw above for a 2-layer network (we'll continue to use the σ to stand for any non-linearity, whether sigmoid, ReLU or other).

$$\begin{aligned} x &= [x_1, x_2, \ldots x_N] \\ h &= \sigma(Wx + b) \\ z &= Uh \\ \hat{y} &= softmax(z) \end{aligned}$$

(each x_i is a hand-designed feature)

- A sketch of this architecture is shown in next slide.
- As mentioned earlier, adding this hidden layer to our logistic regression classifier allows the network to represent the non-linear interactions between features. This alone might give us a better sentiment classifier.

Feedforward networks for NLP: Classification

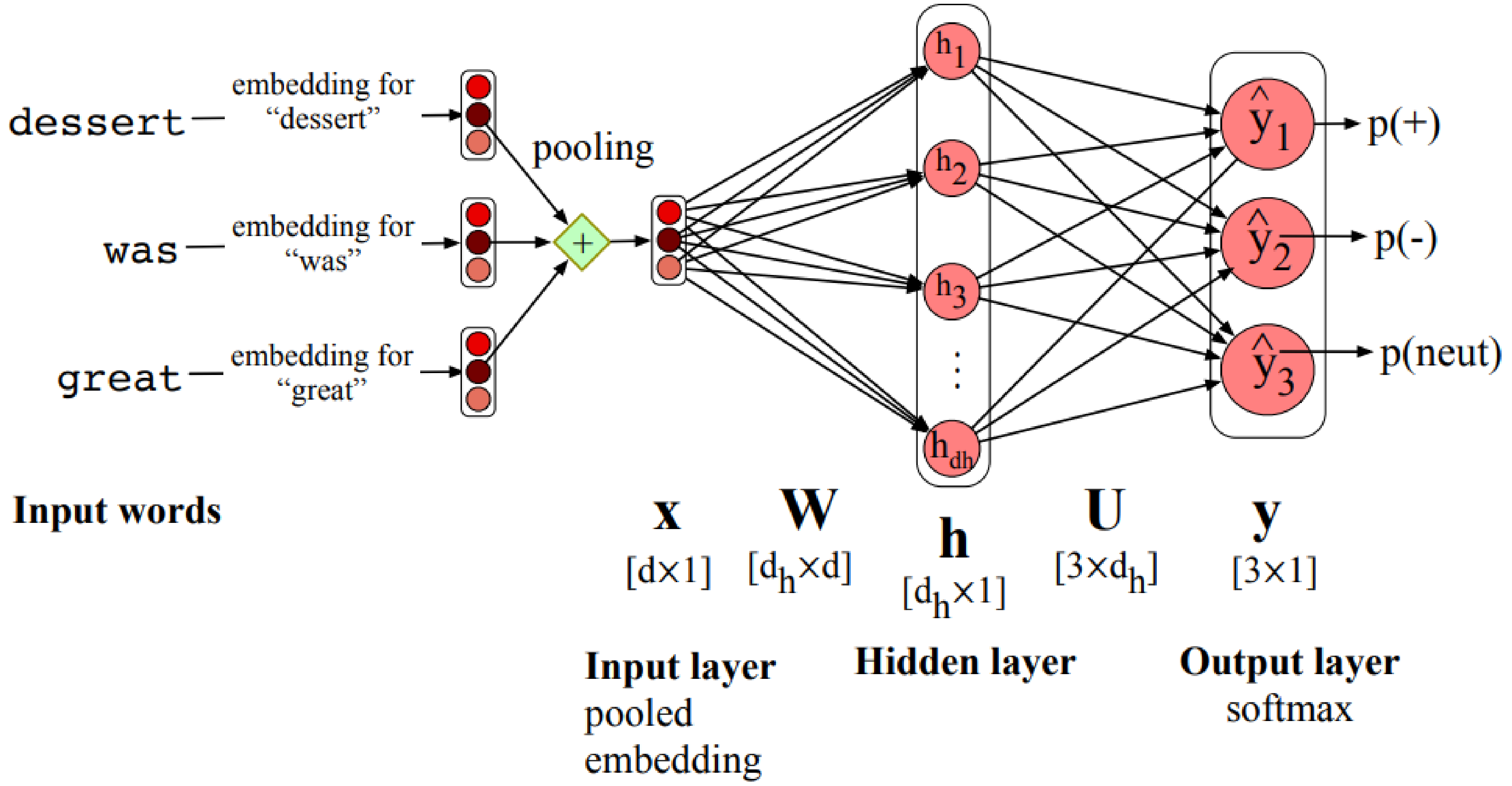


Feedforward network sentiment analysis using traditional hand-built features of the input text.

Feedforward networks for NLP: Classification

- However, most neural NLP applications do something different.
- Instead of using hand-built human-engineered features as the input to our classifier, we draw on deep learning’s ability to learn features from the data by representing words as embeddings, like the word2vec or GloVe embeddings.
- There are various ways to represent an input for classification.
- One simple baseline is to apply some sort of pooling function to the embeddings of all the words in the input.

Feedforward networks for NLP: Classification



Feedforward sentiment analysis using a pooled embedding of the input words.

Feedforward networks for NLP: Classification

- For example, for a text with n input words/tokens $w_1, \dots, w_n$, we can turn the n embeddings $e(w_1), \dots, e(w_n)$ (each of dimensionality d) into a single embedding also of dimensionality d by just summing the embeddings, or by taking their mean:

$$x_{mean} = \frac{1}{n} \sum_{i=1}^{n} e(w_i)$$

- There are many other options, like taking the element-wise max.
- Here are the equations for this classifier assuming mean pooling:

$$\begin{aligned} x &= \text{mean}(e(w_1), e(w_2), \dots, e(w_n)) \\ h &= \sigma(Wx + b) \\ z &= Uh \\ \hat{y} &= \text{softmax}(z) \end{aligned}$$

Feedforward networks for NLP: Classification

- We want to efficiently classify an entire test set of m examples.
- We do this by vectoring the process; instead of using for-loops to go through each example, we'll use matrix multiplication to do the entire computation of an entire test set at once.
- First, we pack all the input feature vectors for each input x into a single input matrix X, with each row i a row vector consisting of the pooled embedding for input example $x^{(i)}$. If the dimensionality of our pooled input embedding is d, X will be a matrix of shape $[m \times d]$.

Feedforward networks for NLP: Classification

- We will then need to slightly modify the equations.
- X is of shape $[m \times d]$ and W is of shape $[d_h \times d]$, so we'll have to reorder how we multiply X and W and transpose W so they correctly multiply to yield a matrix H of shape $[m \times d_h]$.
- The bias vector b of shape $[1 \times d_h]$ will now have to be replicated into a matrix of shape $[m \times d_h]$.
- We'll need to similarly reorder the next step and transpose U.
- Finally, our output matrix $\hat{Y}$ will be of shape $[m \times 3]$ (or more generally $[m \times d_o]$, where d_o is the number of output classes), with each row i of our output matrix $\hat{Y}$ consisting of the output vector $\hat{y}^{(i)}$.
- Here are the final equations for computing the output class distribution for an entire test set:

$$\begin{aligned} H &= \sigma(XW^T + b) \\ Z &= HU^T \\ \hat{Y} &= \text{softmax}(Z) \end{aligned}$$

Feedforward networks for NLP: Classification

- The idea of using word2vec or GloVe embeddings as our input representation—and more generally the idea of relying on another algorithm to have already learned an embedding representation for our input words—is called **pretraining**.
- Using pretrained embedding representations, whether simple static word embeddings like word2vec or some other much more powerful contextual embeddings, is one of the central ideas of deep learning.
- It’s also possible to train the word embeddings as part of an NLP task; we’ll talk about how to do this in a later section in the context of the neural language modeling task.

Feedforward Neural Language Modeling

- As our second application of feedforward networks, let’s consider **language modeling**: predicting upcoming words from prior word context.
- Neural language modeling is an important NLP task in itself, and it plays a role in many important algorithms for tasks like machine translation, summarization, speech recognition, grammar correction, and dialogue.
- We’ll describe simple feedforward neural language models, first introduced by Bengio et al. (2003).
- While modern neural language models use more powerful architectures like the recurrent nets or transformer networks, the feedforward language model introduces many of the important concepts of neural language modeling.

Feedforward Neural Language Modeling

- Neural language models have many advantages over the n-gram language models. Compared to n-gram models, neural language models
  - can handle much longer histories
  - can generalize better over contexts of similar words
  - are more accurate at word-prediction
- On the other hand, neural net language models
  - are much more complex
  - are slower and need more energy to train
  - are less interpretable than n-gram models
- So for many (especially smaller) tasks an n-gram language model is still the right tool.

Feedforward Neural Language Modeling

- A feedforward neural LM is a feedforward network that takes as input at time t a representation of some number of previous words (w_{t-1}, w_{t-2}, etc) and outputs a probability distribution over possible next words.
- Thus—like the n-gram LM—the feedforward neural LM approximates the probability of a word given the entire prior context $P(w_t|w_{1:t-1})$ by approximating based on the $N-1$ previous words:
$$P(w_t|w_1,\cdots,w_{t-1}) \approx P(w_t|w_{t-N+1},\cdots,w_{t-1})$$
- In the following examples we'll use a 4-gram example, so we'll show a net to estimate the probability
$$P(w_t = i|w_{t-1,}w_{t-2,}w_{t-3})$$

Feedforward Neural Language Modeling

- Neural language models represent words in this prior context by their embeddings, rather than just by their word identity as used in n-gram language models.
- Using embeddings allows neural language models to generalize better to unseen data.
- For example, suppose we've seen this sentence in training:

  I have to make sure that the cat gets fed.

  but we've never seen the word "gets fed" after the words "dog".
- Our test set has the prefix "I forgot to make sure that the dog gets". What's the next word?

Feedforward Neural Language Modeling

- An n-gram language model will predict “fed” after “that the cat gets”, but not after “that the dog gets”.
- But a neural LM knowing that “cat” and “dog” have similar embeddings, will be able to generalize from the “cat” context to assign a high enough probability to “fed” even after seeing “dog”.

Feedforward Neural Language Modeling

Forward inference in the neural language model

Let's walk through forward inference or decoding for neural language models.

- Forward inference is the task, given an input, of running a forward pass on the network to produce a probability distribution over possible outputs, in this case next words.
- We first represent each of the N previous words as a one-hot vector of length one-hot vector $|V|$, i.e., with one dimension for each word in the vocabulary.
- A one-hot vector is a vector that has one element equal to 1—in the dimension corresponding to that word's index in the vocabulary— while all the other elements are set to zero.

Feedforward Neural Language Modeling

Forward inference in the neural language model

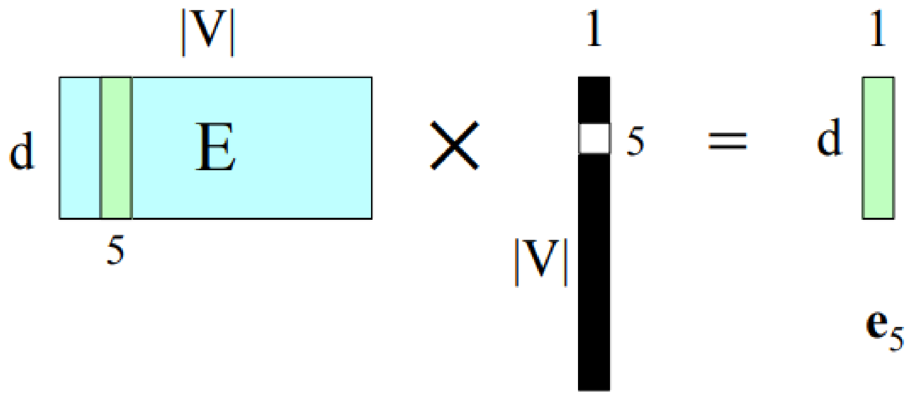
- Thus in a one-hot representation for the word "toothpaste", supposing it is V_5, i.e., index 5 in the vocabulary, $x_5 = 1$, and $x_i = 0 \; \forall i \neq 5$, as shown here:

[0	0	0	0	1	0	0	…	0	0	0	0]
1	2	3	4	5	6	7	…	…	…	…	|V|

Feedforward Neural Language Modeling

Forward inference in the neural language model

- The feedforward neural language model (see the sketch in next slide) has a moving window that can see N words into the past.
- The embedding weight matrix E has a column for each word, each a column vector of d dimensions, and hence has dimensionality $d \times |V|$.
- Multiplying by a one-hot vector that has only one non-zero element $x_i = 1$ simply selects out the relevant column vector for word i, resulting in the embedding for word i, as shown below.

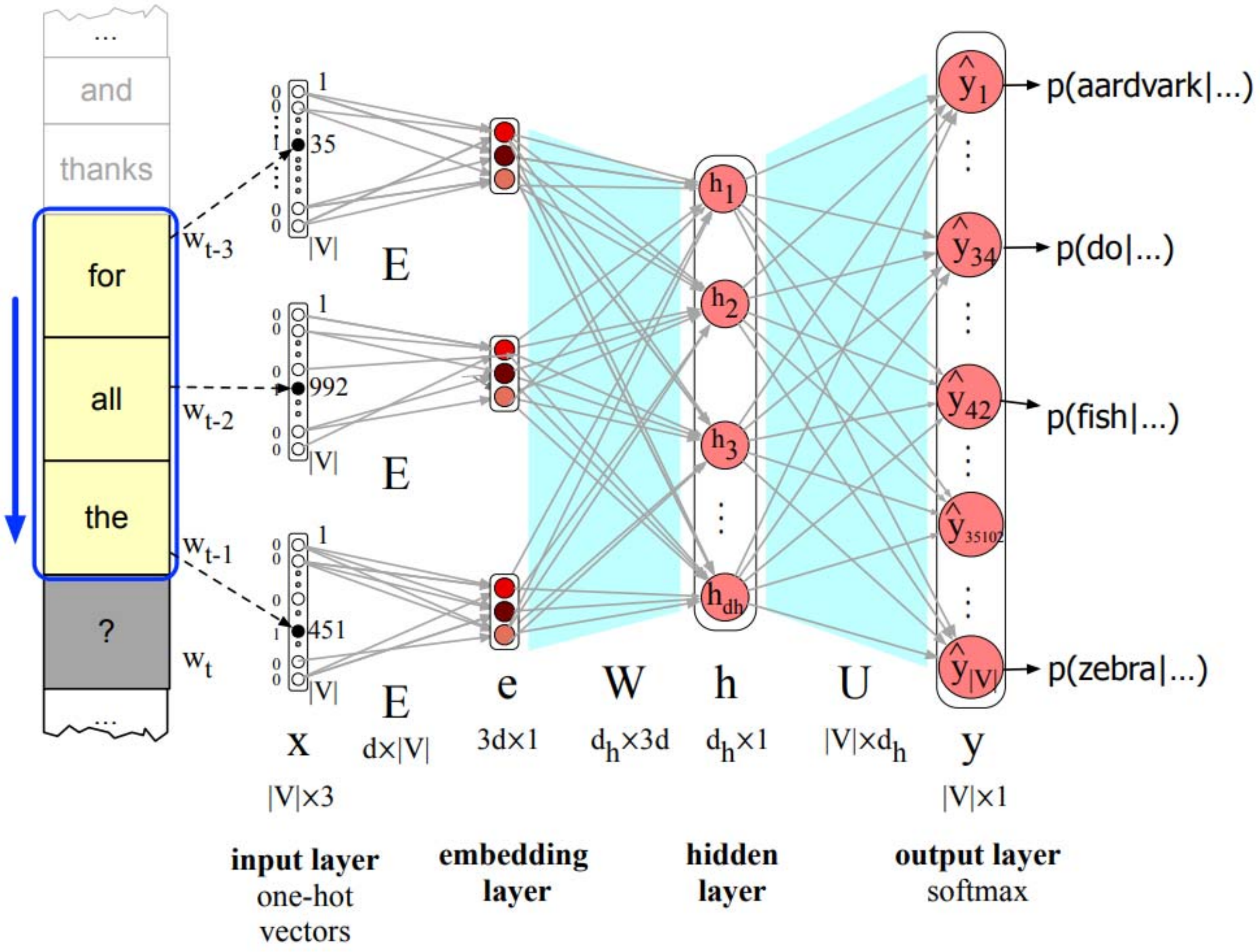


Selecting the embedding vector for word V_5 by multiplying the embedding matrix E with a one-hot vector with a 1 in index 5.

Feedforward Neural Language Modeling

Forward inference in the neural language model

Forward inference in a feedforward neural language model. At each timestep t the network computes a d-dimensional embedding for each context word (by multiplying a one-hot vector by the embedding matrix E), and concatenates the 3 resulting embeddings to get the embedding layer e.



The embedding vector e is multiplied by a weight matrix W and then an activation function is applied element-wise to produce the hidden layer h, which is then multiplied by another weight matrix U. Finally, a softmax output layer predicts at each node i the probability that the next word w_t will be vocabulary word V_i.

Feedforward Neural Language Modeling

Forward inference in the neural language model

- The 3 resulting embedding vectors are concatenated to produce e, the embedding layer. This is followed by a hidden layer and an output layer whose softmax produces a probability distribution over words.
- For example y_{42}, the value of output node 42, is the probability of the next word w_t being V_{42}, the vocabulary word with index 42 (which is the word ‘fish’ in our example).

Feedforward Neural Language Modeling

Forward inference in the neural language model

Here's the algorithm in detail for our mini example:

1. Select three embeddings from E:

- Given the three previous words, we look up their indices, create 3 one-hot vectors, and then multiply each by the embedding matrix *E*.
- Consider w_{t-3}. The one-hot vector for 'for' is (index 35) is multiplied by the embedding matrix E, to give the first part of the first hidden layer, the **embedding layer**.
- Since each column of the input matrix E is just an embedding for a word, and the input is a one-hot column vector x_i for word V_i, the embedding layer for input w will be $Ex_i = e_i$, the embedding for word i.
- We now concatenate the three embeddings for the context words to produce the embedding layer e.

Feedforward Neural Language Modeling
Forward inference in the neural language model

2. Multiply by W:

- We multiply by W (and add b) and pass through the ReLU (or other) activation function to get the hidden layer h.

3. Multiply by U:

- h is now multiplied by U

4. Apply softmax:

- After the softmax, each node i in the output layer estimates the probability $P(w_t = i \,|w_{t-1}, w_{t-2}, w_{t-3})$

Feedforward Neural Language Modeling

Forward inference in the neural language model

- In summary, the equations for a neural language model with a window size of 3, given one-hot input vectors for each input context word, are:

$$
\begin{aligned}
e &= [Ex_{t-3}; Ex_{t-2}; Ex_{t-1}] \\
\mathrm{h} &= \sigma(We + b) \\
z &= Uh \\
\hat{y} &= \mathrm{softmax}(z)
\end{aligned}
$$

- Note that we formed the embedding layer e by concatenating the 3 embeddings for the three context vectors; we'll often use semicolons to mean concatenation of vectors.

Training Neural Nets

- A feedforward neural net is an instance of supervised machine learning in which we know the correct output y for each observation x.
- What the system produces is $\hat{y}$, the system's estimate of the true y. The goal of the training procedure is to learn parameters $W^{[i]}$ and $b^{[i]}$ for each layer i that make $\hat{y}$ for each training observation as close as possible to the true y .

Training Neural Nets

- First, we’ll need a **loss function** that models the distance between the system output and the gold output, and it’s common to use the loss used for logistic regression, the **cross-entropy loss**.
- Second, to find the parameters that minimize this loss function, we’ll use the gradient descent optimization algorithm.

$$w^{t+1} = w^t - \eta \frac{d(f(x; w))}{dw}$$

where η is the learning rate

Training Neural Nets

- Third, gradient descent requires knowing the gradient of the loss function, the vector that contains the partial derivative of the loss function with respect to each of the parameters.
- How do we partial out the loss over all those intermediate layers?
- The answer is the algorithm called **error backpropagation** or **backward differentiation**.

Training Neural Nets

Loss Functions

- The cross-entropy loss that is used in neural networks is the same one for logistic regression.
- If the neural network is being used as a binary classifier, with the sigmoid at the final layer, the loss function is:

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

- If we are using the network to classify into 3 or more classes, the loss function is exactly the same as the loss for multinomial regression. Let’s summarize the explanation next.

Training Neural Nets

Loss Functions

- When we have more than 2 classes we'll need to represent both y and $\hat{y}$ as vectors.
- Assume we're doing *hard classification*, where only one class is the correct one. The true label y is then a vector with K elements, each corresponding to a class, with $y_c = 1$ if the correct class is c, with all other elements of y being 0.
- A vector like this, with one value = 1 and the rest 0, is called a **one-hot vector**. Now let $\hat{y}$ be the vector output from the network.
- And our classifier will produce an estimate vector with K elements $\hat{y}$, each element $\hat{y}_k$ of which represents the estimated probability $p(y_k = 1|x)$.

Training Neural Nets

Loss Functions

- The loss function for a single example x is the negative sum of the logs of the K output classes, each weighted by their probability y_k:

$$L_{CE}(\hat{y}, y) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

- We can simplify this equation further; let's first rewrite the equation using the function $\mathbb{1}\{\}$ which evaluates to 1 if the condition in the brackets is true and to 0 otherwise. This makes it more obvious that the terms in the sum will be 0 except for the term corresponding to the true class for which $y_k = 1$:

$$L_{CE}(\hat{y}, y) = -\sum_{k=1}^{K} \mathbb{1}\{y_k = 1\} \log \hat{y}_k$$

Training Neural Nets

Loss Functions

- In other words, the cross-entropy loss is simply the *negative log of the output probability corresponding to the correct class*, and we therefore also call this the **negative log likelihood loss**:
$$L_{CE}(\hat{y}, y) = -\log \hat{y}_c$$
where c is the correct class

- Plugging in the softmax formula $\widehat{y_c} = \mathrm{softmax}(z_c)$ with K the number of classes:
$$L_{CE}(\hat{y}, y) = -\log \frac{\exp(z_c)}{\sum_{j=1}^{K} \exp(z_j)}$$
where c is the correct class

Training Neural Nets

Computing the Gradient

- How do we compute the gradient of this loss function?
- Computing the gradient requires the partial derivative of the loss function with respect to each parameter.
- For a network with one weight layer and sigmoid output, we could simply use the derivative of the loss used for logistic regression.

$$\frac{\partial L_{CE}(\hat{y}, y)}{\partial w_j} = (\hat{y} - y)x_j$$
$$= (\sigma(w \cdot x + b) - y)x_j$$

Training Neural Nets

Computing the Gradient

- Or for a network with one hidden layer and softmax output (=multinomial logistic regression), we could use the derivative of the softmax loss (shown for a particular w_k and input x_i):

$$\frac{\partial L_{CE}(\hat{y}, y)}{\partial w_{k,i}} = -(y_k - \hat{y}_k)x_i$$
$$= -\big(y_k - p(y_k = 1|x)\big)x_i$$
$$= -\left(y_k - \frac{\exp(w_k \cdot x + b_k)}{\sum_{j=1}^{K} \exp(w_j \cdot x + b_j)}\right)x_i$$

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

Training Neural Nets

Computing the Gradient

- For deep networks, computing the gradients for each weight is much more complex, since we are computing the derivative with respect to weight parameters that appear all the way back in the very early layers of the network, even though the loss is computed only at the very end of the network.
- The solution to computing this gradient is an algorithm called **error backpropagation** or **backprop**.
- While backprop was invented specially for neural networks, it turns out to be the same as a more general procedure called backward differentiation, which depends on the notion of computation graphs.

Training Neural Nets

Computation Graphs

- A computation graph is a representation of the process of computing a mathematical expression, in which the computation is broken down into separate operations, each of which is modeled as a node in a graph.
- Consider computing the function

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

  If we make each of the component addition and multiplication operations explicit, and add names (d and e) for the intermediate outputs, the resulting series of computations is:

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

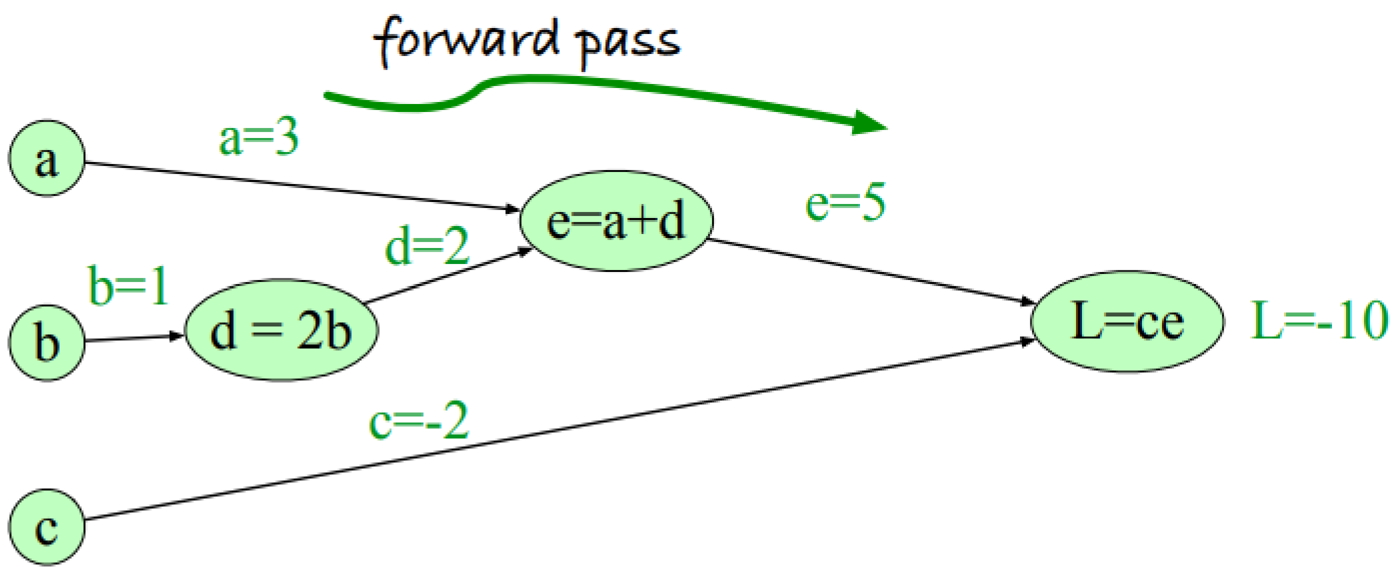
Training Neural Nets

Computation Graphs

- We can now represent this as a graph, with nodes for each operation, and directed edges showing the outputs from each operation as the inputs to the next, as in the following figure.
- The simplest use of computation graphs is to compute the value of the function with some given inputs.
- In the following figure, we've assumed the inputs $a = 3, b = 1, c = -2$, and we've shown the result of the forward pass to compute the result $L(3, 1, -2) = 10$.
  - In the forward pass of a computation graph, we apply each operation left to right, passing the outputs of each computation as the input to the next node.

Training Neural Nets

Computation Graphs



Computation graph for the function $L(a, b, c) = c(a + 2b)$, with values for input nodes $a = 3, b = 1, c = -2$, showing the forward pass computation of L.

Training Neural Nets

Backward differentiation on computation graphs

- 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.
- In this example our goal is to compute the derivative of the output function L with respect to each of the input variables, i.e., $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.
- The derivative $\frac{\partial L}{\partial a}$, tells us how much a small change in a affects L.

Training Neural Nets
Backward differentiation on computation graphs

- Backwards differentiation makes use of the chain rule in calculus. Suppose we are computing the derivative of a composite function $f(x) \ = \ u(v(x))$.
- The derivative of $f(x)$ is the derivative of $u(x)$ with respect to $v(x)$ times the derivative of $v(x)$ with respect to x:

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

Training Neural Nets

Backward differentiation on computation graphs

- The chain rule extends to more than two functions. If computing the derivative of a composite function $f(x) = u(v(w(x)))$, the derivative of $f(x)$ is:

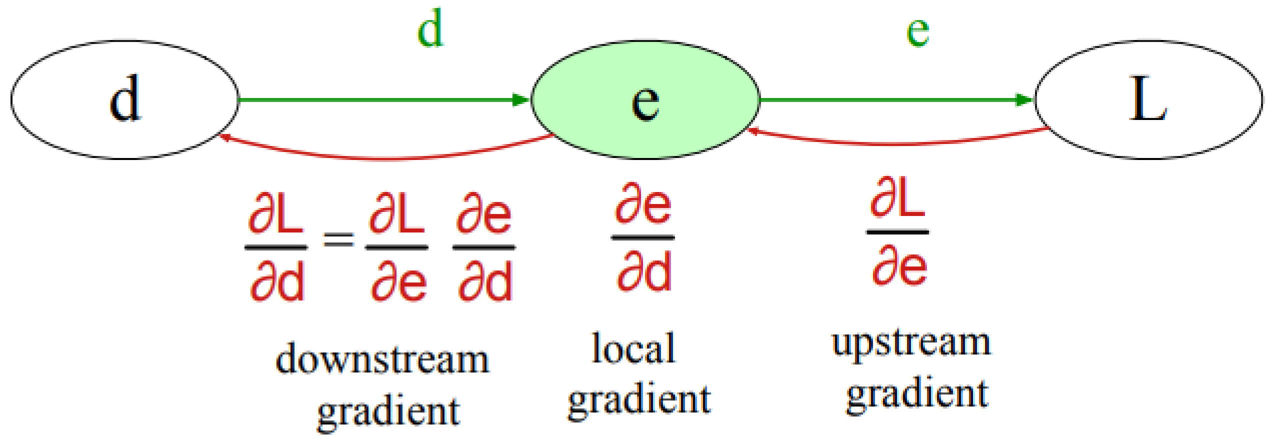
$$\frac{df}{dx} = \frac{du}{dv}\frac{dv}{dw}\frac{dw}{dx}$$

- The intuition of backward differentiation is to pass gradients back from the final node to all the nodes in the graph.

Training Neural Nets

Backward differentiation on computation graphs

- Each node takes an upstream gradient that is passed in from its parent node to the right, and for each of its inputs computes a local gradient (the gradient of its output with respect to its input), and uses the chain rule to multiply these two to compute a downstream gradient to be passed on to the next earlier node.


d
e
d
e
L
$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e} \frac{\partial e}{\partial d}$
$\frac{\partial e}{\partial d}$
$\frac{\partial L}{\partial e}$
downstream gradient
local gradient
upstream gradient

Each node (like *e* here) 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.

Training Neural Nets

Backward differentiation on computation graphs

- Let's now compute the 3 derivatives we need. Since in the computation graph $L = ce$, we can directly compute the derivative $\frac{\partial L}{\partial c}$:

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

- For the other two, we'll need to use the chain rule:

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

Training Neural Nets

Backward differentiation on computation graphs

- Thus five intermediate derivatives are required: $\frac{\partial L}{\partial e}, \frac{\partial L}{\partial c}, \frac{\partial e}{\partial a}, \frac{\partial e}{\partial d}$, and $\frac{\partial d}{\partial b}$, which are as follows (making use of the fact that the derivative of a sum is the sum of the derivatives):

$$L{=}ce: \qquad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e{=}a{+}d: \qquad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b: \qquad \frac{\partial d}{\partial b} = 2$$

Training Neural Nets

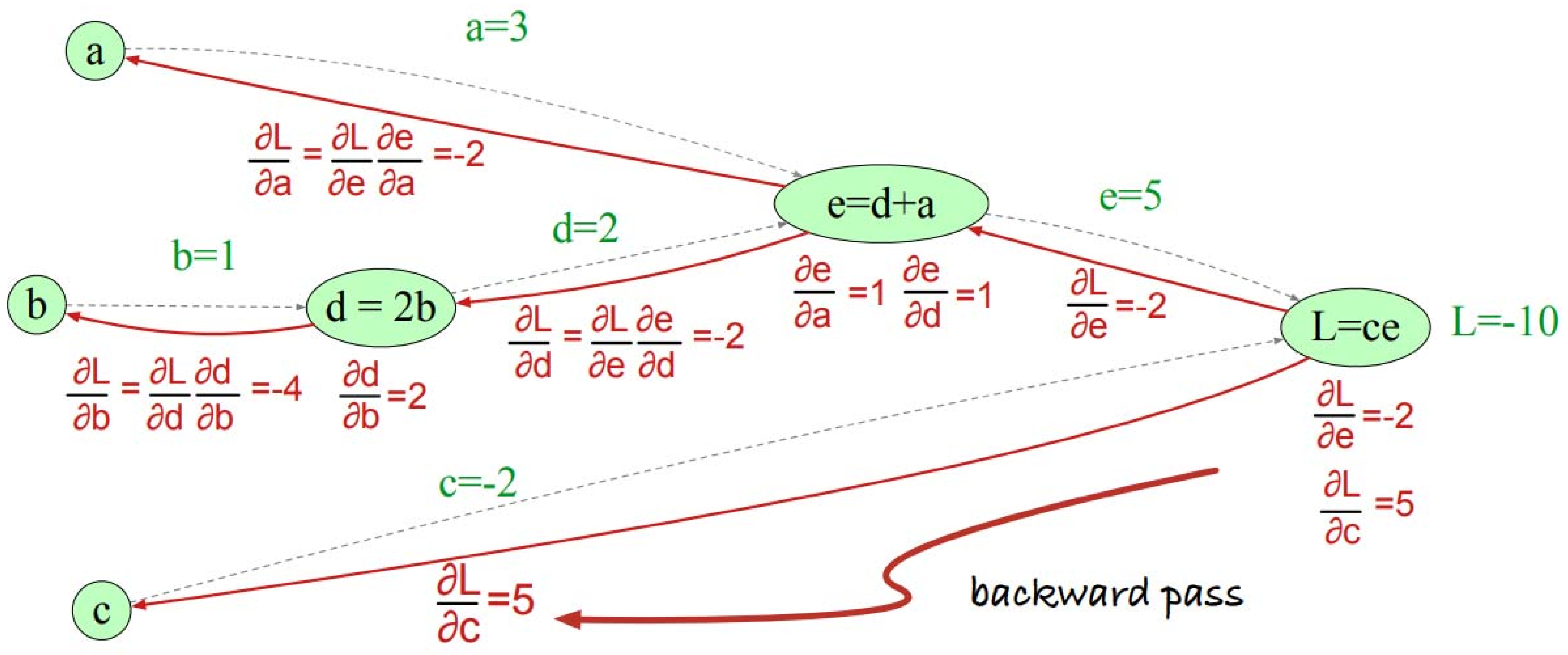
Backward differentiation on computation graphs

- In the backward pass, we compute each of these partials along each edge of the graph from right to left, using the chain rule just as we did above.
    - We begin by computing the downstream gradients from node L, which are $\frac{\partial L}{\partial e}$ and $\frac{\partial L}{\partial c}$.
    - For node e, we then multiply this upstream gradient $\frac{\partial L}{\partial e}$ by the local gradient (the gradient of the output with respect to the input), $\frac{\partial e}{\partial d}$ to get the output we send back to node d: $\frac{\partial L}{\partial d}$.
    - And so on, until we have annotated the graph all the way to all the input variables.
- The forward pass conveniently already will have computed the values of the forward intermediate variables we need (like d and e) to compute these derivatives.

Training Neural Nets

Backward differentiation on computation graphs

- The following diagram shows the backward pass.



a
a=3
$\frac{\partial L}{\partial a}=\frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$=-2
e=d+a
e=5
d=2
b=1
b
d = 2b
$\frac{\partial e}{\partial a}$=1 $\frac{\partial e}{\partial d}$=1
$\frac{\partial L}{\partial e}$=-2
L=ce
L=-10
$\frac{\partial L}{\partial d}=\frac{\partial L}{\partial e}\frac{\partial e}{\partial d}$=-2
$\frac{\partial L}{\partial b}=\frac{\partial L}{\partial d}\frac{\partial d}{\partial b}$=-4
$\frac{\partial d}{\partial b}$=2
$\frac{\partial L}{\partial e}$=-2
$\frac{\partial L}{\partial c}$=5
c=-2
c
$\frac{\partial L}{\partial c}$=5
backward pass

Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

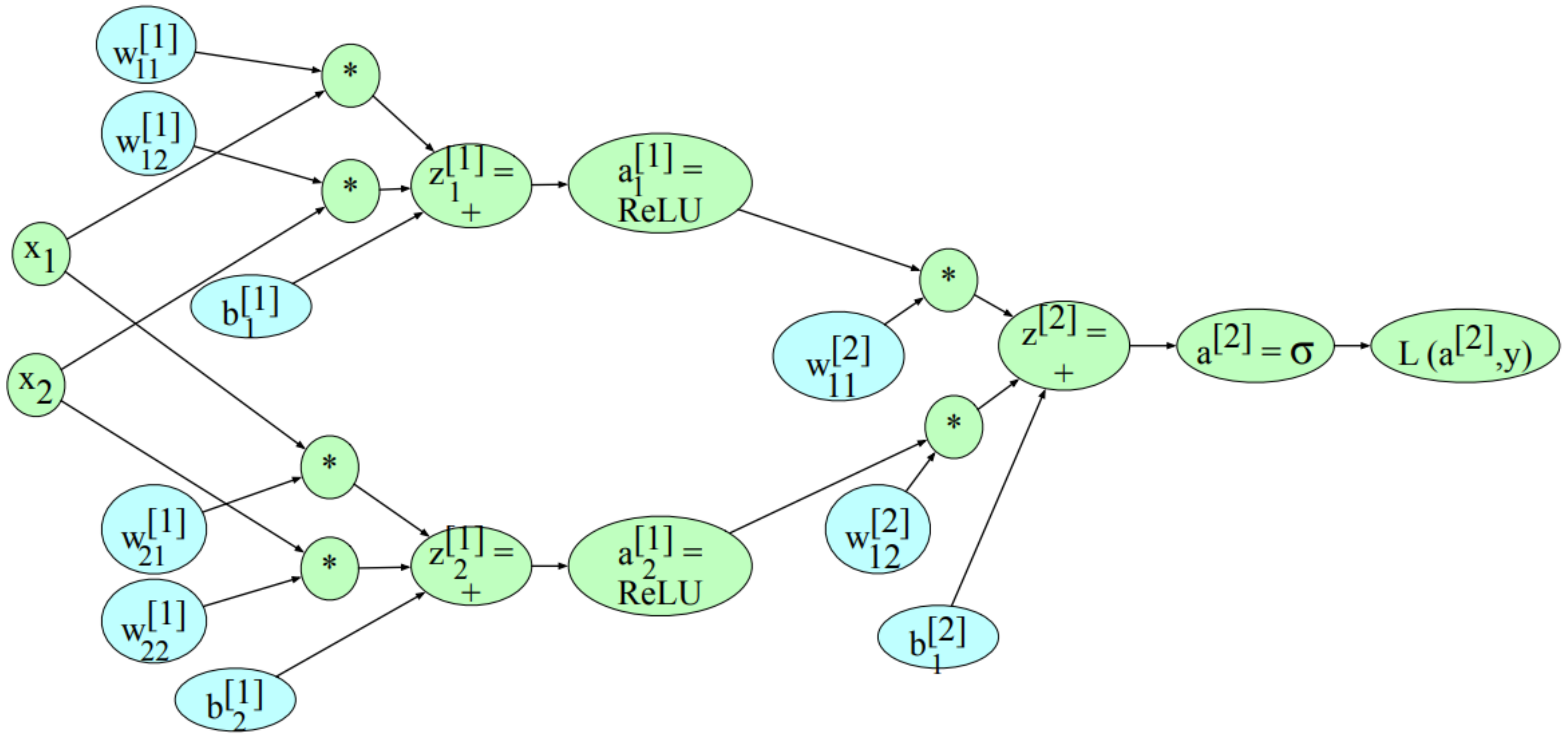
Training Neural Nets

Backward differentiation for a neural network

- Of course computation graphs for real neural networks are much more complex.
- This following figure shows a sample computation graph for a 2-layer neural network with $n_0 = 2, n_1 = 2$, and $n_2 = 1$, assuming binary classification and hence using a sigmoid output unit for simplicity.

Training Neural Nets

Backward differentiation on computation graphs



Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input units and 2 hidden units. We've adjusted the notation a bit to avoid long equations in the nodes by just mentioning the function that is being computed, and the resulting variable name. Thus the $*$ to the right of node $w_{11}^{[1]}$ means that $w_{11}^{[1]}$ is to be multiplied by x_1, and the node $z^{[1]} = +$ means that the value of $z^{[1]}$ is computed by summing the three nodes that feed into it (the two products, and the bias term $b_i^{[1]}$)

Training Neural Nets

Backward differentiation for a neural network

- The function that the computation graph is computing is:

$$\begin{aligned} z^{[1]} &= W^{[1]}x + b^{[1]} \\ a^{[1]} &= \mathrm{ReLU}(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned}$$

- For the backward pass we'll also need to compute the loss L. The loss function for binary sigmoid output is

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

- Our output $\hat{y} = a^{[2]}$, so we can rephrase this as

$$L_{CE}(a^{[2]}, y) = -\left[y \log a^{[2]} + (1-y)\log\left(1-a^{[2]}\right)\right]$$

- The weights that need updating (those for which we need to know the partial derivative of the loss function) are shown in teal.

Training Neural Nets

Backward differentiation for a neural network

- In order to do the backward pass, we'll need to know the derivatives of all the functions in the graph.
  The derivative of the sigmoid σ:
$$\frac{d\sigma(z)}{dz} = \sigma(z)\big(1 - \sigma(z)\big)$$
- We'll also need the derivatives of each of the other activation functions. The derivative of tanh is:
$$\frac{d\tanh(z)}{dz} = 1 - \tanh^2(z)$$
- The derivative of the ReLU is

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0\,, for\ x < 0 \\ 1\,, for\ x \geq 0 \end{cases}$$

Training Neural Nets

Backward differentiation for a neural network

- We'll give the start of the computation, computing the derivative of the loss function L with respect to z, or $\frac{\partial L}{\partial z}$ (and leaving the rest of the computation as an exercise for the reader). By the chain rule:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z}$$

- So let's first compute $\frac{\partial L}{\partial a^{[2]}}$:

$$L_{CE}\left(a^{[2]}, y\right) = -\left[y \log a^{[2]} + (1-y) \log\left(1 - a^{[2]}\right)\right]$$

$$\frac{\partial L}{\partial a^{[2]}} = -\left(\left(y \frac{\partial \log\left(a^{[2]}\right)}{\partial a^{[2]}}\right) + (1-y) \frac{\partial \log\left(1 - a^{[2]}\right)}{\partial a^{[2]}}\right)$$

$$= -\left(\left(y \frac{1}{a^{[2]}}\right) + (1-y) \frac{1}{1 - a^{[2]}}(-1)\right)$$

$$= -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1 - a^{[2]}}\right)$$

Training Neural Nets

Backward differentiation for a neural network

- Next, by the derivative of the sigmoid:

$$\frac{\partial a^{[2]}}{\partial z} = a^{[2]}(1 - a^{[2]})$$

- Finally, we can use the chain rule:

$$\begin{aligned}\frac{\partial L}{\partial z} &= \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z} \\ &= -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}}\right) \mathrm{a}^{[2]}(1 - \mathrm{a}^{[2]}) \\ &= a^{[2]} - y\end{aligned}$$

- Continuing the backward computation of the gradients (next by passing the gradients over $b_1^{[2]}$and the two product nodes, and so on, back to all the teal nodes), is left as an exercise for the reader.

Training Neural Nets

More Details on Learning

- Optimization in neural networks is a non-convex optimization problem
- We need to initialize the weights with small random numbers.
- It's also helpful to normalize the input values to have 0 mean and unit variance.
- Various forms of regularization are used to prevent overfitting.
- One of the most important is **dropout**: randomly dropping some units and their connections from the network during training

Training Neural Nets

More Details on Learning

- The parameters of a neural network are the weights W and biases b; those are learned by gradient descent.
- The hyperparameters are things that are set by the algorithm designer
  - Hyperparameters include the learning rate η, the minibatch size, the model architecture (the number of layers, the number of hidden nodes per layer, the choice of activation functions), how to regularize, etc.
  - Optimal values are tuned on a devset (development set) rather than by gradient descent learning on the training set.

Training Neural Nets

More Details on Learning

- Gradient descent itself also has many architectural variants such as Adam.
- Most modern neural networks are built using computation graph formalisms that make it easy and natural to do gradient computation and parallelization on vector-based GPUs (Graphic Processing Units).
- PyTorch and TensorFlow are two of the most popular.

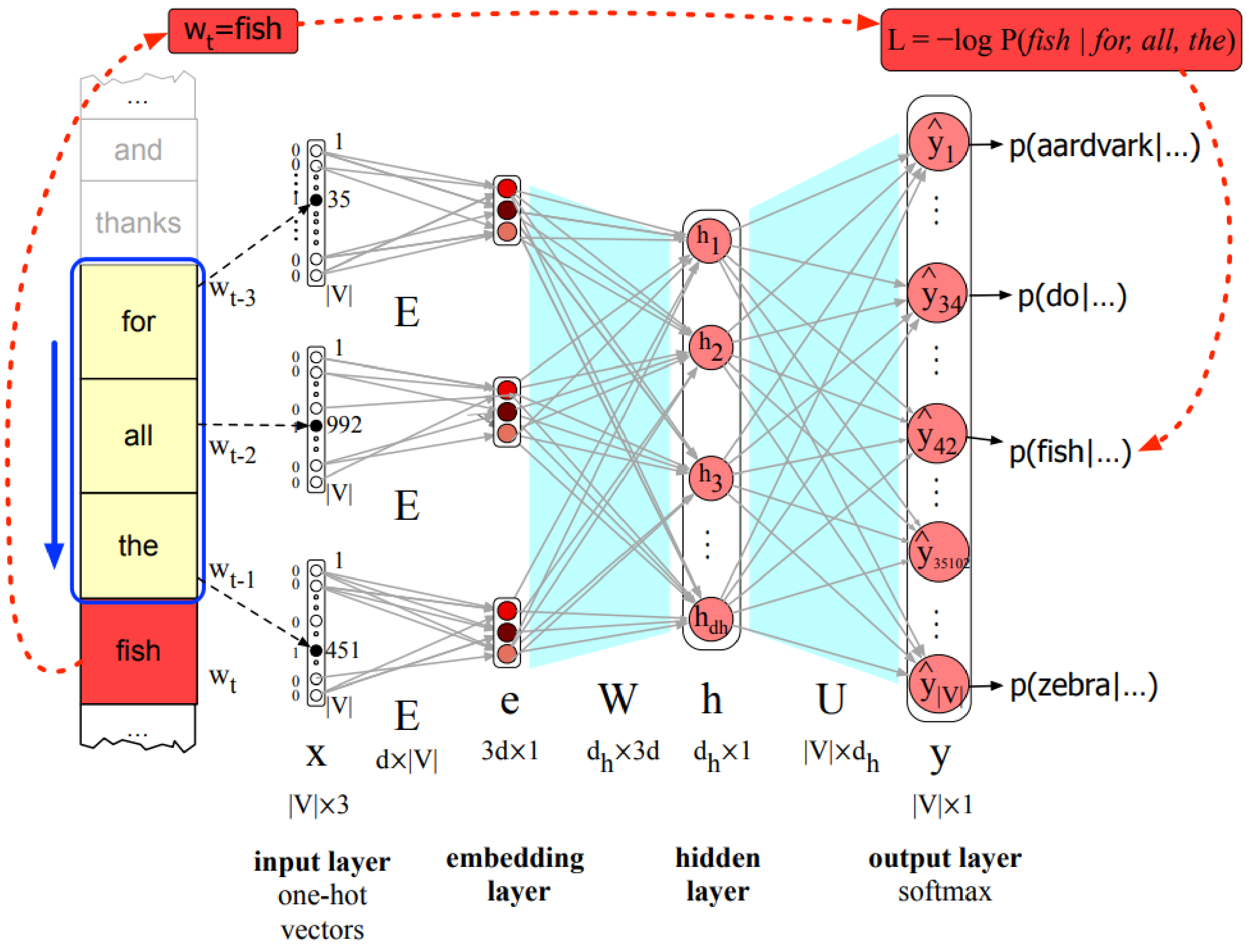
Training the neural language model

- Now that we've seen how to train a generic neural net, let's talk about the architecture for training a neural language model, setting the parameters $\theta = E, W, U, b$.
- For some tasks, it's ok to freeze the embedding layer E with initial word2vec values.
    - Freezing means we use word2vec or some other pretraining algorithm to compute the initial embedding matrix E, and then hold it constant while we only modify W, U, and b, i.e., we don't update E during language model training.
- However, often we'd like to learn the embeddings simultaneously with training the network.
- This is useful when the task the network is designed for (like sentiment classification, translation, or parsing) places strong constraints on what makes a good representation for words.

Training the neural language model

- To train the model, i.e. to set all the parameters $\theta = E, W, U, b$, we do gradient descent, using error back propagation on the computation graph to compute the gradient.
- Training thus not only sets the weights W and U of the network, but also as we're predicting upcoming words, we're learning the embeddings E for each words that best predict upcoming words.
- The following figure shows the set up for a window size of $N = 3$ context words.

Training the neural language model



Learning all the way back to embeddings. Again, the embedding matrix E is shared among the 3 context words.

Training the neural language model

- The input x consists of 3 one-hot vectors, fully connected to the embedding layer via 3 instantiations of the embedding matrix E.
- We don't want to learn separate weight matrices for mapping each of the 3 previous words to the projection layer. We want one single embedding dictionary E that's shared among these three.
- That's because over time, many different words will appear as w_{t-2} or w_{t-1}, and we'd like to just represent each word with one vector, whichever context position it appears in.
- Recall that the embedding weight matrix E has a column for each word, each a column vector of d dimensions, and hence has dimensionality $d \times |V|$.

Training the neural language model

- Generally training proceeds by taking as input a very long text, concatenating all the sentences, starting with random weights, and then iteratively moving through the text predicting each word w_t
- At each word w_t, the cross-entropy (negative log likelihood) loss is:

$$L_{CE}(\hat{y}, y) = -\log \hat{y_i} \quad \text{(where } i \text{ is the correct class)}$$

- For language modelling, the classes are the word in the vocabulary, so $\hat{y_i}$ here means the probability that the model assigns to the correct next word w_t:

$$L_{CE} = -\log p(w_t | w_{t-1}, \cdots, w_{t-n+1})$$

- The parameter update for stochastic gradient descent for this loss from step s to $s+1$ is then:

$$\theta^{s+1} = \theta^s - \eta \frac{\partial -\log p(w_t | w_{t-1}, \ldots, w_{t-n+1})}{\partial \theta}$$

Training the neural language model

- This gradient can be computed in any standard neural network framework which will then backpropagate through $\theta = E, W, U, b$.
- Training the parameters to minimize loss will result both in an algorithm for language modeling (a word predictor) but also a new set of embeddings E that can be used as word representations for other tasks.