Natural Language Processing CS 6320 Lecture 6 Neural Language Models

Instructor: Sanda Harabagiu

In this lecture

- *We shall cover:*

➢ *Deep Neural Models for Natural Language Processing*

  ❑ *Introduce Feed Forward Networks*

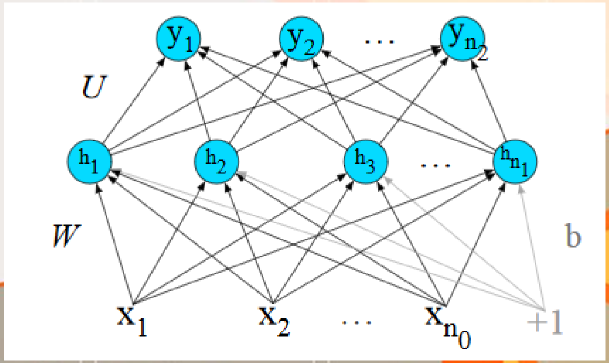
  ❑ *Use them to build a neural language model*

Neural Networks

- *Neural networks are an essential computational tool for language processing, and a very old one. They are called neural because their origins lie in the McCulloch-Pitts neuron (McCulloch and Pitts, 1943), 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.*

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. In this lecture we discuss a neural net applied to classification. The neural architecture we focus on 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).*



Neurons

- *The building block of a neural network is a single computational unit (neuron). 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**. Thus given a set of inputs $x_1 ... x_n$, a unit has a set of corresponding weights $w_1 ... w_n$ and a bias b, so the weighted sum z can be represented as:

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

Or in vector notation:

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

Activation functions

Instead of using z (a linear function of x) as the output of the neural unit, it is better to apply a non-linear function f to z and use the result as the output of the neural unit. We will refer to the output of this function as the activation value for the unit, a.

Let us refer to the final output of the neural network as y. If the neural network has only one unit, then:

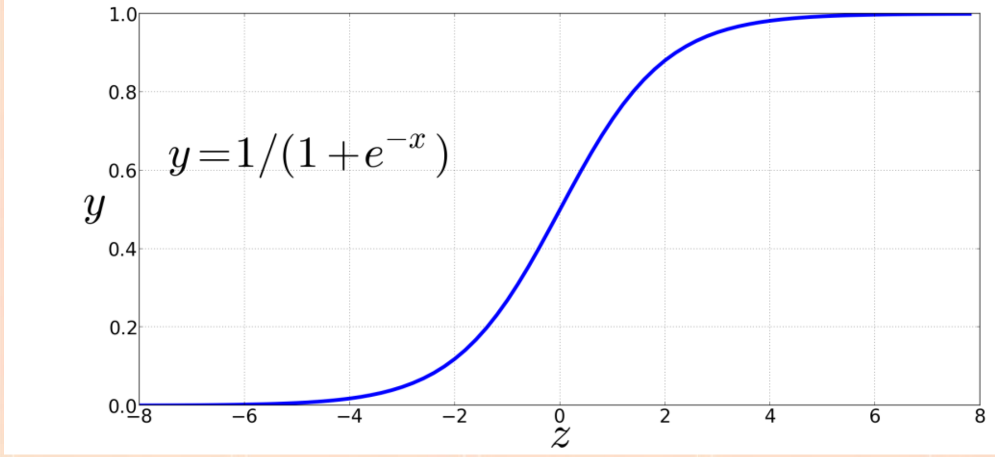
$$y = a = f(z)$$

EXAMPLE of three popular non-linear functions *f():*

1. the sigmoid,
2. The tanh, and
3. the rectified linear unit (ReLU)

The sigmoid function

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



The sigmoid function takes a real value and maps it to the range [0,1]. Because it is nearly linear around 0 but has a sharp slope toward the ends, it tends to squash outlier values toward 0 or 1.

How do we use it in a neural network?

$$y = a = f(z) \quad \Rightarrow \quad y = \sigma(w \cdot x + b) = \frac{1}{1+exp(-(w \cdot x + b))}$$

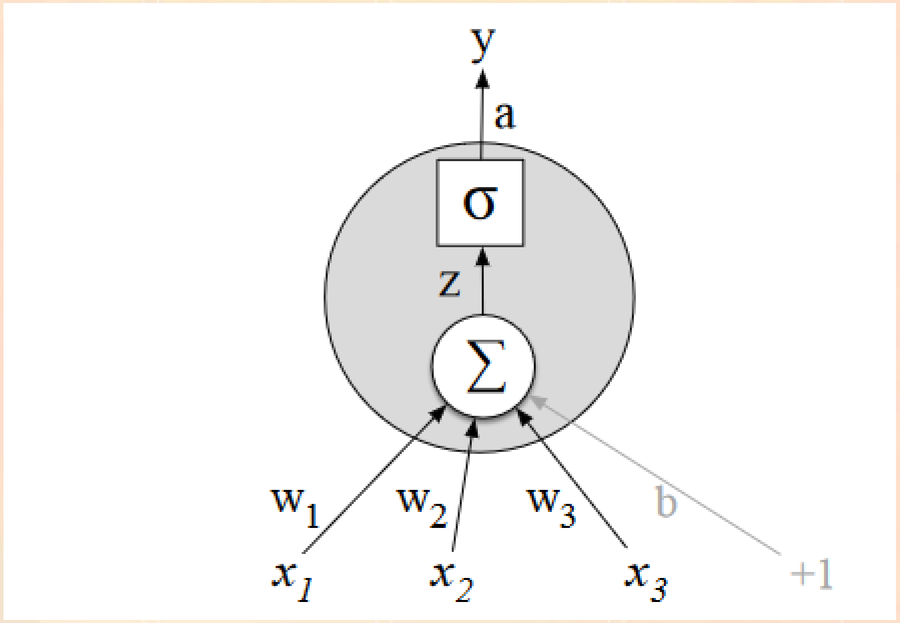
Neural Cell with Sigma function

- *EXAMPLE*

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

with input vector:

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



What is the output????

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

NOTE: I*n practice, the sigmoid is not commonly used as an activation function. This is because very high values of z result in values of y that are* ***saturated*** *i.e., extremely close to 1, which causes problems for learning.*

The tanh and ReLU activation functions

- *tanh is a variant of the sigmoid that ranges from -1 to +1:*

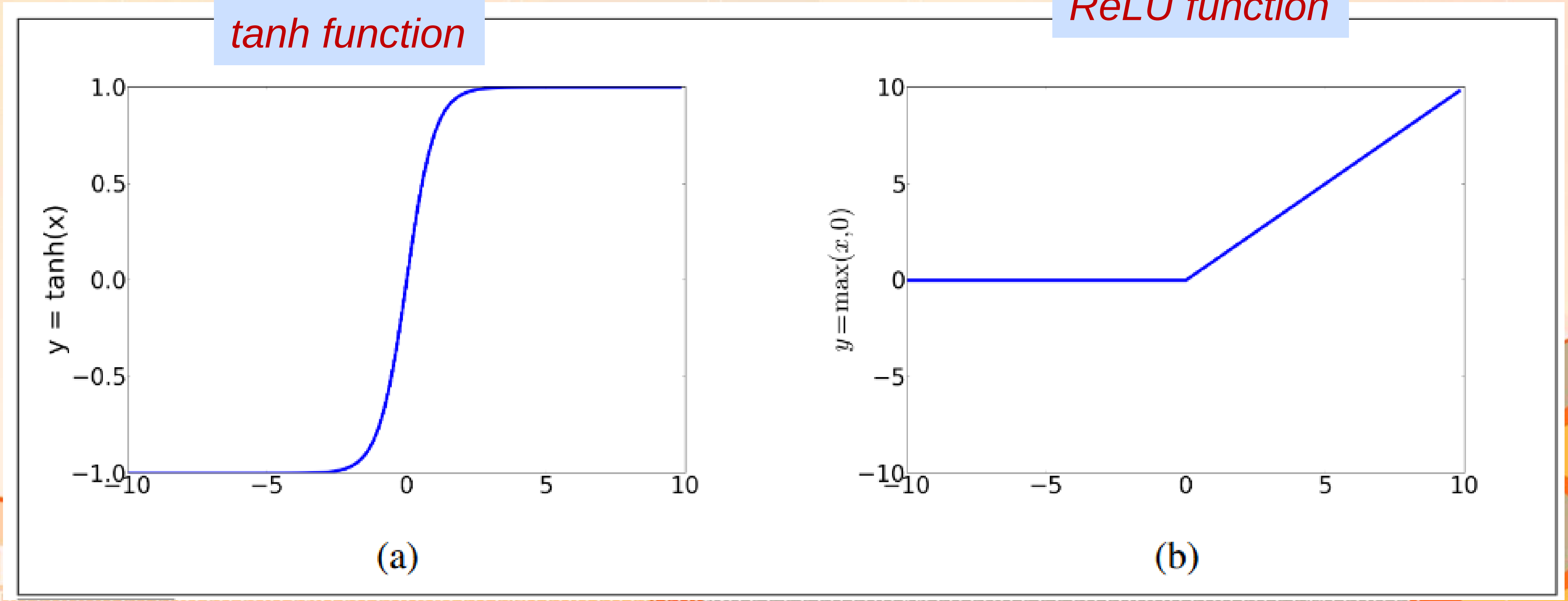
$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- *The simplest activation function, and perhaps the most commonly used, is the rectified linear unit, also called the ReLU. It is defined as :*

$$y = max(x, 0)$$

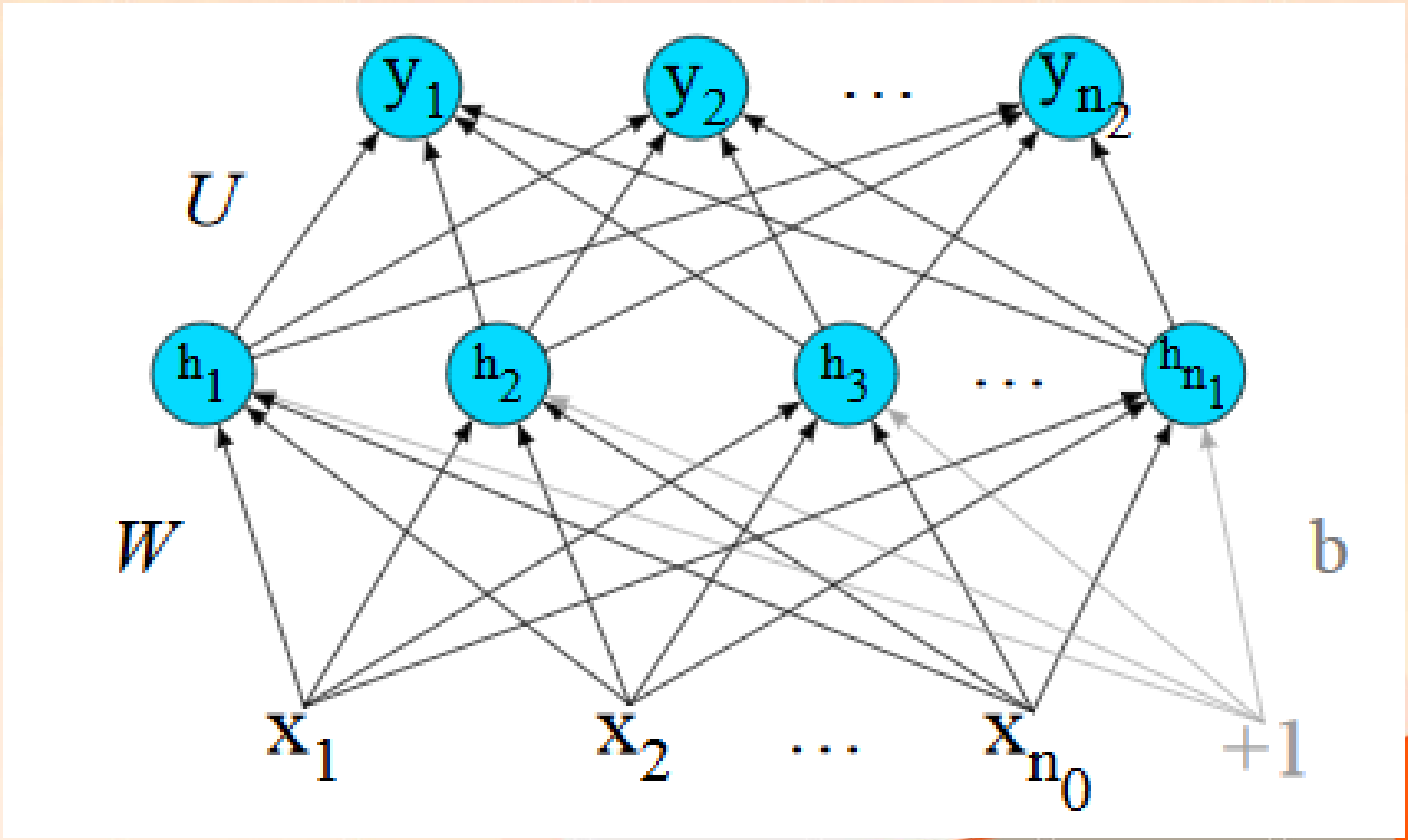
tanh function

ReLU function



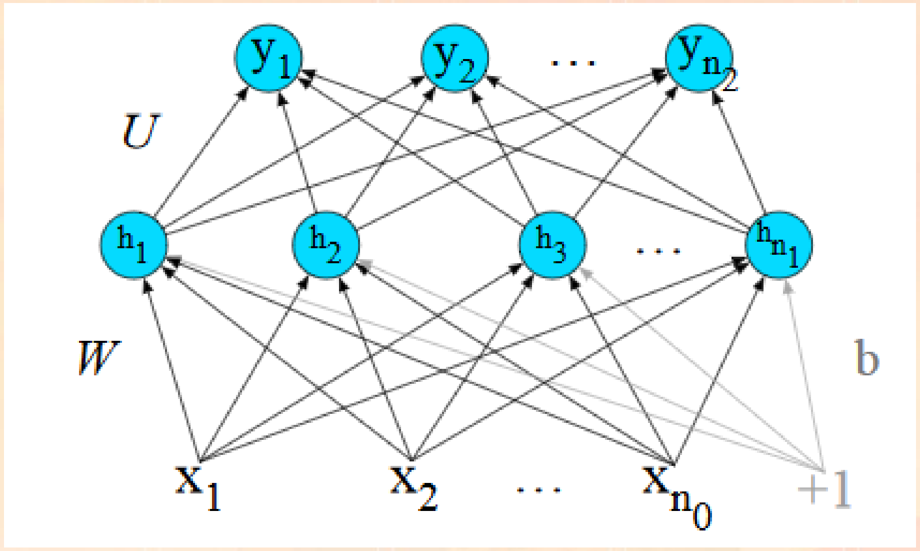
Feed-Forward Neural Networks

A feed-forward network is a multilayer neural 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.



Units in Feed-Forward Neural Networks

- *Simple feed-forward networks have three kinds of nodes: input units, hidden units, and output units.*



In the standard architecture, <u>each layer is fully-connected</u> , meaning that each unit in each layer takes as input the outputs fully-connected from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers.

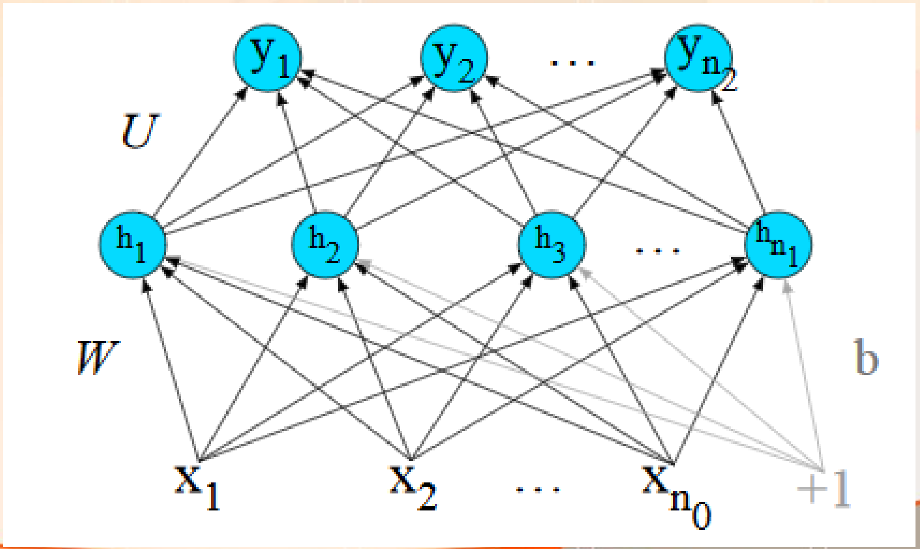
Thus each hidden unit sums over all the input units.

Neural Network parameters

- *Recall that a single hidden unit has parameters w (the weight vector) and b (the bias scalar).*

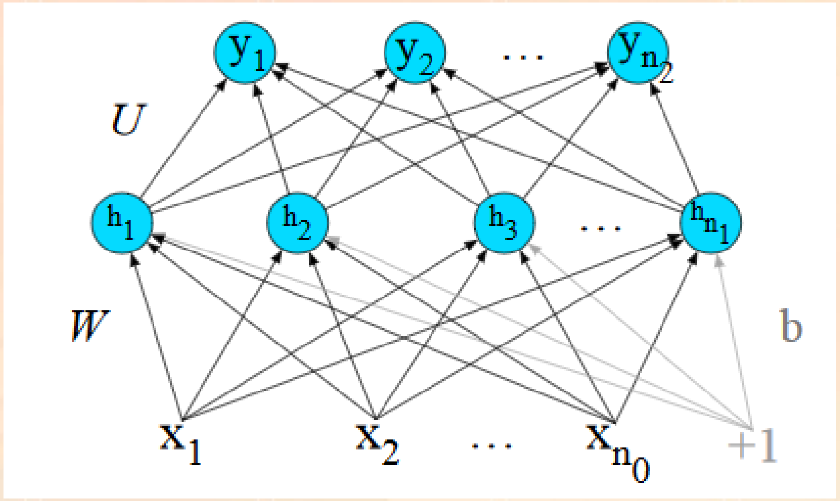
$$z = b + \sum_i w_i x_i$$

We represent the parameters for the entire hidden layer by combining the weight vector w_i and bias b_i for each unit i into a single weight matrix W and a single bias vector b for the whole layer .



Interpretation

Each element w_{ij} of the weight matrix W represents the weight of the connection from the i-th input unit x_i to the j-th hidden unit h_j.



The output of the hidden layer, the vector h, is thus the following, using the activation function g, which can be the sigmoid function σ:

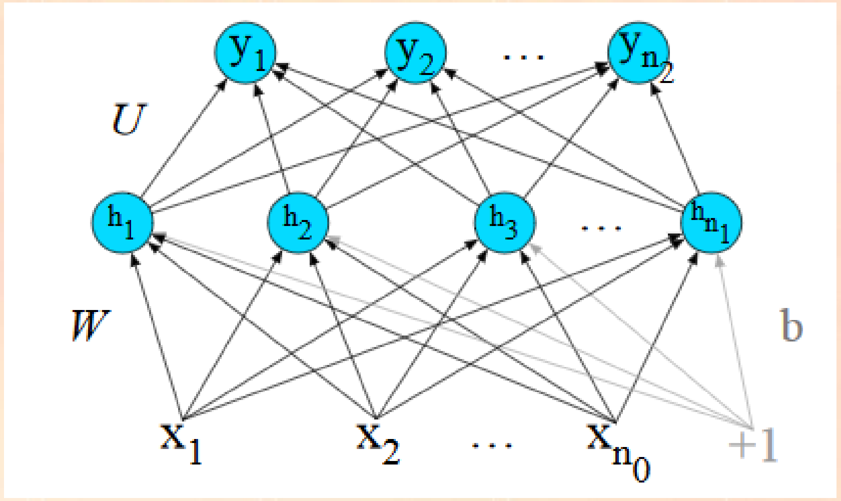
$$h = \sigma(Wx + b)$$

Note: we allow σ(·), and indeed any activation function g(·), to apply to a vector element-wise, so $g[z_1,z_2,z_3] = [g(z_1),g(z_2),g(z_3)]$.

Role of Layers of the Feed-forward network

The resulting value h (for hidden layer) forms a representation of the input. → $h = \sigma(Wx + b)$

The role of the output layer is to take this new representation h and compute the 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.

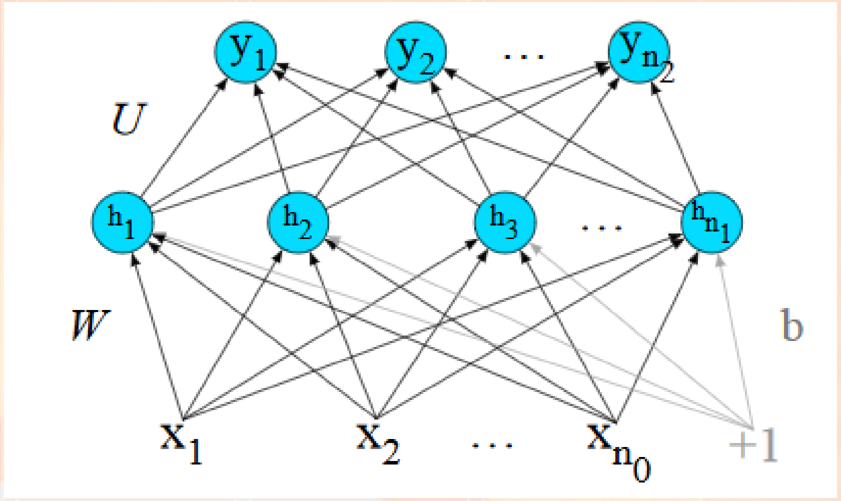


- *If we are doing a binary classification, we might have a single output node, and its value y is the probability of one class vs. the other class.*
- *If we are doing multinomial classification, such as assigning a part-of-speech tag, we might have <u>one output node for each potential part-of-speech</u>, 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 of the Feed-forward Network

➢ *how this happens? Like the hidden layer, the output layer has a weight matrix (let's call it U), but output layers may not have a bias vector b, so we'll simplify by eliminating the bias vector in this example.*

1. *The weight matrix is multiplied by its input vector (h) to produce the intermediate output z.* $z = Uh$



2. *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 function for normalizing a vector of real values to a vector that encodes a probability distribution (all the numbers lie between 0 and 1 and sum to 1): the **softmax function**

The softmax function

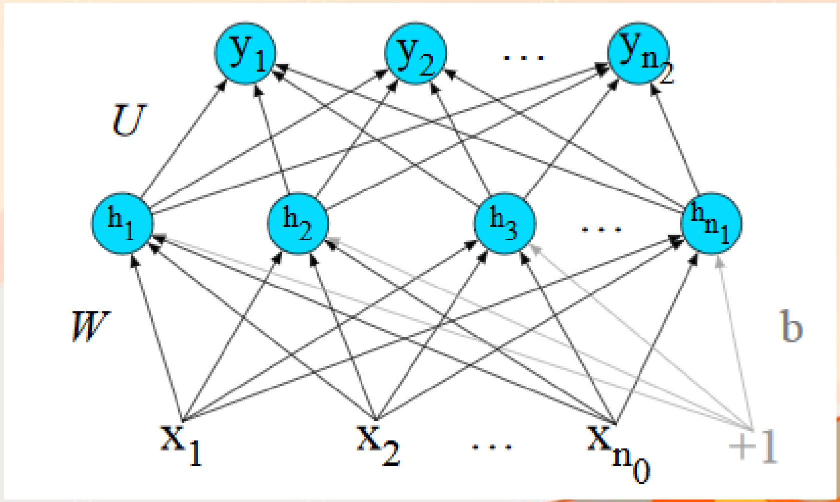
- *Definition:*

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

Example: given a vector *z*=[0.6 1.1 -1.5 1.2   3.2   -1.1],
softmax(z) is: [0.055   0.090   0.0067   0.10   0.74   0.010].

Final equations for a feed-forward network with a single hidden layer:

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



Observation

We can think of 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 logistic regression on the features that the network develops in h.

- *No feature engineering!*

So a neural network is like logistic regression, but:

a) with many layers, since a deep neural network is like layer after layer of logistic regression classifiers, and

b) rather than forming the features by feature templates, the prior layers of the network **induce** the feature representations themselves.

Deeper networks

- *More hidden layers! E.g. 2 hidden*

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

3-layer net

a$^{[i]}$ is the output from layer *i* and z$^{[i]}$ is the combination of weights and biases. The 0-th layer is for inputs, so the inputs *x* become a$^{[0]}$

- *N-layer Deep networks:*

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

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

Neural Language Models

As our first application of neural networks, let's consider *<u>language modeling</u>:*

➢ *predicting upcoming words from prior word context.*

Neural net-based language models turn out to have many advantages over the n-gram language models:

- neural language models don't need smoothing,
- they can handle much longer histories, and
- they can generalize over contexts of similar words
- a neural language model has much higher predictive accuracy than an n-gram language model

Disadvantage: neural language models are strikingly slower to train than traditional language models, and so for many tasks an n-gram language model is still the right tool.

Feedforward neural language model, (Bengio et al. 2003).

A feedforward neural LM is a standard 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.

- *like the n-gram LM—the feedforward neural LM approximates the probability of a word given the entire prior context by approximating based on the N previous words:*

$$P(w_t|w_1^{t-1}) \approx P(w_t|w_{t-N+1}^{t-1})$$

<u>Example:</u> we'll use a 4-gram example, so we'll show a neural net capable to estimate the probability $P(w_t{=}i|w_{t-1},w_{t-2},w_{t-3})$

Embeddings in Neural Language Models

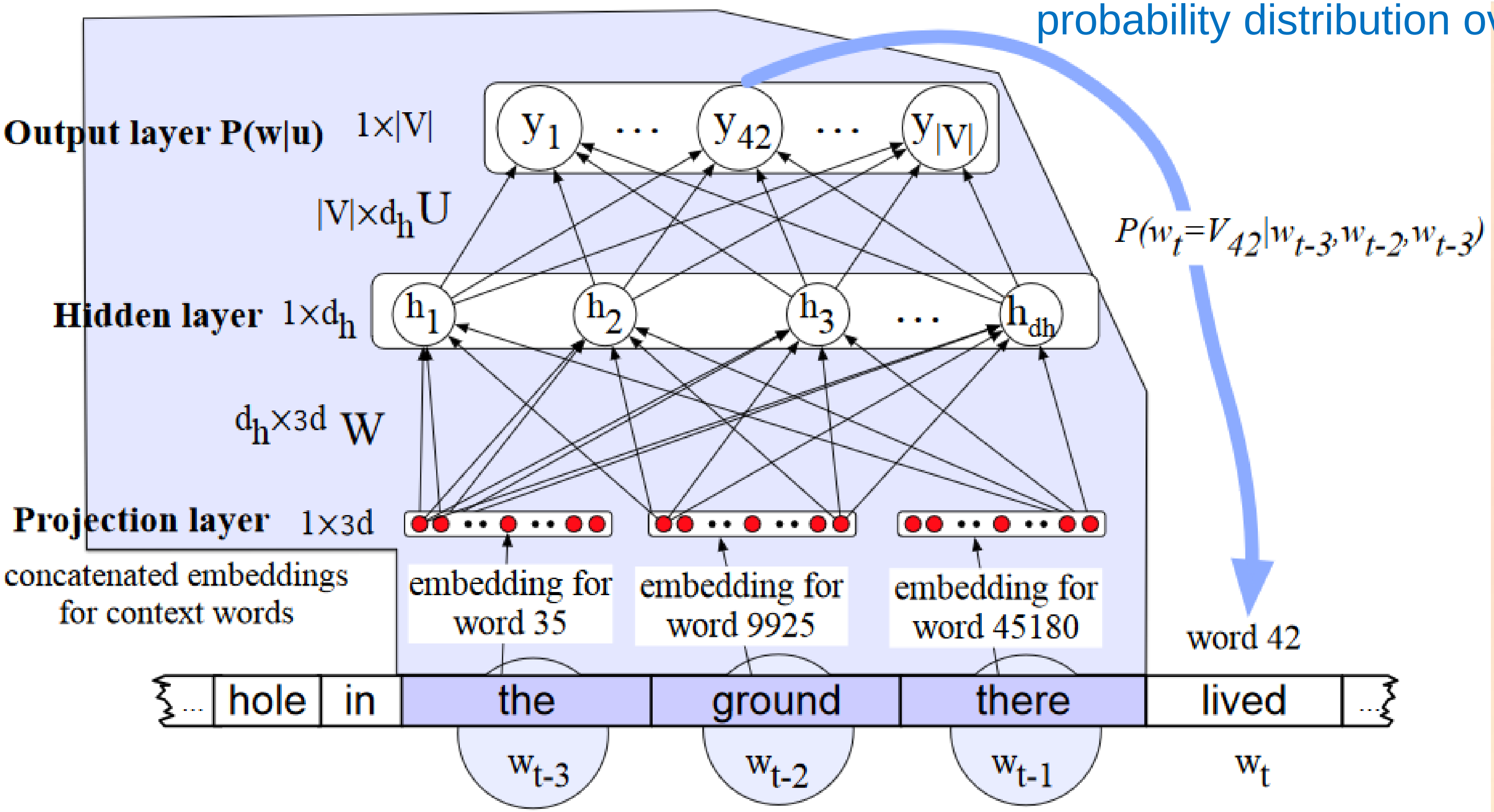
- *In neural language models, the prior context is represented by **embeddings of the previous words**.*

Representing the prior context as embeddings, rather than by exact words as used in n-gram language models, allows neural language models to generalize to unseen data much better than n-gram language models.

We build an embedding dictionary E that gives us, for each word in our vocabulary V, the embedding for that word, perhaps precomputed by an algorithm like word2vec.

Simplified View

The output is a *softmax* with a probability distribution over words.



Output layer P(w|u) 1×|V|
y_1 ... y_{42} ... $y_{|V|}$
$|V|\times d_h$ U
Hidden layer $1\times d_h$
h_1 h_2 h_3 ... h_{dh}
$d_h\times 3d$ W
Projection layer 1×3d
concatenated embeddings for context words
embedding for word 35
embedding for word 9925
embedding for word 45180
$P(w_t=V_{42}|w_{t-3},w_{t-2},w_{t-3})$
word 42
... hole in the ground there lived ...
w_{t-3}
w_{t-2}
w_{t-1}
w_t

A simplified FFNNLM with N=3; we have a moving window at time *t* with an embedding vector representing each of the 3 previous words (w_{t-1}, w_{t-2}, and w_{t-3}). These 3 vectors are concatenated together to produce *x*, the input layer.

Training the FFNNLM

➢ *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 .

How we do that??

1. 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.

2. We'll need to find the parameters that minimize this loss function, e.g. we'll use the gradient descent optimization algorithm.

Not so simple!!!!

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.

Here is one part where learning for neural networks is **more complex** than for logistic regression!!!!

In logistic regression, for each observation we could directly compute the derivative of the loss function with respect to an individual *w* or *b*.
But for neural networks, with millions of parameters in many layers, it's much harder to see how to compute the partial derivative of some weight in layer 1 when the loss is attached to some much later layer.

How do we partial out the loss over all those intermediate layers?
ANSWER: **the algorithm called error back-propagation.**
Gradient descent has an implementation for NNs:
Adam (Kingma and Ba, 2015).

What else???

Recommend: read about computation graphs and their usage in gradient descent in Chapter 7 of the 3rd edition of textbook.

❑ *Most modern neural networks are built using computation graph formalisms that make all the work of gradient computation and parallelization onto vector-based GPUs (Graphic Processing Units) very easy and natural.*

- *Pytorch (Paszke et al., 2017) and*
- *TensorFlow (Abadi et al., 2015) are two of the most popular frameworks.*

Optimization in neural networks is a non-convex optimization problem, more complex than for logistic regression, and for that and other reasons there are many best practices for successful learning.

- For logistic regression we can initialize gradient descent with all the weights and biases having the value 0.
- In neural networks, by contrast, 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.

2nd option for the neural language model

What happens if we'd like to learn the embeddings simultaneously with training the language neural network?

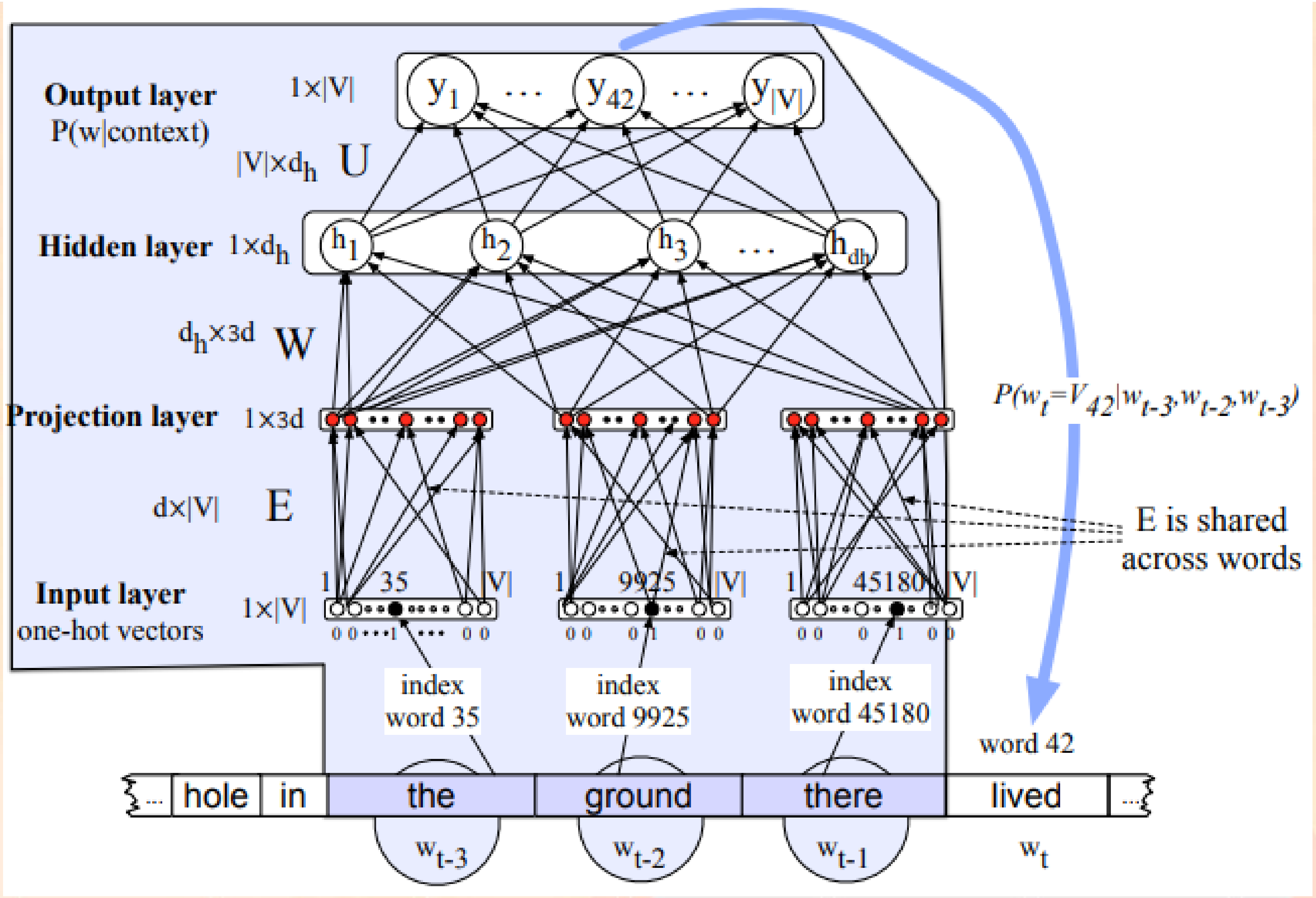
NECESSARY CHANGES:

1. we'll add an extra layer to the network, and propagate the error all the way back to the embedding vectors, starting with embeddings with random values and slowly moving toward sensible representations.

2. instead of pre-trained embeddings, we're going to represent each of the N=3 previous words as a one-hot vector of length |*V*|,

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.

Modified Architecture



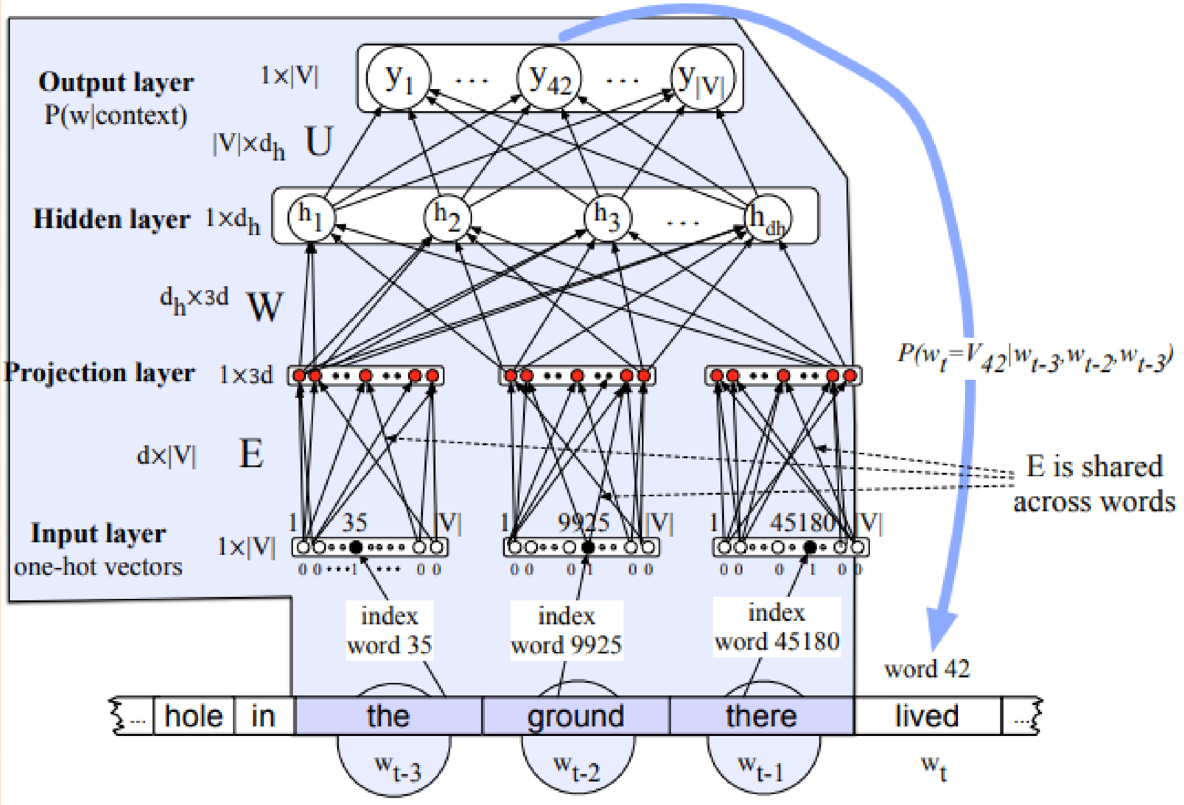
NEW: The embedding weight matrix E thus has a row for each word, each a vector of *d* dimensions, and hence has dimensionality $V \times d$.

Projection Layer

Also new is the projection layer!

- Since the input is a one-hot vector x_i for word V_i, which results in an embedding $e_i = Ex_i$
- in the projection layer we concatenate the embeddings of the previous context.

$$
\begin{aligned}
e &= (Ex_1, Ex_2, \ldots, Ex) \\
h &= \sigma(We + b) \\
z &= Uh \\
y &= \text{softmax}(z)
\end{aligned}
$$



Summary

- Neural networks are built out of neural units, originally inspired by human neurons but now simple an abstract computational device.
- Each neural unit multiplies input values by a weight vector, adds a bias, and then applies a non-linear activation function like sigmoid, tanh, or rectified linear.

- In a fully-connected, feedforward network, each unit in layer *i* is connected to each unit in layer *i+1*, and there are no cycles.
- The power of neural networks comes from the ability of early layers to learn representations that can be utilized by later layers in the network.

- Neural networks are trained by optimization algorithms like gradient descent.
- Error back propagation, backward differentiation on a computation graph, is used to compute the gradients of the loss function for a network.

- Neural language models use a neural network as a probabilistic classifier, to compute the probability of the next word given the previous n words.
- Neural language models can use pretrained embeddings, or can learn embeddings from scratch in the process of language modeling.