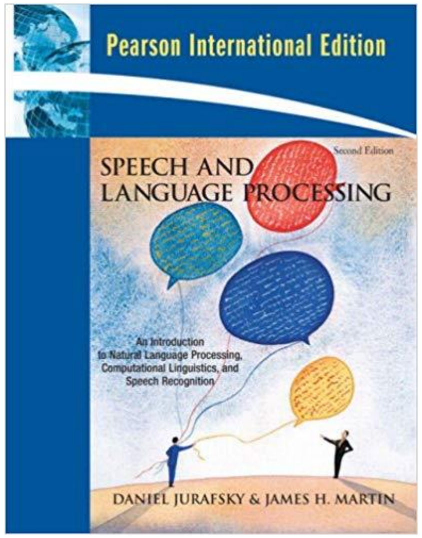
NLP & IR



Chapter 7

Neural Networks and
Neural Language Models

Dell Zhang
Birkbeck, University of London

Neural Networks

- 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.
- The architecture that we introduce in this chapter is called a **feed-forward network**, because the computation proceeds iteratively from one layer of units to the next.
- In this chapter we'll see feedforward networks as *classifiers*, and apply them to the simple task of *language modeling*: assigning probabilities to word sequences and predicting upcoming words.

NN vs LR

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

Deep Learning

- The use of modern neural nets is often called **deep learning**, because modern networks are often deep (have many layers).
  - Nets that are very deep are particularly good at *representation learning*.
  - For that reason, deep neural nets are the right tool for large scale problems that offer sufficient data to learn features *automatically*.

Neural Units

- The weights and bias

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$
$$= w \cdot x + b$$

- The sigmoid
(a special case of logistic function)

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

Does it look familiar?

Neural Units

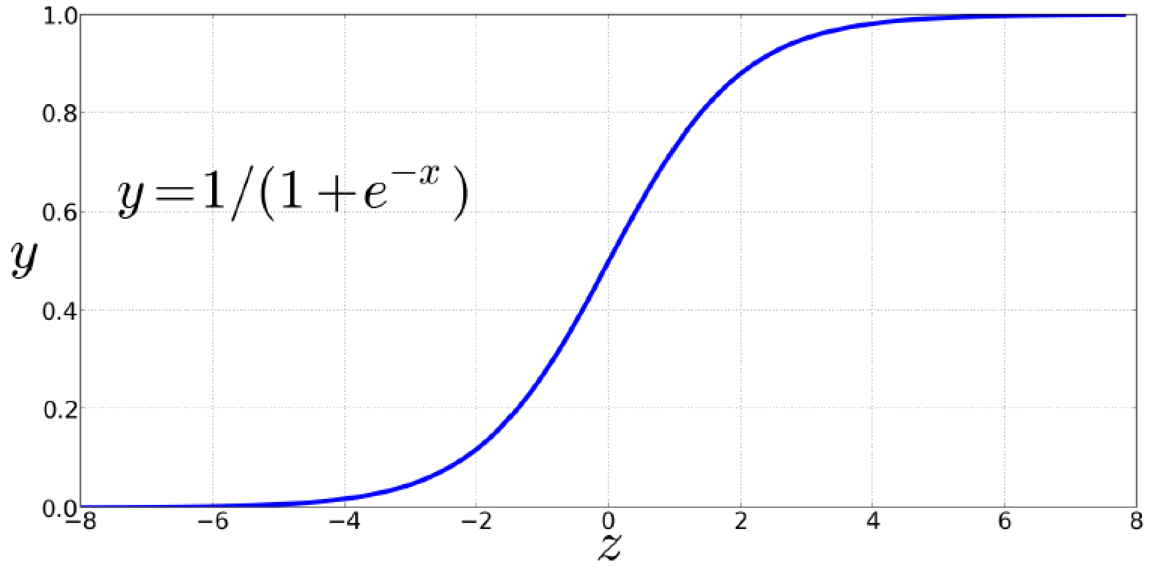


Figure 7.1

Figure 5.1 The sigmoid function $y = \frac{1}{1+e^{-z}}$ 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.

Neural Units

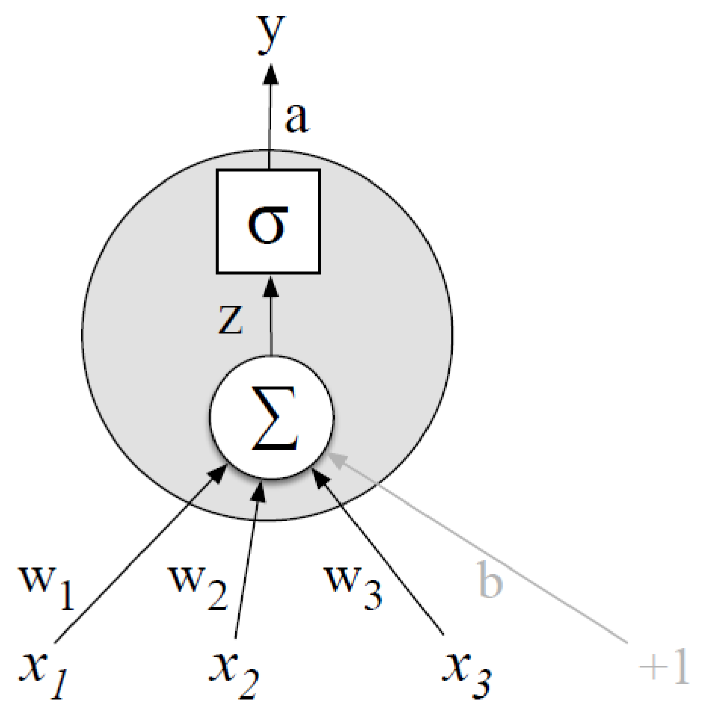


Figure 7.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a. In this case the output of 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.

Neural Units

- Example

Let's 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:

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

$$= \frac{1}{1 + e^{-(.5*.2+.6*.3+.1*.9+.5)}} = e^{-0.87} = .70$$

Neural Units

- Activation Function $y = a = f(z)$
  - 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.
    - It 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 just the same as x when x is positive, and 0 otherwise.

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

Neural Units

- Activation Function

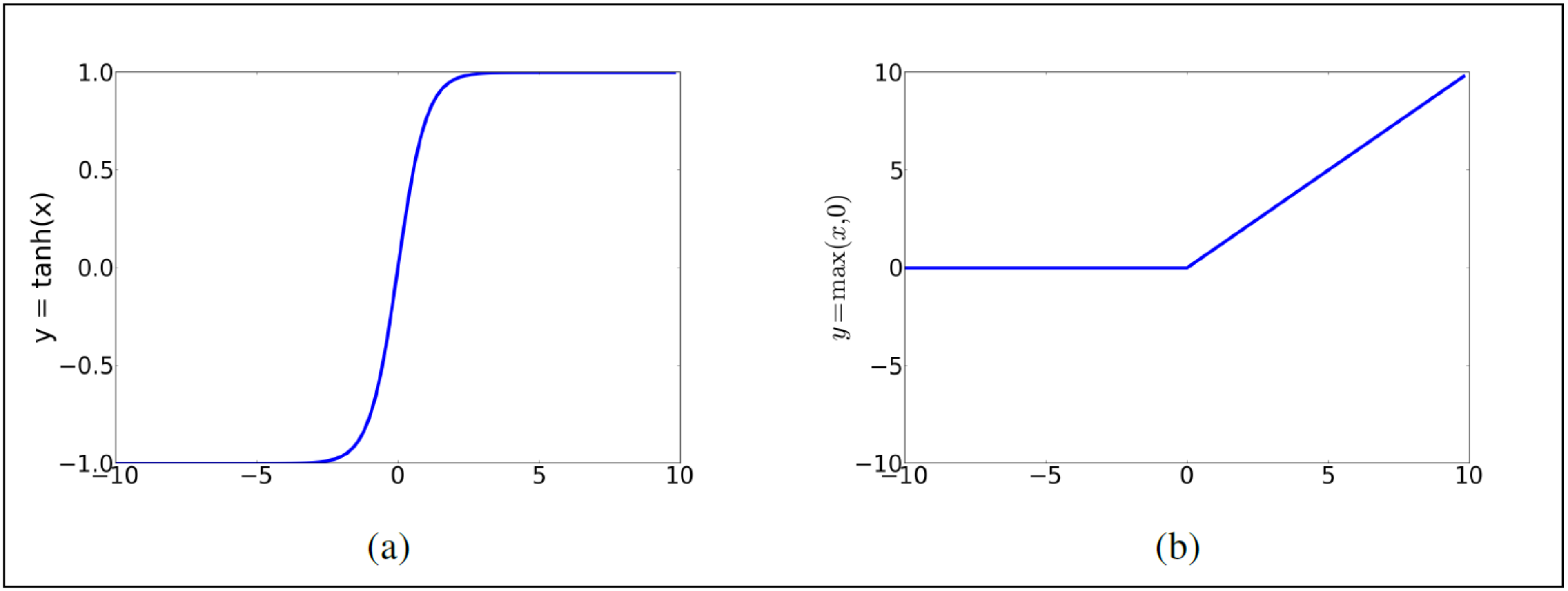


Figure 7.3 The tanh and ReLU activation functions.

Neural Units

- Activation Function
  - 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, which causes problems for learning. ReLU does not have this problem, since the output of values close to 1 also approaches 1 in a nice gentle linear way.
  - By contrast, the tanh function has the nice properties of being smoothly differentiable and mapping outlier values toward the mean.

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, multi-layer feedforward networks, are sometimes called multi-layer perceptrons (MLPs)
  - Simple feed-forward networks have three kinds of nodes: input units, hidden units, and output units.
  - In the standard architecture, each layer is **fully-connected**.

Feed-Forward Neural Networks

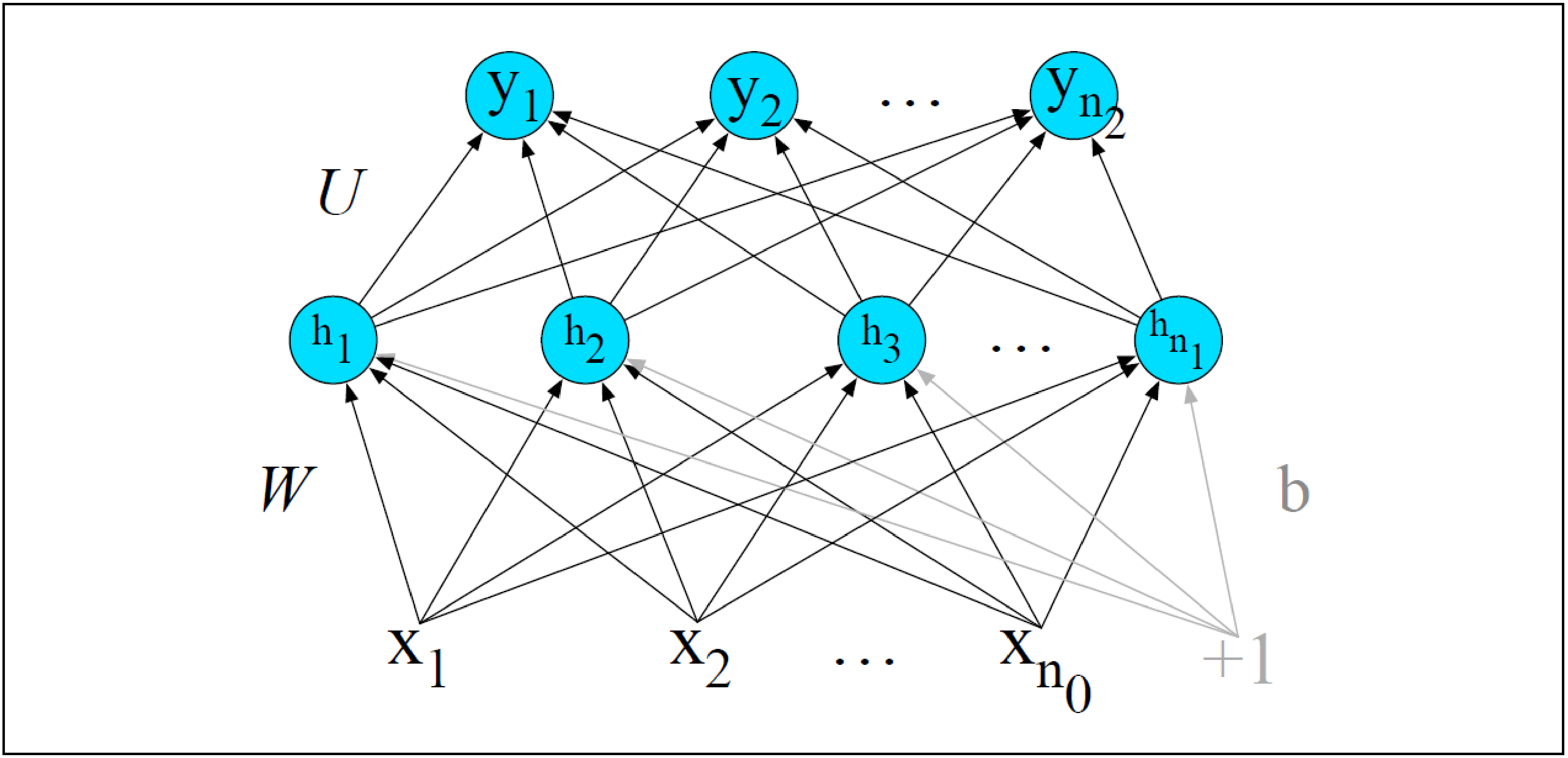


Figure 7.8 A simple 2-layer feed-forward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

Feed-Forward Neural Networks

- 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.

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

Feed-Forward Neural Networks

- A 3-layer net
  - 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.

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

Feed-Forward Neural Networks

- The algorithm for computing the forward step in an n-layer feed-forward network, given the input vector $a^{[0]}$, is thus simply:

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

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.

TensorFlow

PYTORCH

Computation Graphs

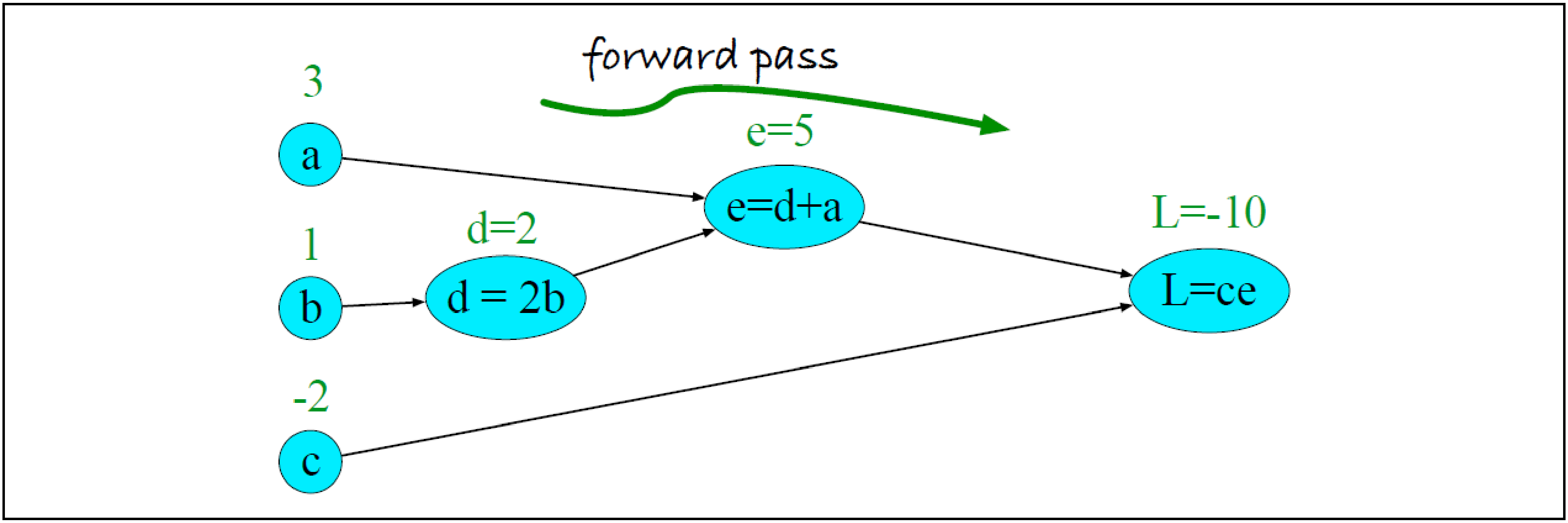


Figure 7.9 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -1$, showing the forward pass computation of L.

Computation Graphs

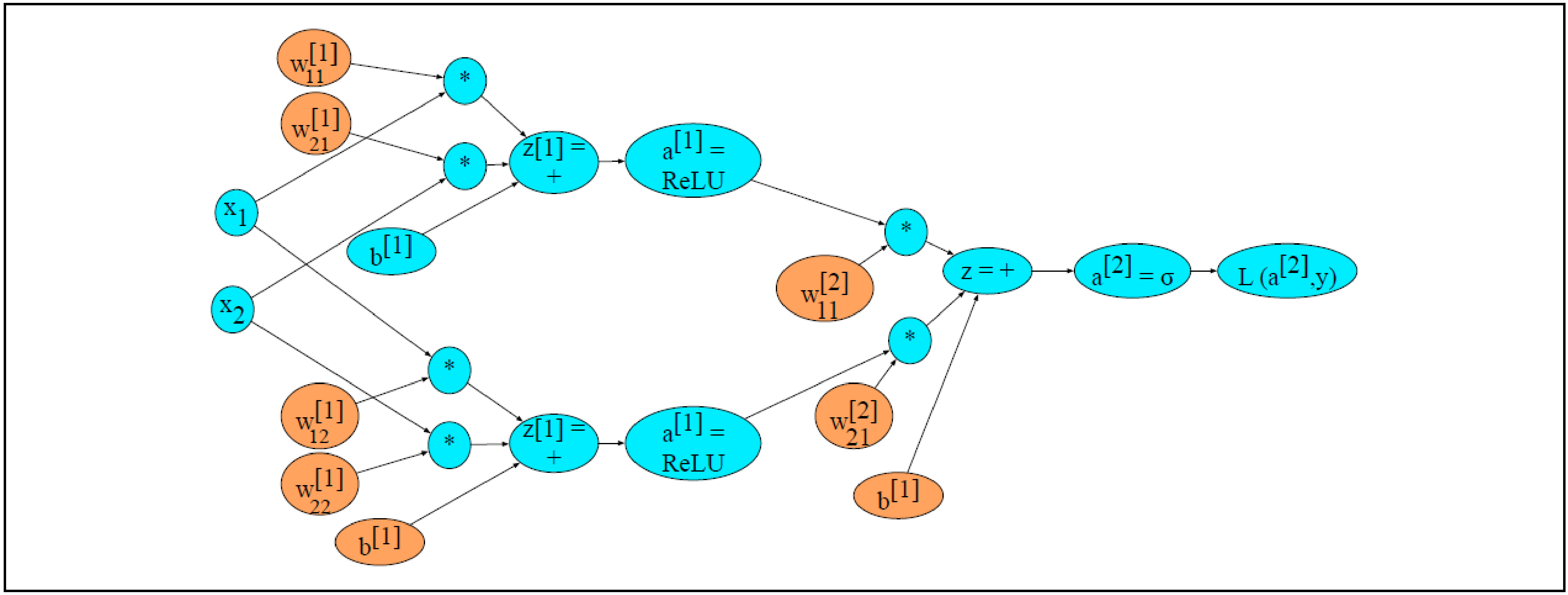


Figure 7.11 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input dimensions and 2 hidden dimensions.

NN vs LR (again)

- So a neural network is like logistic regression, but
- (a) with many layers.
  - A deep neural network is like layer after layer of logistic regression classifiers.
  - A logistic regression classifier is simply a one-layer neural network.
- (b) rather than forming the features by feature templates, the prior layers of the network induce the feature representations themselves.

Training Neural Nets

- For classification problems, neural networks use the **cross entropy** loss function, which is exactly the same one we saw in logistic regression.
- Backpropagation (backward differentiation on computation graphs).

Neural Language Models

- For a training set of a given size, a neural language model has **much higher predictive accuracy** than an n-gram language model due to the following advantages:
  - neural language models *don’t need smoothing*,
  - they can *handle much longer histories*, and
  - they can *generalize over contexts of similar words*.
- On the other hand, neural language models are **strikingly slower to train** than traditional n-gram language models.

Neural Language Models

- A feedforward neural LM is a standard feedforward network that takes as input at time t a representation of some number of previous words ($w_{t\text{-}1}$, $w_{t\text{-}2}$, etc.) and outputs a probability distribution over possible next words w_t.
- Like in the n-gram LM, the probability of a word given the entire prior context is approximated based on the N previous words:

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

Neural Language Models

- Embeddings
  - Representing the prior context as embeddings, rather than by exact words as used in n-gram language models would allow neural language models to generalize to unseen data much better than n-gram language models.
  - For example,
    - Training:
      "I have to make sure when I get home to *feed the cat*."
    - Testing:
      "I forgot when I got home to *feed the* _?_".

      What is the probability of the word "dog", using n-gram LM and neural LM respectively?

Neural Language Models

- Embeddings:
  Use pretrained embeddings
  - The embeddings could be learnt separately in advance by another algorithm like word2vec.
  - In this case, we have an embedding dictionary E that gives us, for each word in our vocabulary V, the embedding for that word.
  - Fig. 7.12 shows a sketch of this simplified neural LM with $N=3$.

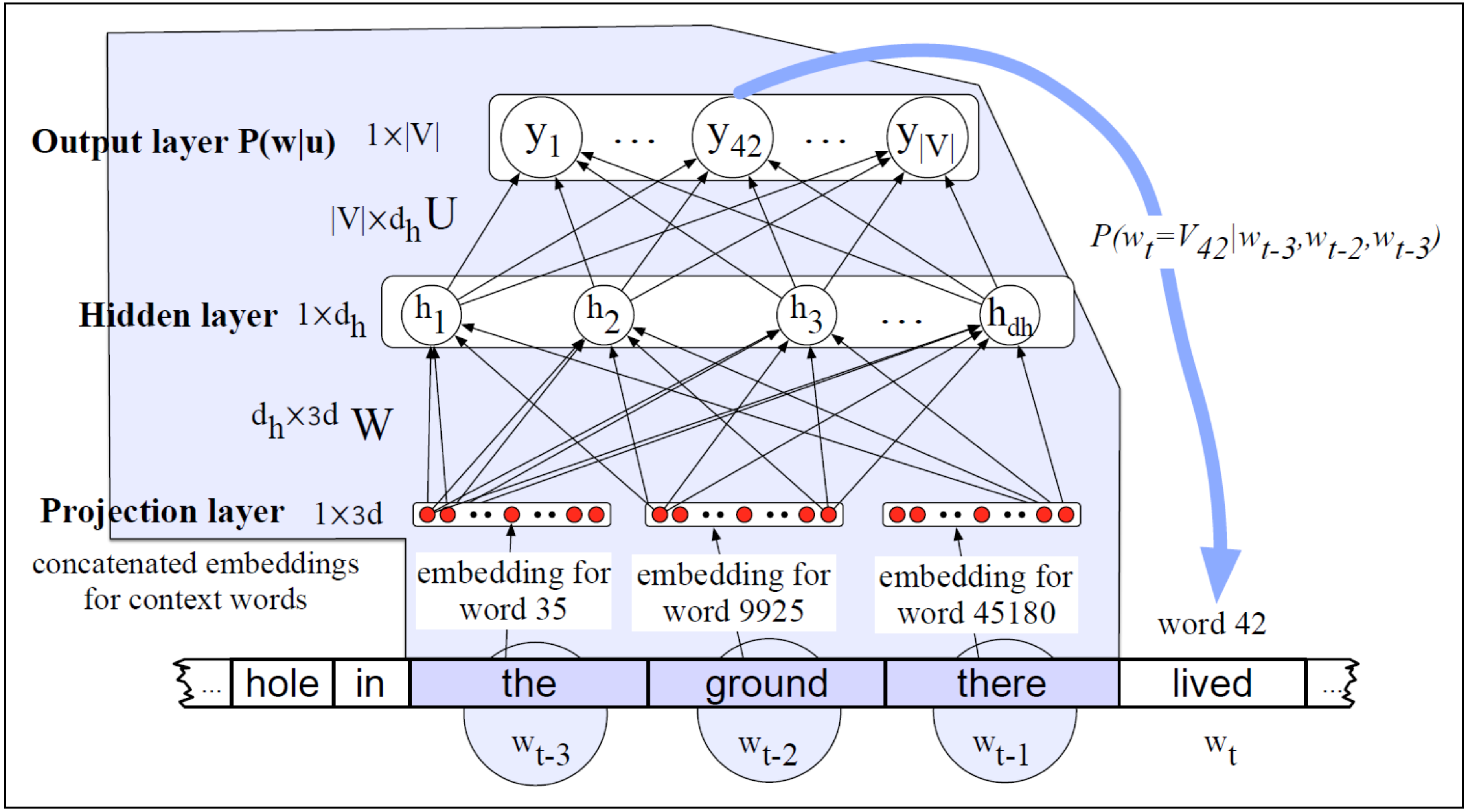

Output layer P(w|u) 1×|V|
y_1 … y_{42} … $y_{|V|}$
|V|×d_h U
Hidden layer 1×d_h
h_1 h_2 h_3 … h_{dh}
d_h×3d W
Projection layer 1×3d
concatenated embeddings for context words
embedding for word 35
embedding for word 9925
embedding for word 45180
$P(w_t=V_{42}|w_{t-3},w_{t-2},w_{t-3})$
word 42
… hole in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

Figure 7.12 A simplified view of a feedforward neural language model moving through a text. At each timestep *t* the network takes the 3 context words, converts each to a *d*-dimensional embeddings, and concatenates the 3 embeddings together to get the $1 \times Nd$ unit input layer *x* for the network. These units are multiplied by a weight matrix *W* and bias vector *b* and then an activation function to produce a hidden layer *h*, which is then multiplied by another weight matrix *U*. (For graphic simplicity we don't show *b* in this and future pictures). Finally, a softmax output layer predicts at each node *i* the probability that the next word w_t will be vocabulary word V_i. (This picture is simplified because it assumes we just look up in an embedding dictionary *E* the *d*-dimensional embedding vector for each word, precomputed by an algorithm like word2vec.)

Neural Language Models

- Embeddings:
  Learn embeddings simultaneously
  - It is desirable when whatever task the network is designed for places strong constraints on what makes a good representation.
  - To do this, we’ll add an extra layer to the network, and propagate the error all the way back to the embedding vectors. The embeddings would be initialized with random values, and slowly moved toward sensible representations.

Neural Language Models

- Embeddings:
  Learn embeddings simultaneously
  - At the input layer, instead of using pre-trained embeddings, we represent each of the N previous words as a *one-hot vector* of length $|V|$, 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 0.
    - For example, supposing the word "toothpaste" happens to have index 5 in the vocabulary, its one-hot representation is:

$$\begin{array}{cccccccccccc} [0 & 0 & 0 & 0 & 1 & 0 & 0 & \ldots & 0 & 0 & 0 & 0] \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & \ldots & & \ldots & & |V| \end{array}$$

Neural Language Models

- Embeddings:
  Learn embeddings simultaneously
  - Fig. 7.13 shows the additional layers needed to learn the embeddings during the training of LM.
  - Note that we want *one single embedding dictionary* E that's *shared* among all the context words, because we'd like to just represent each word with one vector whichever context position it appears in.
  - The embedding weight matrix E thus has a row for each word, each a vector of d dimensions, and hence has dimensionality $|V|\times d$.

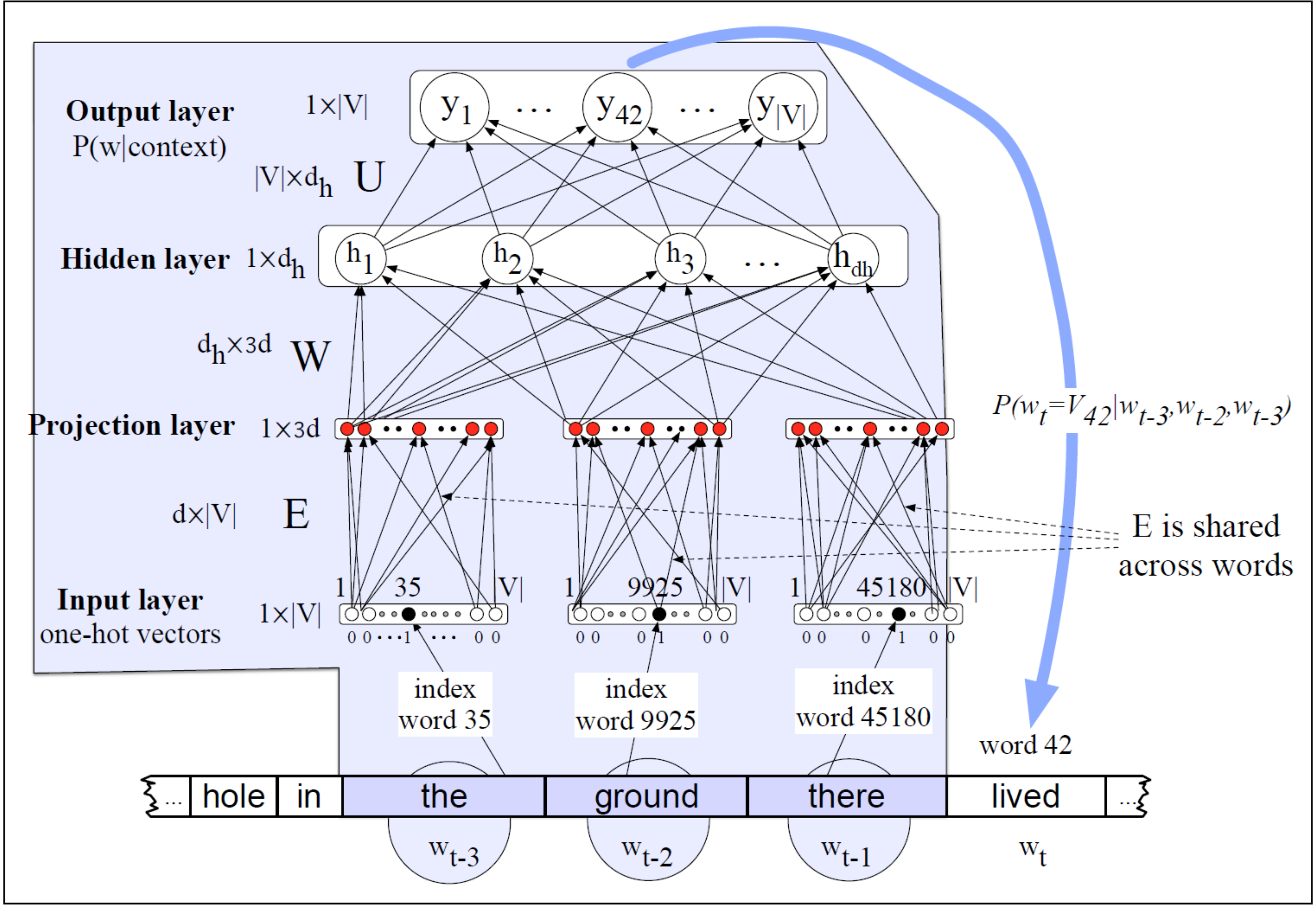
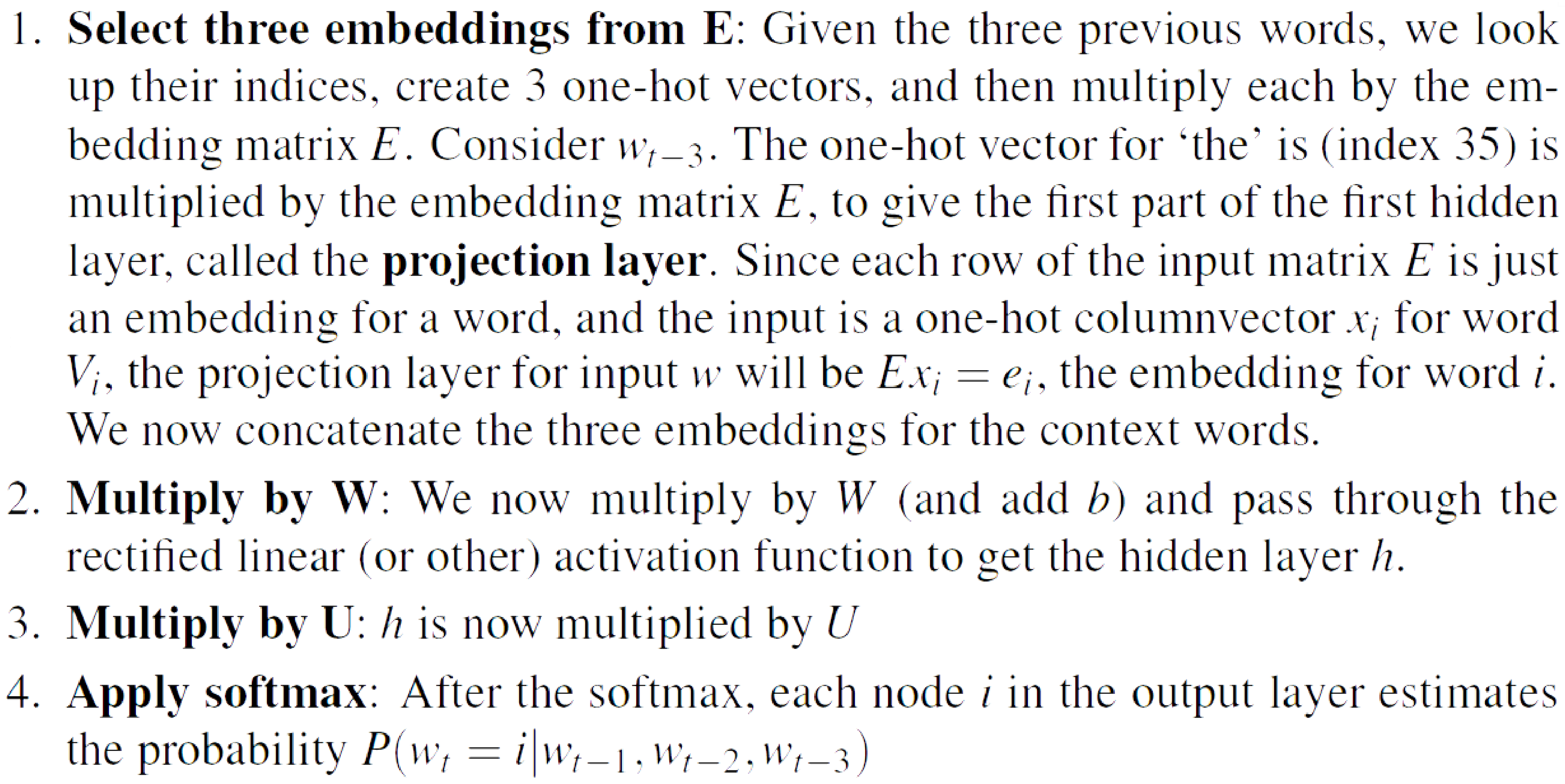


Figure 7.13 learning all the way back to embeddings. notice that the embedding matrix *E* is shared among the 3 context words.

Neural Language Models

- The forward pass of this neural LM (Fig. 7.13)



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 'the' is (index 35) is multiplied by the embedding matrix E, to give the first part of the first hidden layer, called the **projection layer**. Since each row of the input matrix E is just an embedding for a word, and the input is a one-hot columnvector x_i for word V_i, the projection 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.
2. **Multiply by W**: We now multiply by W (and add b) and pass through the rectified linear (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 $P(w_t = i|w_{t-1}, w_{t-2}, w_{t-3})$

Neural Language Models

- The equations of this neural LM (Fig. 7.13)

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

Neural Language Models

- Train the neural LM
  - Training proceeds by taking as input a very long text, concatenating all the sentences, start 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 $L = -\log p(w_t | w_{t-1}, \ldots, w_{t-n+1})$
  - Training will result in not only an algorithm for language modeling (*a word predictor*), but also a new set of embeddings which can be used as *word representations* for other tasks.

Distributional Hypothesis

- Words that occur in *similar contexts* tend to have *similar meanings*.

"You shall know a word by the company it keeps."
(Firth, J. R. 1957:11)