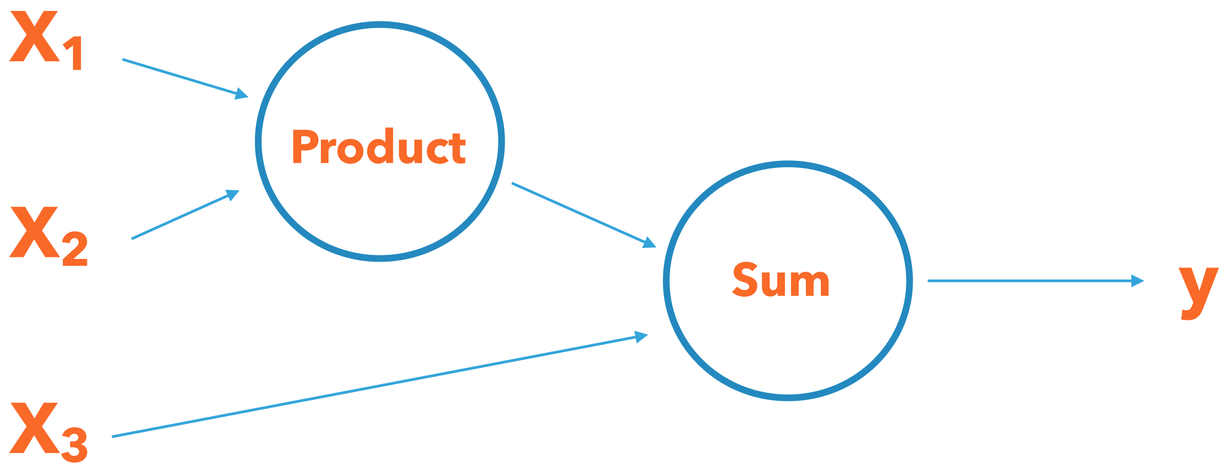
CMPE 297, INSTRUCTOR: JORJETA JETCHEVA

# NEURAL NETWORKS & DEEP LEARNING

# COMPUTATION GRAPHS

- Computation graphs are a way to think about computation
- Each node in the graph represents an elementary computation

$$y = x_1 \times x_2 + x_3$$

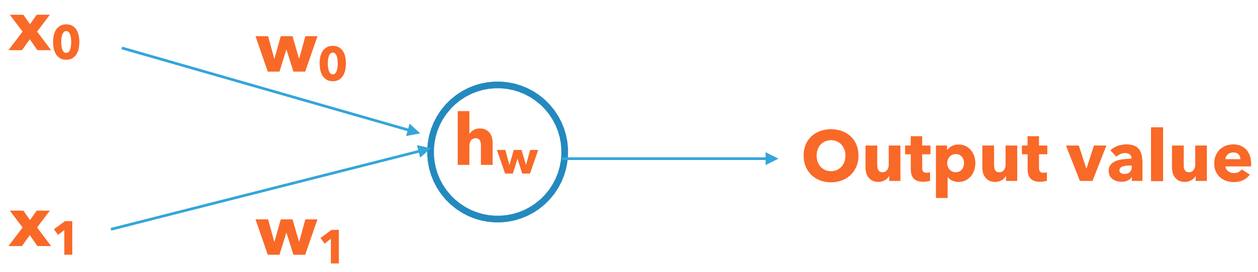


# VERY BASIC NEURAL NETWORK

The graph below is equivalent to a simple linear regression: $h_w(x) = w_1x + w_0$

There is an additional input added for convenience $x_0 = 1$:

$$h_w(x) = w_1x_1 + w_0x_0$$



# VERY BASIC NEURAL NETWORK FOR CLASSIFICATION

- The below network is equivalent to a logistic regression function that we use for binary classification

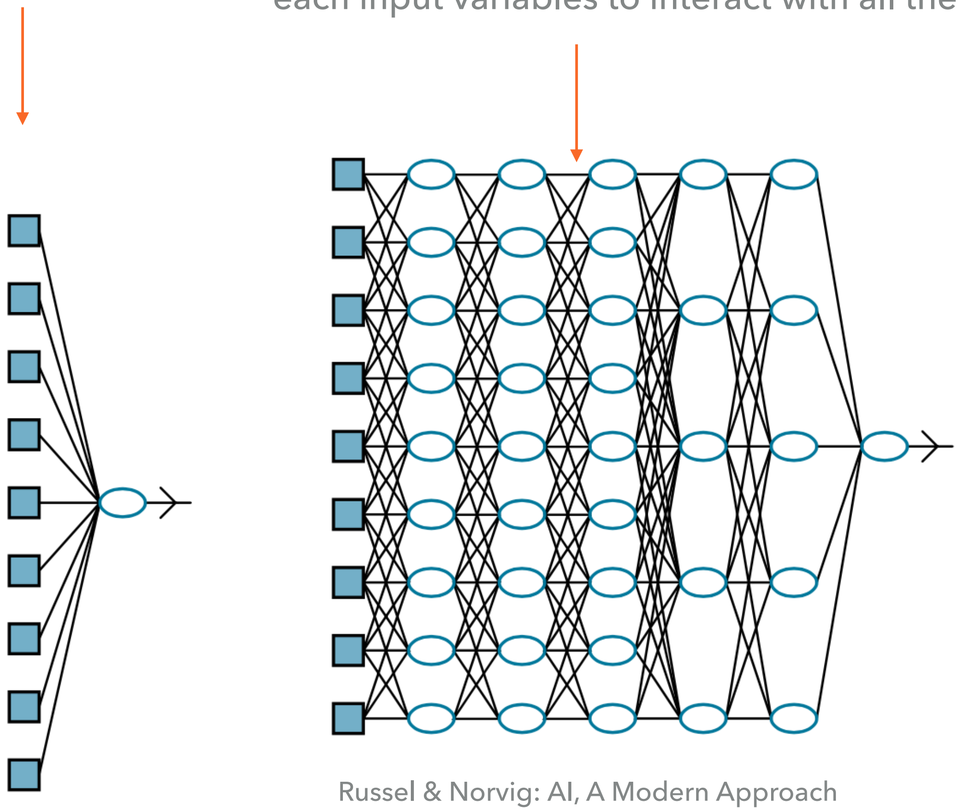

$h_w(x) = \sigma(w_1x_1 + w_0x_0)$ where $\sigma$ is the sigmoid function we use for logistic regression

Note: $P_{class2} = 1 - P_{class1}$

# NEURAL NETWORK VS. OTHER TYPES OF MODELS

Regression models

Deep neural networks (with multiple hidden layers) allow each input variables to interact with all the others

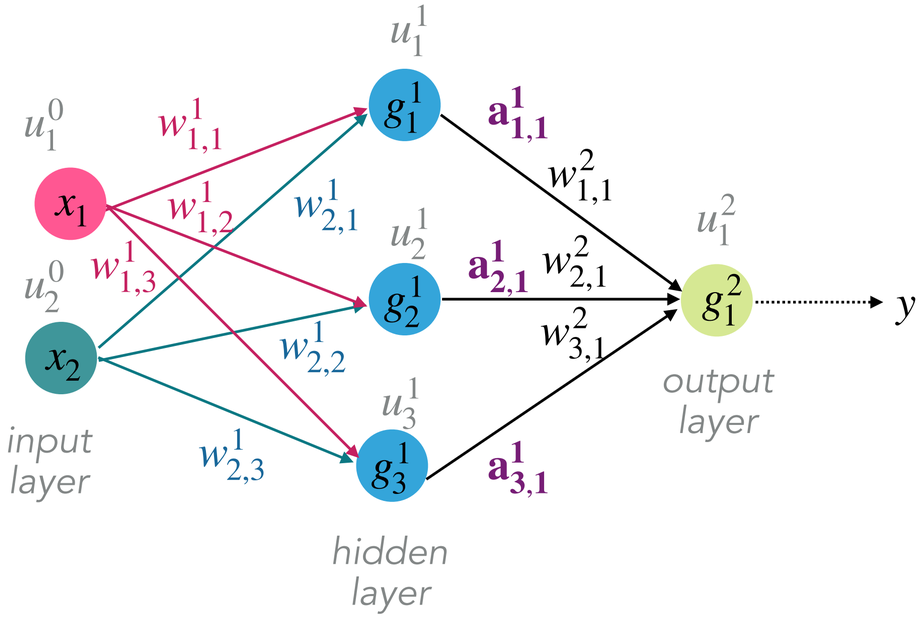


Russel & Norvig: AI, A Modern Approach

## UNIVERSAL APPROXIMATION

Even a simple neural network with one hidden layer can approximate any continuous function to an arbitrary degree of accuracy.

# GENERAL (FEEDFORWARD, FULLY CONNECTED) NEURAL NETWORK




Traditional fully connected configuration – all nodes in a layer are connected to all nodes in the next layer with directed edges

Note that neural networks can have more than one hidden layer, and more than one output, & a different connectivity pattern from the fully connected pattern

$u_j^l$ - *unit* $j$ *at layer* $l$

$w_{i,j}^l$ - *weight from unit* $u_i^{l-1}$ *to unit* $u_j^l$

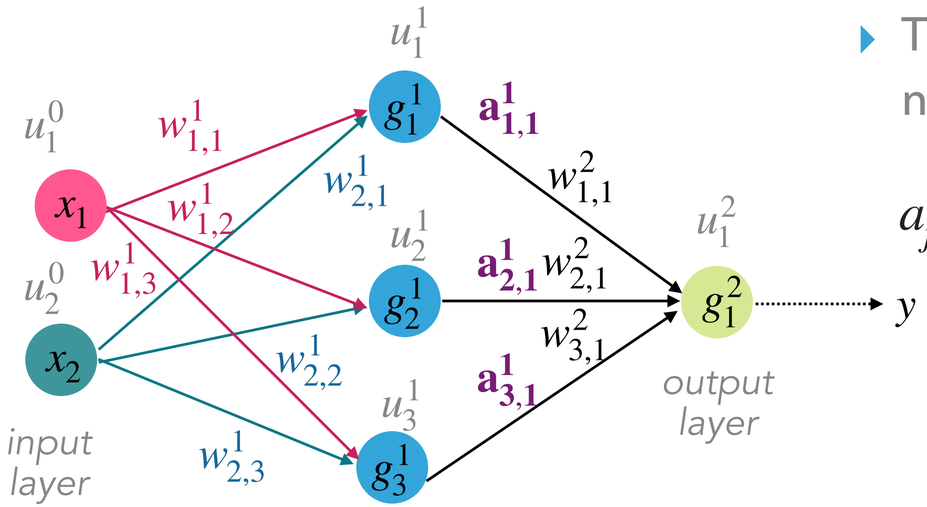
$g_j^l$ - *activation function applied in unit* $u_j^l$

$a_j^l$ - *activation output by unit* $u_j^l$

The computation is the same as for our earlier simple computational graph example:


$$h_w(x) = w_1x_1 + w_0x_0$$

# GENERAL (FEEDFORWARD, FULLY CONNECTED) NEURAL NETWORK (CONT.)




- Each node (or unit) **j** contains an activation function $\mathbf{g_j}$
- The activation function $\mathbf{g_j}$ at node **j** is applied to the inputs that go into node **j**, and its value, $\mathbf{a_j}$ (a.k.a. activation) becomes the output of node **j**

$a_j^l = g_j^l \left( \sum_i w_{i,j}^l \, a_i^l \right)$, where $\mathbf{a_i}$ is the input passed from node **i** to node **j**,

$\mathbf{w_{i,j}}$ is the weight of the edge from node **i** to node **j** and the superscript $l$ denotes the layer

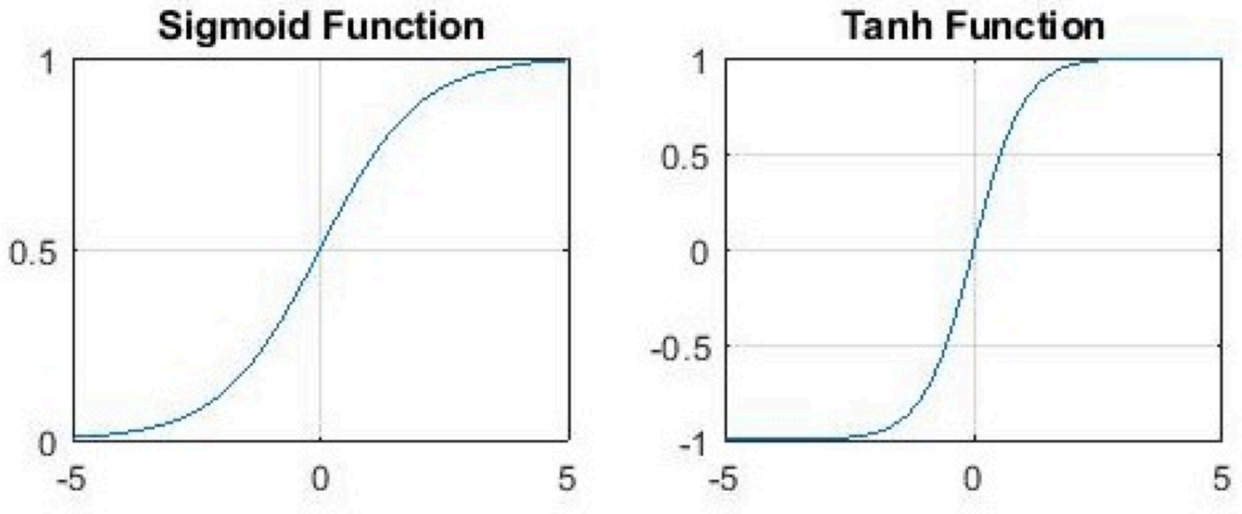
For ex.: $a_{1,1}^1 = g_1^1 (w_{1,1}^1 x_1 + w_{2,1}^1 x_2)$ & $y = g_1^2 (w_{1,1}^2 a_{1,1}^1 + w_{2,1}^2 a_{2,1}^1 + w_{3,1}^2 a_{3,1}^1)$

The process is similar to our earlier simple computational graph example for linear regression:


*Note: The general formulation for computing the output of a neural network unit dates back to the original neural networks paper by McCulloch & Pitts, 1943: a unit calculates the sum of inputs from its predecessor nodes and then applies a non-linear function to produce its output.*

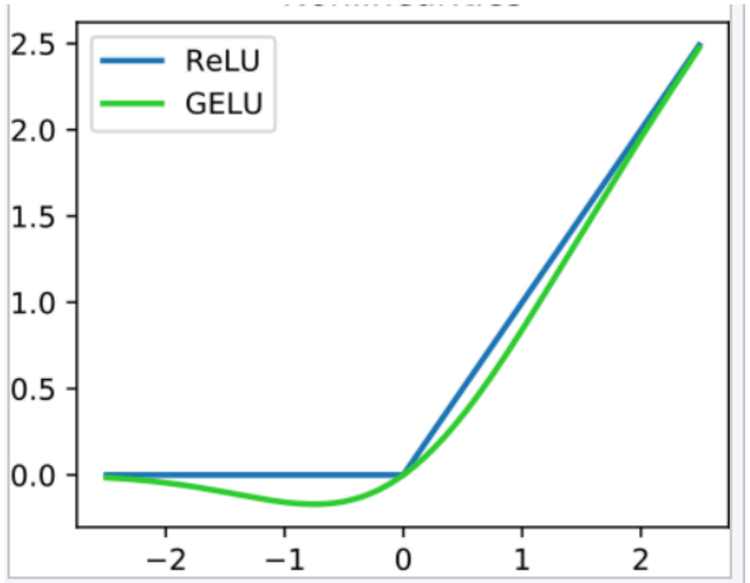
# TYPICAL ACTIVATION FUNCTIONS

- Linear hidden layer activation is not all that useful - the composition of linear functions is linear, so the neural network would be equivalent to a linear model




https://arxiv.org/pdf/1811.03378.pdf

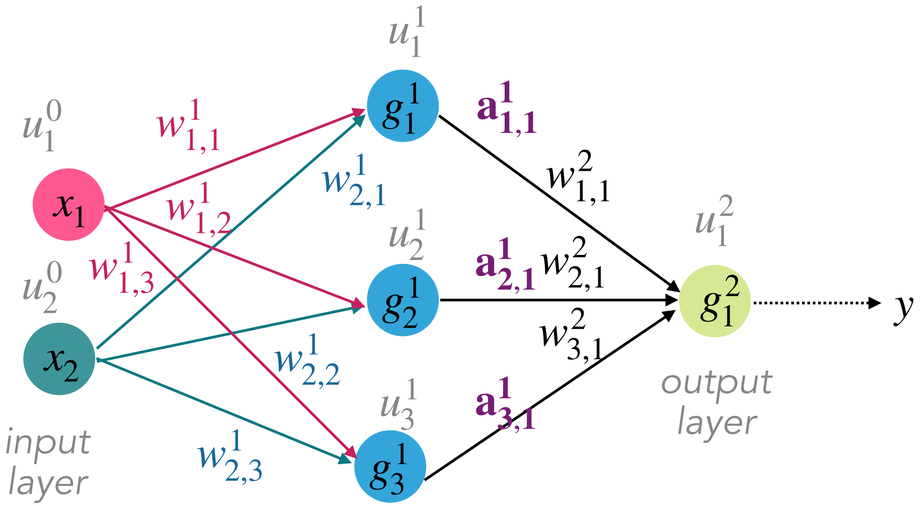
ReLU (Rectified Linear Unit): $max(x,0)$




Plot of the ReLU rectifier (blue) and GELU (green) functions near $x = 0$

http://wikipedia.org

# GENERAL (FEEDFORWARD, FULLY CONNECTED) NEURAL NETWORK (CONT.)




The computation of the network as a whole can be written as a composition of the activation functions as follows:

$$h_w(x) = g^{(2)}(W^{(2)}g^{(1)}(W^{(1)}x))$$

where the superscripts denote the layer of the network,

and $W^l$ denotes the set of weights in the particular layer $l$

# DUMMY INPUTS

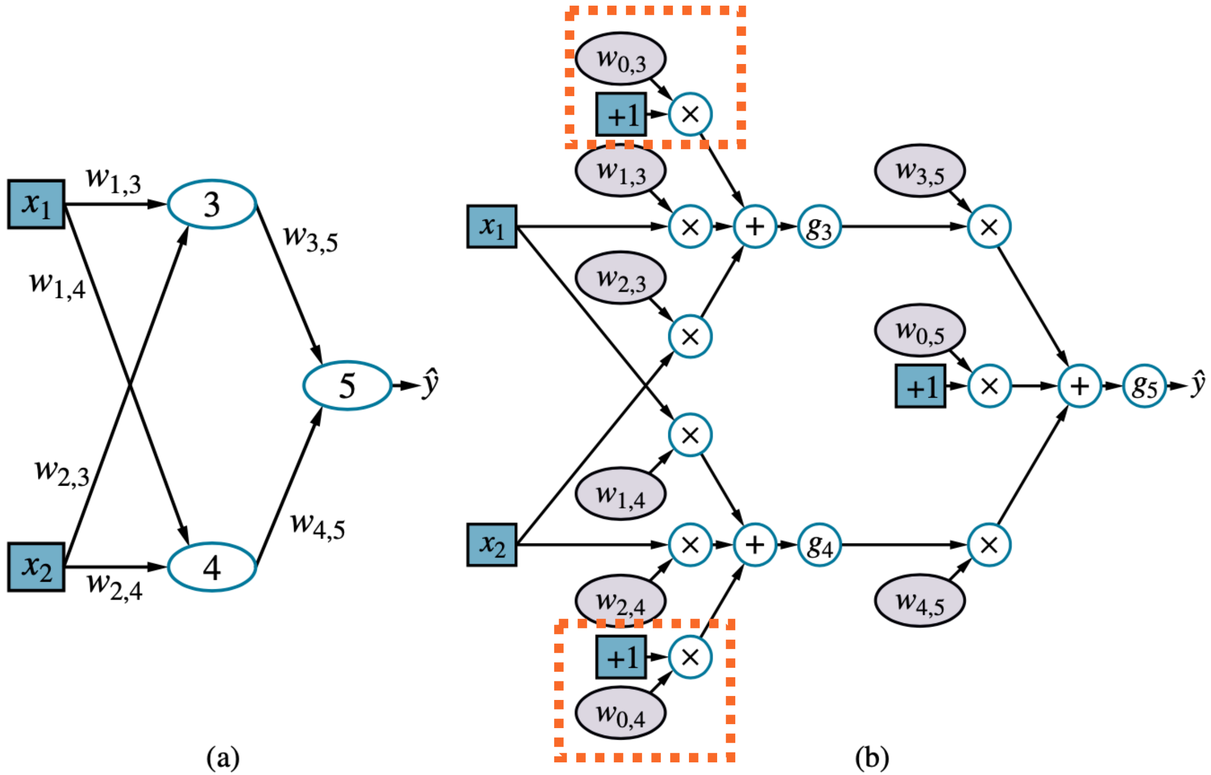



**Figure 21.3** (a) A neural network with two inputs, one hidden layer of two units, and one output unit. Not shown are the dummy inputs and their associated weights. (b) The network in (a) unpacked into its full computation graph. Russel & Norvig: AI, A Modern Approach

- Similarly to the bias term b in the line equation (y = ax + b), we have a bias term added to each node in the neural network
  - The bias term is a constant that allows us to adjust the model slightly to enable a better fit
  - The dummy unit ensures that even if the input into a particular node is 0 from all other nodes, it will still have some activation to pass to the next layer
- Each unit j has a dummy – an extra input from a dummy unit with index 0:
  - This input is always +1
  - The weight for the link from the dummy unit to node j is $w_{0,j}$
- The output node sometimes has a dummy input as well

*The dummy units are typically not shown for simplicity, but they are always there in the computation.*

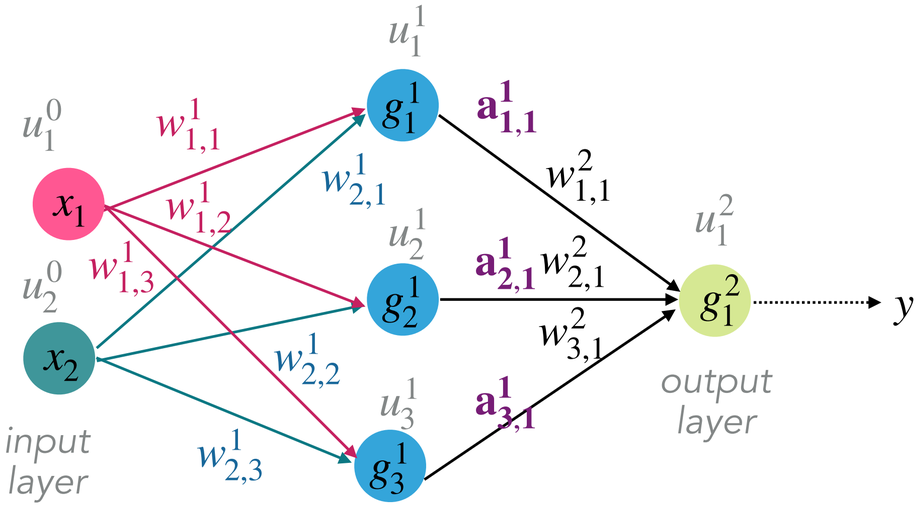
# NEURAL NETWORK PREDICTION

Recall that the computation of the network as a whole can be written as a composition of the activation functions as follows:

$h_w(x) = g^{(2)}(W^{(2)}g^{(1)}(W^{(1)}x))$

where the superscripts denote the layer of the network,
$g^{(l)}$ is the activation function at layer $l$,
$W^l$ denotes the set of weights in the particular layer $l$
and there are $L$ hidden layers

- Is a forward pass through the (*already trained*) neural network
- The output is either a value (if we are predicting a value) or class probabilities (if we are doing classification)



# FORWARD PASS (INPUT LAYER TO HIDDEN LAYER)

*Each row in the matrix W are the weights going into a particular unit in the current layer from all units in the previous layer*

$$W^{(1)} = \begin{bmatrix} w^{(1)}_{1,1}, w^{(1)}_{2,1}, \ldots, w^{(1)}_{m,1} \\ w^{(1)}_{1,2}, w^{(1)}_{2,2}, \ldots, w^{(1)}_{m,2} \\ \vdots \\ w^{(1)}_{1,n1} w^{(1)}_{2,n1}, \ldots, w^{(1)}_{m,n1} \end{bmatrix} , x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

*n1 is the number of units in layer 1, n2 is the number of units in layer 2, etc.*

*m is the number of input units*

$$W^{(1)} \times x = \begin{bmatrix} w^{(1)}_{1,1} \times x_1 + w^{(1)}_{2,1} \times x_2 \ldots + w^{(1)}_{m,1} \times x_m \\ w^{(1)}_{1,2} \times x_1 + w^{(1)}_{2,2} \times x_2 \ldots + w^{(1)}_{m,2} \times x_m \\ \vdots \\ w^{(1)}_{1,n1} \times x_1 + w^{(1)}_{2,n1} \times x_2 \ldots + w^{(1)}_{m,n1} \times x_m \end{bmatrix}$$

$$g^{(1)}(W^{(1)} \times x) = \begin{bmatrix} a^{(1)}_{1,1} \\ a^{(1)}_{2,1} \\ \vdots \\ a^{(1)}_{n1,1} \end{bmatrix} = a^{(1)}$$, *where $a^{(1)}$ is the (vector of) activations output by Layer 1*

Recall that the computation of the network as a whole can be written as a composition of the activation functions as follows:

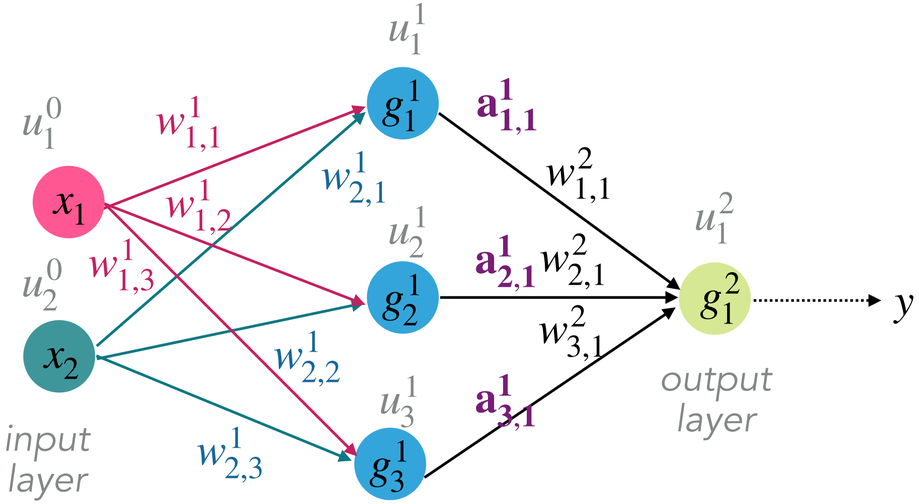
$h_w(x) = g^{(2)}(W^{(2)}g^{(1)}(W^{(1)}x))$

where the superscripts denote the layer of the network,

$g^{(l)}$ is the activation function at layer l,

$W^l$ denotes the set of weights in the particular layer $l$

and there are L hidden layers



# FORWARD PASS (HIDDEN LAYER TO OUTPUT LAYER)

Recall that the computation of the network as a whole can be written as a composition of the activation functions as follows:

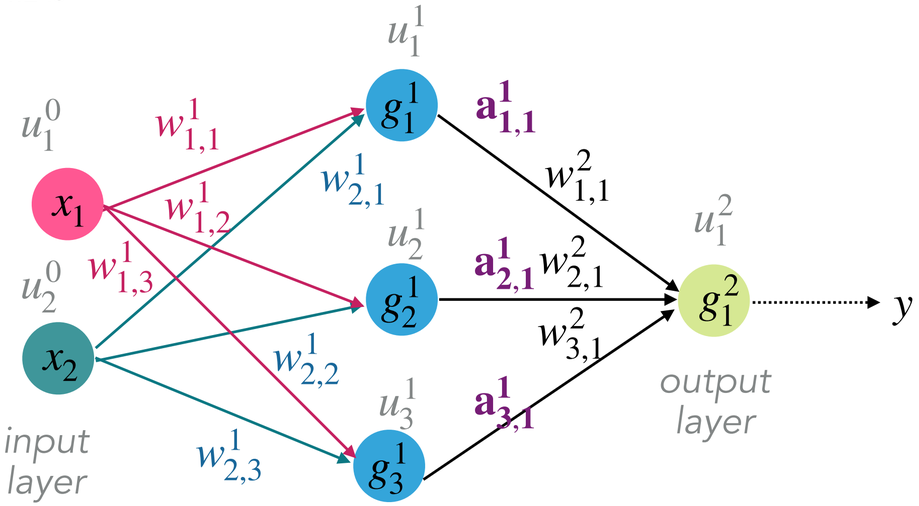
$h_w(x) = g^{(2)}(W^{(2)}g^{(1)}(W^{(1)}x))$

where the superscripts denote the layer of the network,
$g^{(l)}$ is the activation function at layer l,
$W^l$ denotes the set of weights in the particular layer $l$
and there are L hidden layers

$$W^{(2)} = \begin{bmatrix} w^{(2)}_{1,1}, w^{(2)}_{2,1}, \ldots, w^{(2)}_{n1,1} \\ w^{(2)}_{1,2}, w^{(2)}_{2,2}, \ldots, w^{(2)}_{n1,2} \\ \vdots \\ w^{(2)}_{1,n2} w^{(2)}_{2,n2}, \ldots, w^{(2)}_{n1,n2} \end{bmatrix}, a^{(1)} = \begin{bmatrix} a^{(1)}_{1,1} \\ a^{(1)}_{1,2} \\ \vdots \\ a^{(1)}_{1,n1} \end{bmatrix}$$

*n1 is the number of units in layer 1, n2 is the number of units in layer 2, etc.*

$$W^{(2)} \times a^{(1)} = \begin{bmatrix} w^{(2)}_{1,1} \times a_1{}^{(1)} + w^{(2)}_{2,1} \times a_2{}^{(1)} \ldots + w^{(2)}_{m,1} \times a_{n1}{}^{(1)} \\ w^{(2)}_{1,2} \times a_1{}^{(1)} + w^{(2)}_{2,2} \times a_2{}^{(1)} \ldots + w^{(2)}_{m,2} \times a_{n1}{}^{(1)} \\ \vdots \\ w^{(2)}_{1,n2} \times a_1{}^{(1)} + w^{(2)}_{2,n2} \times a_2{}^{(1)} \ldots + w^{(2)}_{m,n2} \times a_{n2}{}^{(1)} \end{bmatrix}$$




$$g^{(2)}(W^{(2)} \times a^{(1)}) = \begin{bmatrix} a^{(2)}_{2,1} \\ a^{(2)}_{2,2} \\ \vdots \\ a^{(2)}_{2,n2} \end{bmatrix} = a^{(2)}$$

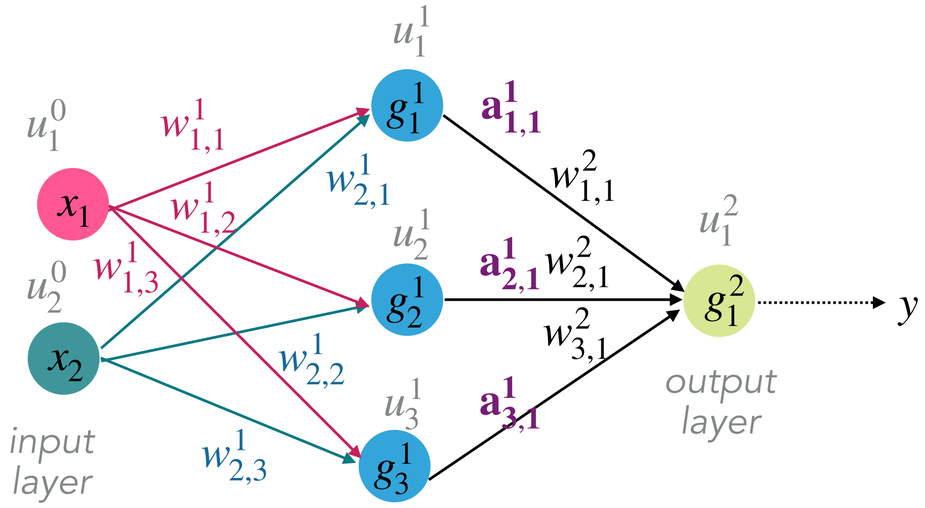
*In the example in the figure, since $g^{(2)}$ is the activation function of the output layer, which has only 1 unit in this case, n2 = 1, and thus $a^{(2)}$ would be a single value, not a vector.*

# TRAINING THE NEURAL NETWORK

Start with random assignment for all the weights, and then repeat until convergence (e.g., can't reduce error further):

- Forward pass through the network (performing all the computations until we get an output value) for all the training examples
- A reverse pass (*backpropagation*) from output towards the input, where we "pass" the error that occurs on the output back through the network and adjust all the weights accordingly

Each pass over the entire training set is a training epoch




*The above loop executes batch gradient descent (we do a forward pass over the whole training set first and then perform backpropagation)*

# TRAINING THE NEURAL NETWORK (CONT.)

**Batch gradient descent (from earlier slide):** Start with random assignment for all the weights, and then repeat until convergence (e.g., can't reduce error further):

- Forward pass through the network (performing all the computations until we get an output value) for all the training examples
- A reverse pass (*backpropagation*) from output towards the input, where we "pass" the error that occurs on the output back through the network and adjust all the weights accordingly

Each pass over the entire training set is a training epoch

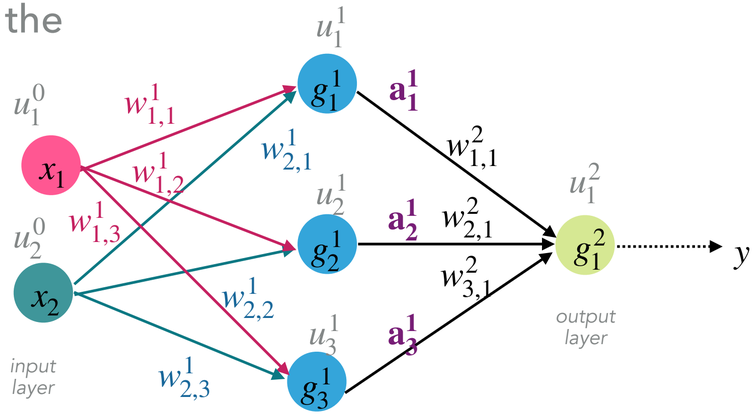
**Stochastic Gradient Descent (SGD) - lower computational complexity than batch gradient descent**

**Option 1:**

- We do a forward pass for each training example, followed by a back propagation pass for the training example
- Convergence tends to occur before we go through the whole training set

**Option 2:**

- We divide the input training examples into small batches, e.g., of size 100
- We do a forward pass for all examples in a batch
- Then we do a back propagation pass




*Training often stops when we run out of time and/or compute resources as opposed to when the error is minimized.*

*Note that there are multiple other options for gradient descent besides SGD, e.g., RMSProp, Adam, etc.*

$$w_i = w_i - \alpha \frac{\partial L}{\partial w_i}$$

# BACKPROPAGATION CHALLENGES – VANISHING GRADIENT PROBLEM

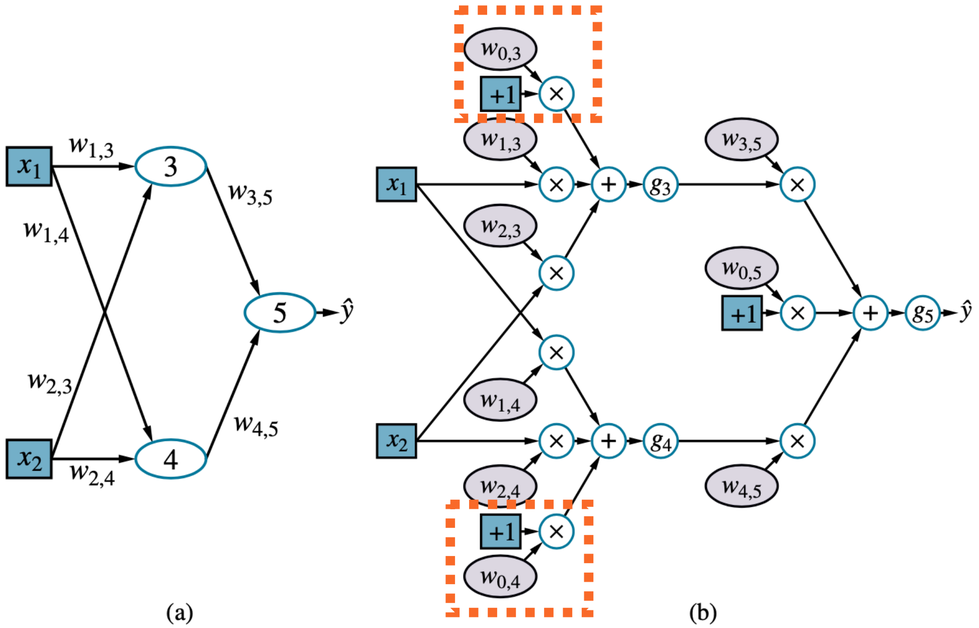



**Figure 21.3** (a) A neural network with two inputs, one hidden layer of two units, and one output unit. Not shown are the dummy inputs and their associated weights. (b) The network in (a) unpacked into its full computation graph. Russel & Norvig: AI, A Modern

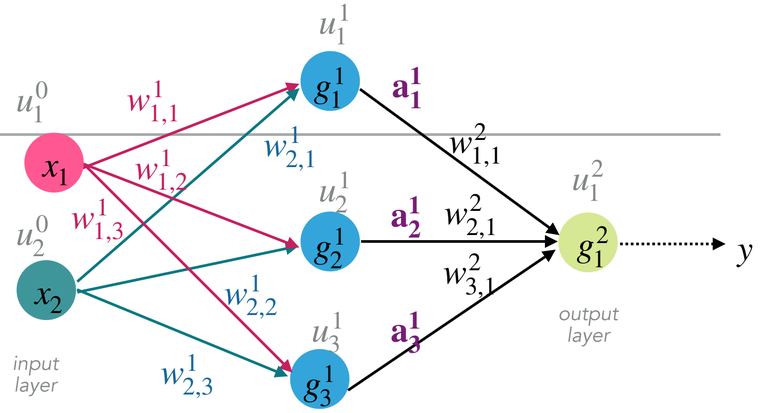
$$\frac{\partial}{\partial w_{1,3}} Loss(h_w) = \frac{\partial}{\partial w_{1,3}} (y - \hat{y})^2 = -2(y - \hat{y}) g_5'(in_5) w_{3,5} g_3'(in_3) x_1$$

- Backpropagation involves multiplying derivatives (e.g., example above)
- In some cases, derivatives at the various nodes can be very close to 0
- When you multiply a sequence of numbers that are very close to 0 (esp when you have a deep neural network), signals propagating through the network are extinguished and you can't make further progress

Popular solution to the vanishing gradient problem: Batch Normalization (for SGD)

- Rescales the values generated in the internal layers of the network for each minibatch of input examples (as if we've added a normalization layer between each two layers)
- Speeds up convergence of SGD
- Note that the reasons for its effectiveness are not well understood

# INPUT ENCODING




- The inputs to a neural network need to be numerical
  - Normalization is important
- Categorical variables are encoded using a one-hot encoding: each value is encoded with a vector which has 1 in the index corresponding to a particular categorical variable value (in the list of values) and the rest of the dimensions of the vector are 0s

| | Red | Green | Blue |
|---|---|---|---|
| **Red** | 1 | 0 | 0 |
| **Green** | 0 | 1 | 0 |
| **Blue** | 0 | 0 | 1 |

*Note that this example would require 3 inputs: $x_1$, $x_2$, $x_3$ for the 3 dimensions of the input vectors*

# OUTPUT LAYER

- Output encoding
  - Target variable encoding (for categorical variables) can also be done using one-hot encoding (e.g., see weather prediction example in table)
- Output activation function
  - Linear for regression problems (no activation function - just pass through the predicted value)
  - Softmax for classification problems (e.g., a vector with probabilities for all the classes that add up to 1), e.g.,

```
Probabilities of classes 1 through 4:
[0.10588, 0.06422, 0.04757, 0.78233]
```

| | Rainy | Sunny | Cloudy |
|---|---|---|---|
| **Rainy** | 1 | 0 | 0 |
| **Sunny** | 0 | 1 | 0 |
| **Cloudy** | 0 | 0 | 1 |

# FORMULATING SENTIMENT CLASSIFICATION AS A NEURAL NETWORK CLASSIFICATION PROBLEM

## Option 1: hand-crafted features

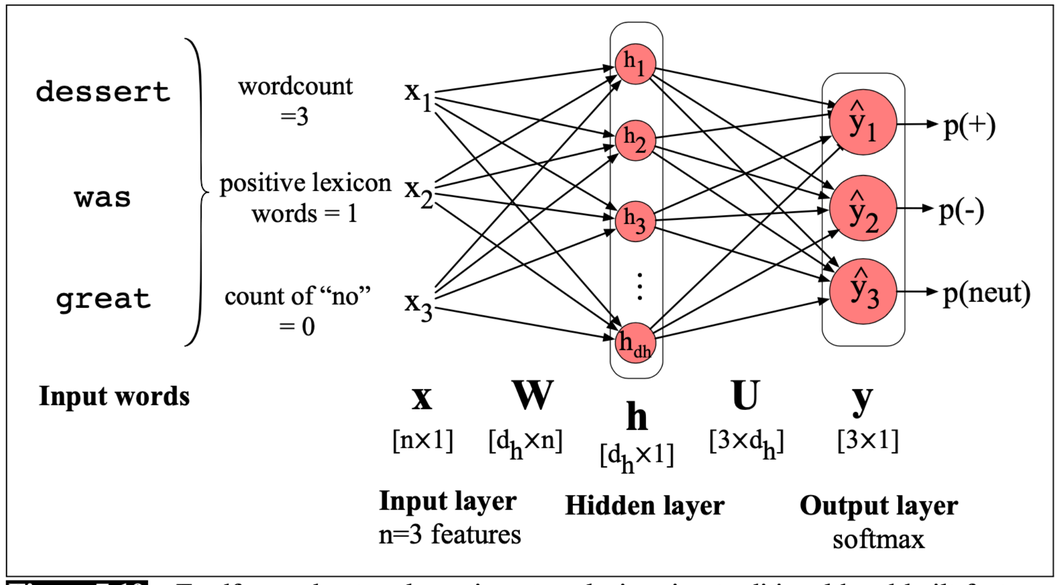



**Figure 7.10** Feedforward network sentiment analysis using traditional hand-built features of the input text

## Option 2 - more common – we allow the neural network to come up with features

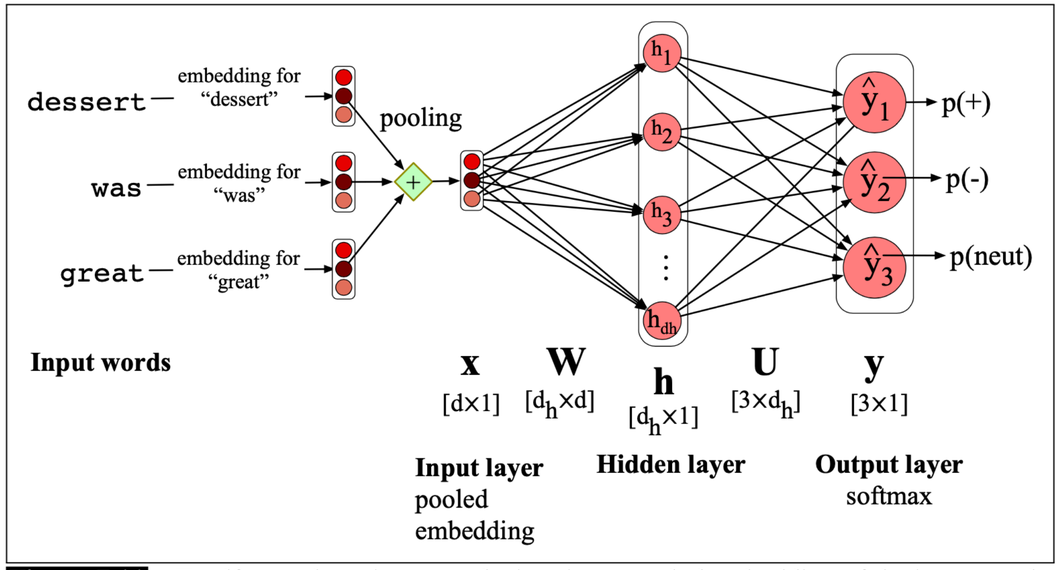



**Figure 7.11** Feedforward sentiment analysis using a pooled embedding of the input words.

- We use the word embedding vectors as inputs
- There can also be a pooling layer that averages the embeddings element-wise, or takes an element-wise max of the embedding vectors

# LANGUAGE MODELING WITH NEURAL NETWORKS

- Recall that in language modeling, we predict upcoming words from prior word context
  - Used in machine translation, summarization, grammar correction, dialog, etc.
- We'll use a feedforward neural network for our initial examples
  - We'd typically use more advanced neural network architectures that a feed-forward neural network (recurrent nets, transformers)

# NEURAL NETWORKS VS. N-GRAM MODELS

- Neural networks
  - Can generalize better
  - Can handle longer context
  - Are more accurate at word prediction
- N-gram models
  - Simpler
  - Faster to train
  - Lower compute requirements
  - More interpretable
  - Preferred for many smaller tasks

Example: “I have to make sure that the cat gets fed” also enabled “gets fed” to be generated after the word “dog” when that combination has not been seen before

The NN can predict “fed” after “the dog gets” because “cat” and “dog” have similar embeddings

The n-gram language model cannot

# NEURAL FEED FORWARD LANGUAGE MODEL

Neural language modeling using a feed-forward network

- Input: some number of previous words
  - Words are represented by vector embeddings in neural networks, as opposed to word IDs in the vocabulary in n-gram LMs
  - As in n-gram LM, we assume that we can estimate the entire prior context by looking at only the last few words: $P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-N+1}, \ldots, w_{t-1})$
- Output: probability distribution over possible next words

# NEURAL LANGUAGE MODELING USING A FEED FORWARD NETWORK

- The input is a sequence of words, and we try to predict the next word
- The output is the probability of each word in the vocabulary

Additional details

- The input is a 1-hot vector for each word which serves as an input index into an embedding matrix E
- The embedding matrix E has a column (embedding vector) for each word in the vocabulary
- This initial vector for each word in E is the initial embedding for the word that we have
- When treated as a weight matrix (that can be updated), E contains the fine tuned word embeddings at the end of the training phase

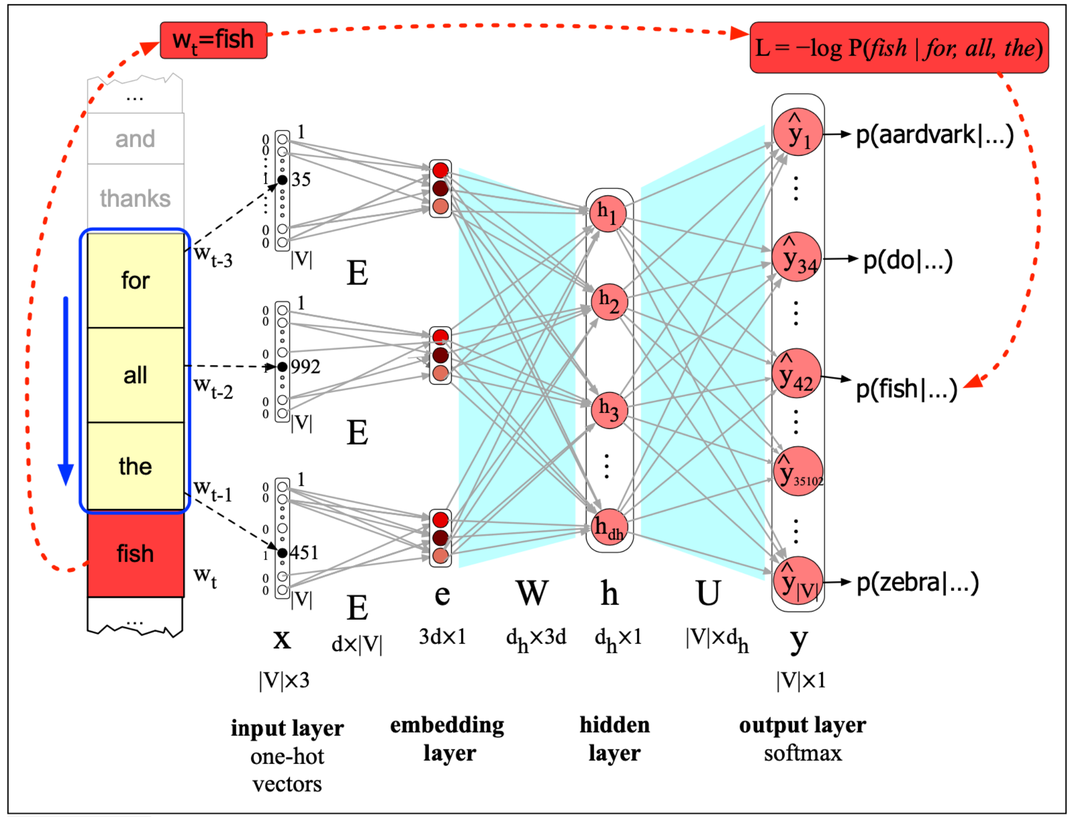



**Figure 7.18** Learning all the way back to embeddings. Again, the embedding matrix **E** is shared among the 3 context words.

# HANDLING OVERFITTING IN NEURAL NETWORKS: WEIGHT DECAY

Recall L2 regularization for logistic regression which adds a penalty term to the loss function:

Our original loss function

$$Regularized\ Loss = L(w) + \lambda \sum_i w_i^2$$

penalty term

$\lambda$: a parameter to control the level of regularization

$w_j$: model coefficients (or weights) of the model we are trying regularize

Among parameters that give you models with minimum/ low loss, choose the one that has the lowest-valued sum of the *squares* of the model coefficients

- Similarly, **weight decay** is a regularization approach for neural networks which involves adding a penalty to the loss function

Our original loss function

$$Regularized\ Loss = Loss(h_w) + \lambda \sum_{i,j} W_{i,j}^2$$

penalty term

*There is some intuition behind why this works, but not really a full understanding of it.*

# HANDLING OVERFITTING IN NEURAL NETWORKS: DROPOUT

- Dropout is used during training to make it more difficult for the neural network to memorize (fit too closely) to the training set
- Applied to SGD, where for each batch, we zero out the output of units in each layer with some probability (which is effectively deactivating the units)
  - P = 0.5 has been found to be effective when applied to units in the inner layers
  - P = 0.8 has been found to be effective when applied to units in the input layer
- Some intuition behind dropout (note: cannot be fully validated theoretically)
  - We are effectively forcing the model to learn in the presence of noise which makes it more robust
  - Dropout results in an ensemble of multiple “thinned out” networks - ensembles have been shown to be powerful

# SUMMARY OF KEY PROPERTIES OF NEURAL NETWORKS

- Can model very complex functions
- Typically more data is needed (than in other models) because the relationships are more complex, and there are more coefficients
- # of hidden units, # layers -> more complex model
- Regularization is needed
- Normalization of the input is needed

# WHAT IS DEEP LEARNING?

- When you have 1 hidden layer, the network is deemed shallow

- Deep learning methods use neural networks with two or more hidden layers
  - Each hidden layer computes higher layer features based on the lower layer features computed by the preceding hidden layer, e.g., first hidden layer detects lines, the next hidden layer detects shapes composed of the lines
  - Deep learning provides a way to do end-to-end learning because it provides good automated feature extraction

# NEURAL NETWORK ARCHITECTURES

- Empirical research has uncovered network architectures that are effective on specific problem classes, e.g., CNNs are effective on images, RNNs for NLP tasks, etc.
- Initial evidence suggests, networks that are deeper (many layers) tend to perform better than networks that are shallow
- Deep learning models tend to produce unintuitive errors where small changes to the input result in very different output
  - Adversarial examples are able to confuse neural networks by making very small modifications to the inputs

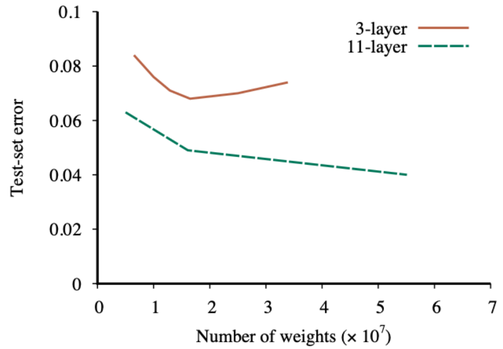



**Figure 21.7** Test-set error as a function of layer width (as measured by total number of weights) for three-layer and eleven-layer convolutional networks. The data come from early versions of Google's system for transcribing addresses in photos taken by Street View cars

# NEURAL NETWORK HYPERPARAMETERS

- Model architecture (number of layers, number of units per layer)
- Activation functions
- Learning rate
- Gradient descent variant
- Learning rate
- Mini-batch size
- Regularization approach and parameters

# DEEP LEARNING NOTES

- More data *along with* bigger networks have resulted in increasingly higher deep learning performance
  - The GPT-3 (language generation model) has 175 billion model parameters, trained on ~500 billion words
- Over the last few years there have been a few algorithmic improvements that have led to more efficient computation
  - E.g., going from sigmoid to a ReLU activation function significantly sped up gradient descent
- Deep learning algorithms are black boxes - hard to understand how and why a particular prediction was made