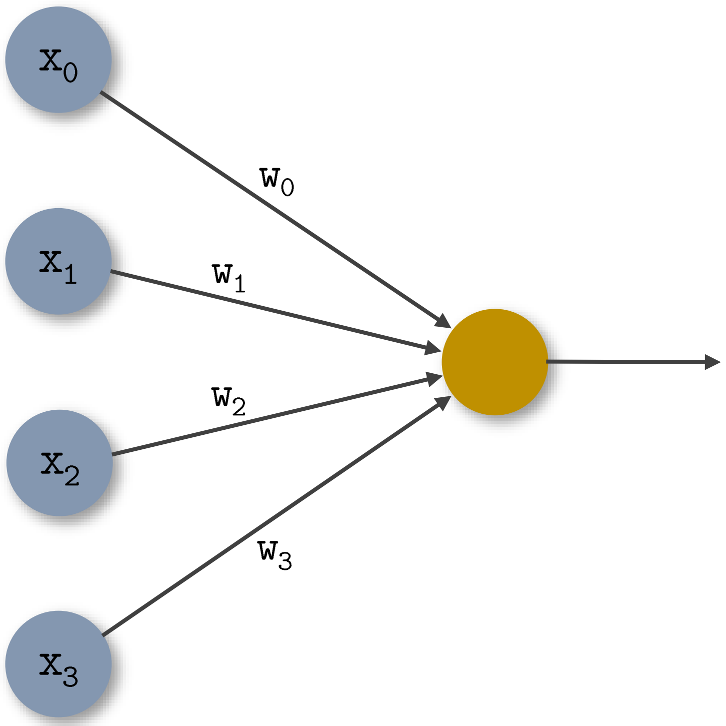
CS 490: NATURAL LANGUAGE PROCESSING

Dan Goldwasser, Abulhair Saparov

Lecture 5: Neural Networks II

PREVIOUSLY: PERCEPTRON

input neurons:
a real value is
assigned to each



Let:

$\mathbf{x} = [x_0,\ x_1,\ x_2,\ x_3]$

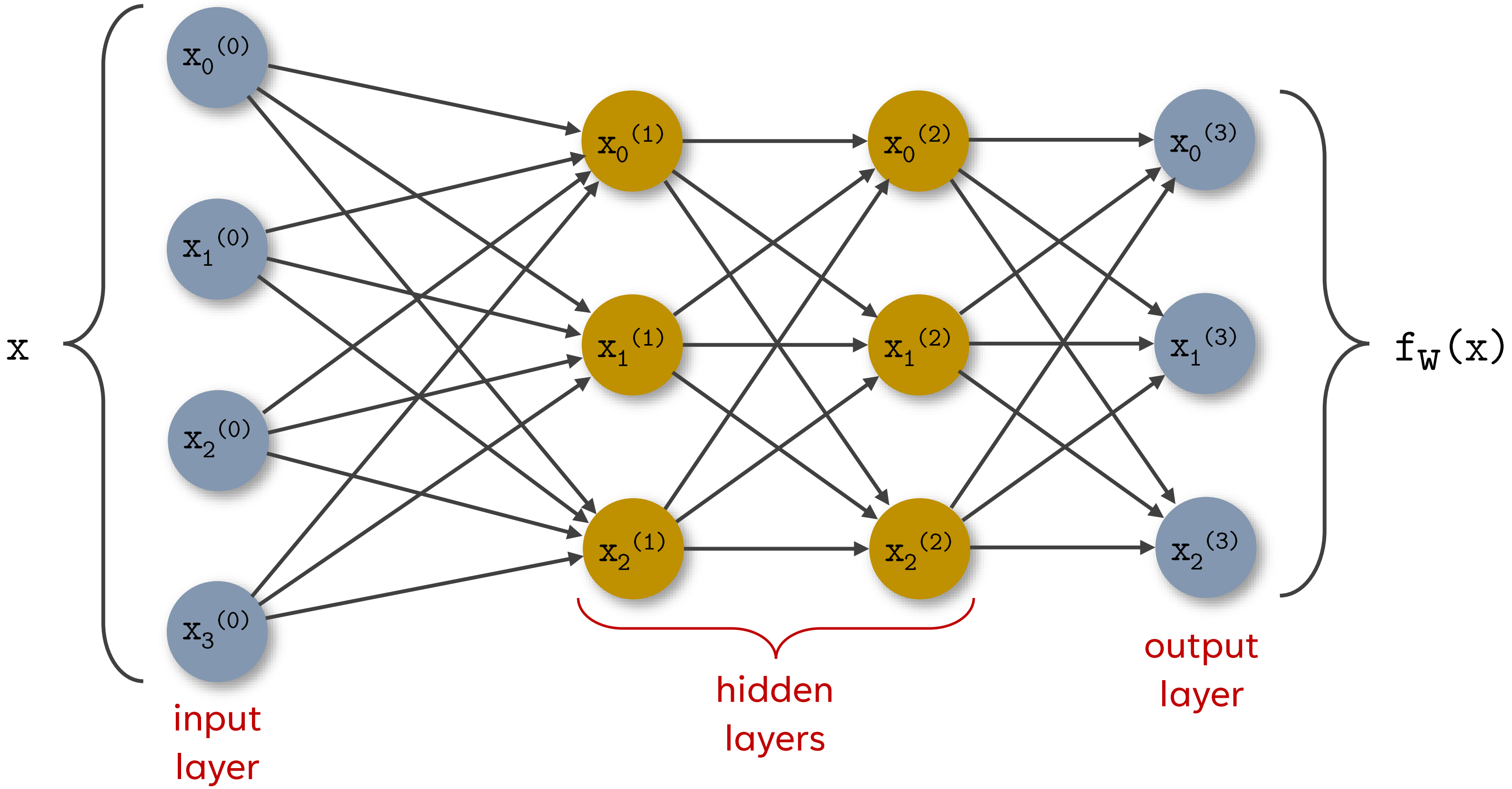
$\mathbf{w} = [w_0,\ w_1,\ w_2,\ w_3]$

single output neuron:

$f_w(x) = \mathrm{sign}\{\mathbf{w}^T\mathbf{x}\}$

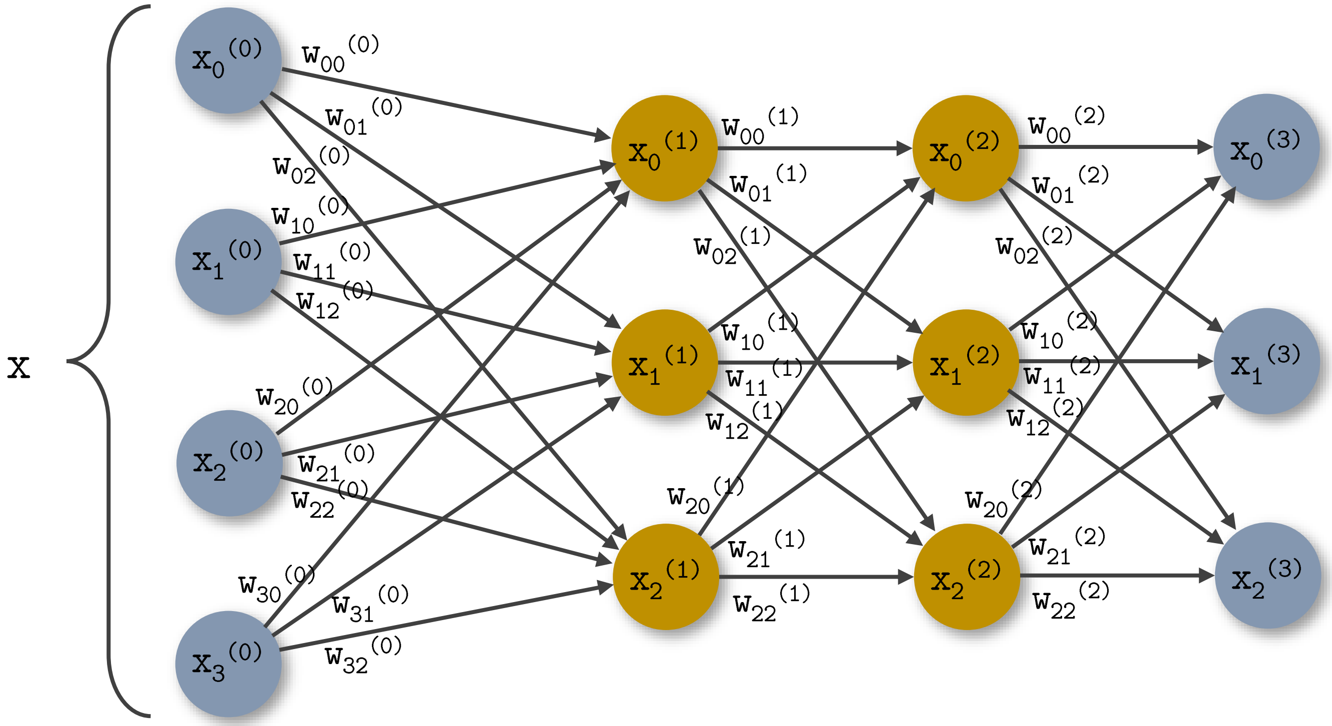
MULTI-LAYER PERCEPTRON

- We can swap f with other machine learning models.



MULTI-LAYER PERCEPTRON

- We can swap $\texttt{f}$ with other machine learning models.



Suppose we're given some input $\mathtt{x}$.
(that is, we're given $\mathtt{x}^{(0)}$)

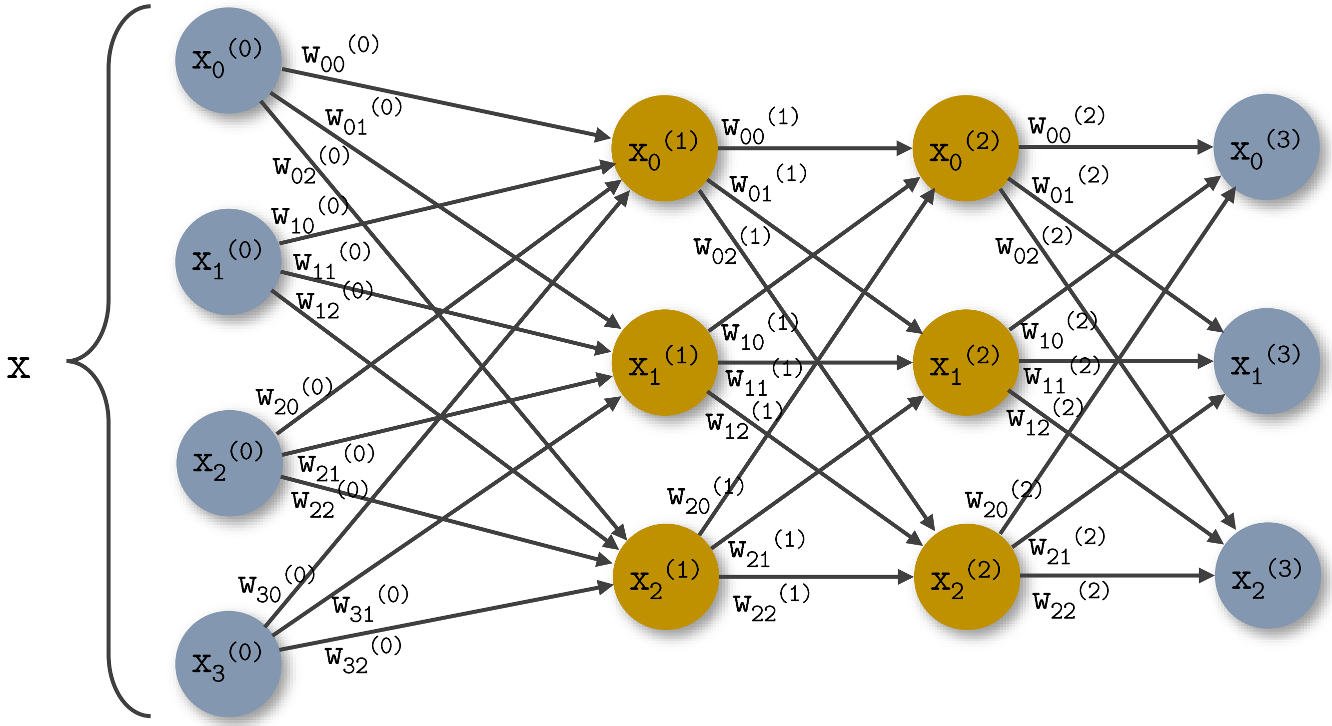
How do we compute $\mathtt{x}_0{}^{(1)}$, for example?
(the first neuron in the second layer)

$$
\begin{aligned}
\mathtt{x}_0{}^{(1)} &= \mathtt{g}\left(\textstyle\sum_{i=0}^{4} \mathtt{w}_{i0}^{(0)}\, \mathtt{x}_i^{(0)}\right) \\
&= \mathtt{g}([\mathtt{w}_{00}{}^{(0)}, \mathtt{w}_{10}{}^{(0)}, \mathtt{w}_{20}{}^{(0)}, \mathtt{w}_{30}{}^{(0)}] \cdot \mathtt{x}^{(0)}) \\
&= \mathtt{g}((\mathtt{w}_{:0}{}^{(0)})^{\mathsf{T}}\mathtt{x}^{(0)}) \\
\mathtt{x}_1{}^{(1)} &= \mathtt{g}((\mathtt{w}_{:1}{}^{(0)})^{\mathsf{T}}\mathtt{x}^{(0)}) \\
\mathtt{x}_2{}^{(1)} &= \mathtt{g}((\mathtt{w}_{:2}{}^{(0)})^{\mathsf{T}}\mathtt{x}^{(0)})
\end{aligned}
$$

Note: g is an activation function.
E.g.: $\mathtt{g}(t) = \mathtt{sign}(t)$
or $\mathtt{g}(t) = \mathtt{tanh}(t)$

MULTI-LAYER PERCEPTRON

- We can swap $\mathtt{f}$ with other machine learning models.



We can write this equivalently as a matrix multiplication:

$$\begin{bmatrix} x_0^{(1)} \\ x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = g\left(\begin{bmatrix} w_{00}^{(0)} & w_{10}^{(0)} & w_{20}^{(0)} & w_{30}^{(0)} \\ w_{01}^{(0)} & w_{11}^{(0)} & w_{21}^{(0)} & w_{31}^{(0)} \\ w_{02}^{(0)} & w_{12}^{(0)} & w_{22}^{(0)} & w_{32}^{(0)} \end{bmatrix} \begin{bmatrix} x_0^{(0)} \\ x_1^{(0)} \\ x_2^{(0)} \\ x_3^{(0)} \end{bmatrix} \right)$$

here, g is applied element-wise

$$x^{(1)} = g(W^{(0)} x^{(0)})$$

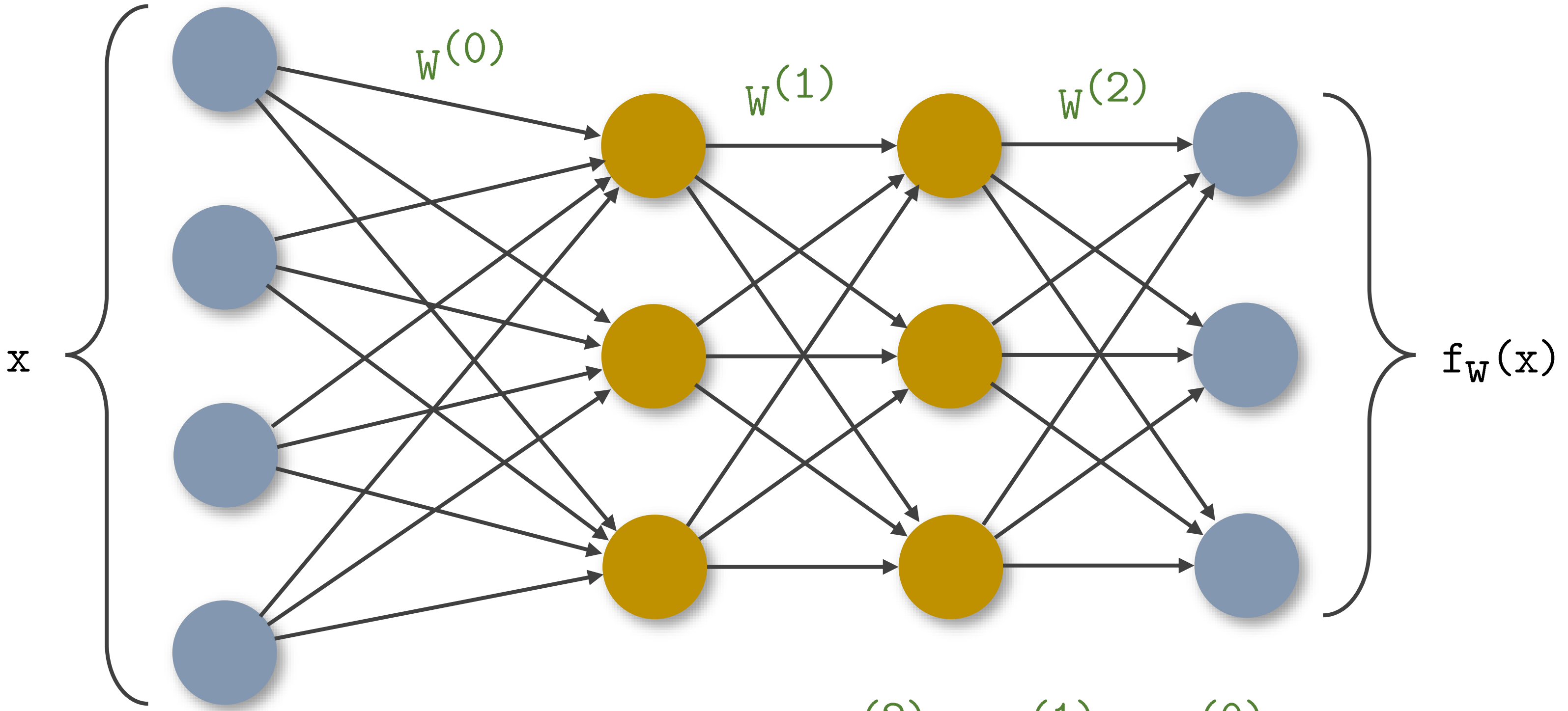
We can similarly compute the other layers:

$$x^{(2)} = g(W^{(1)} x^{(1)})$$
$$x^{(3)} = g(W^{(2)} x^{(2)})$$

The output of the MLP is $x^{(3)}$!

MULTI-LAYER PERCEPTRON

- We can swap $\mathtt{f}$ with other machine learning models.



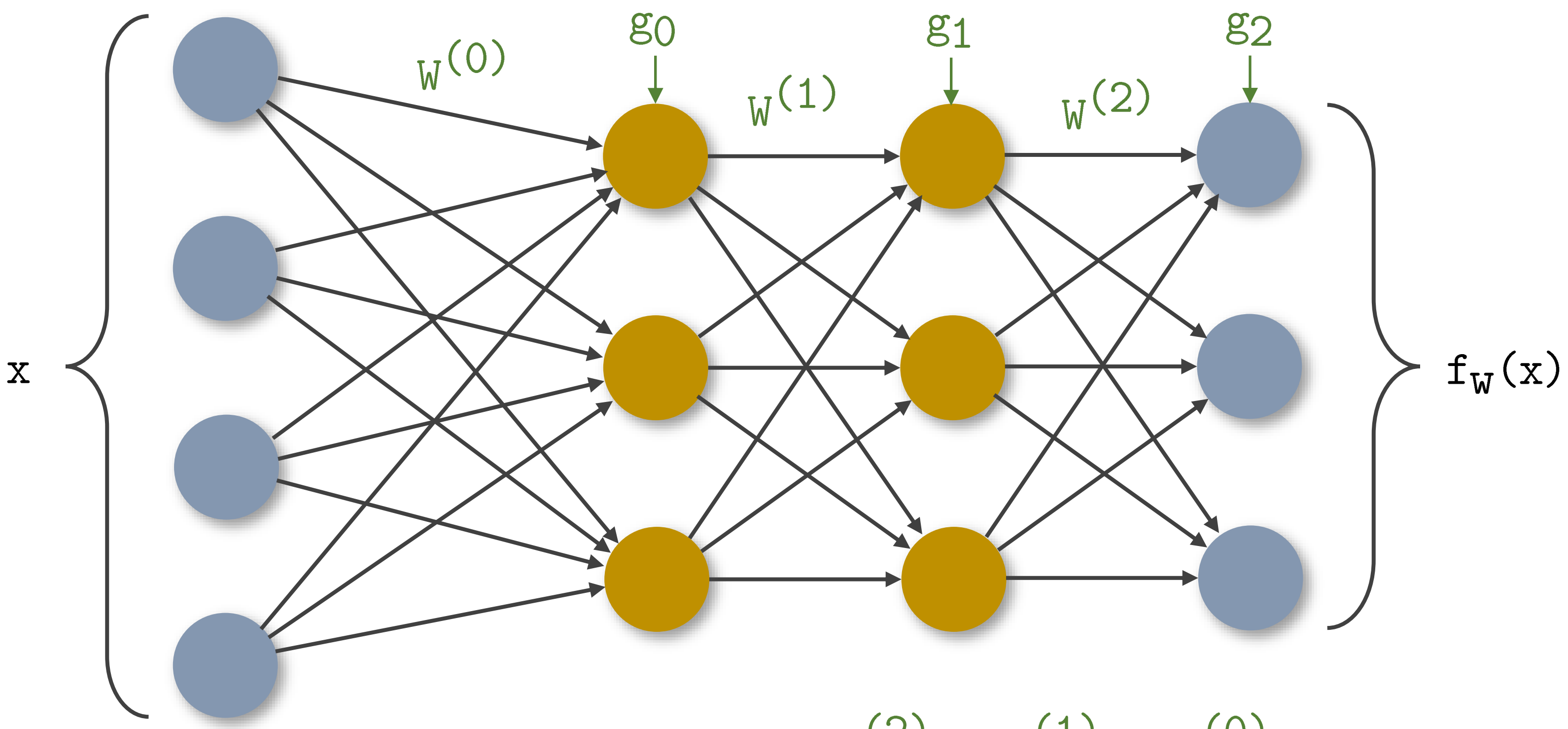
$$\mathtt{f}_{\mathtt{W}}(\mathtt{x}) = \mathtt{g}_2(\mathtt{W}^{(2)}\mathtt{g}_1(\mathtt{W}^{(1)}\mathtt{g}_0(\mathtt{W}^{(0)}\mathtt{x})))$$

$\mathtt{W}^{(0)}$ are the connection weights in the first layer.

$\mathtt{W}^{(0)}$ is a matrix:
Number of rows is the number of neurons in the next layer.
Number of columns is the number of neurons in the previous layer.

$\mathtt{W}^{(0)}_{ij}$ is the connection weight from neuron j in the previous layer to neuron i in the next.

MULTI-LAYER PERCEPTRON



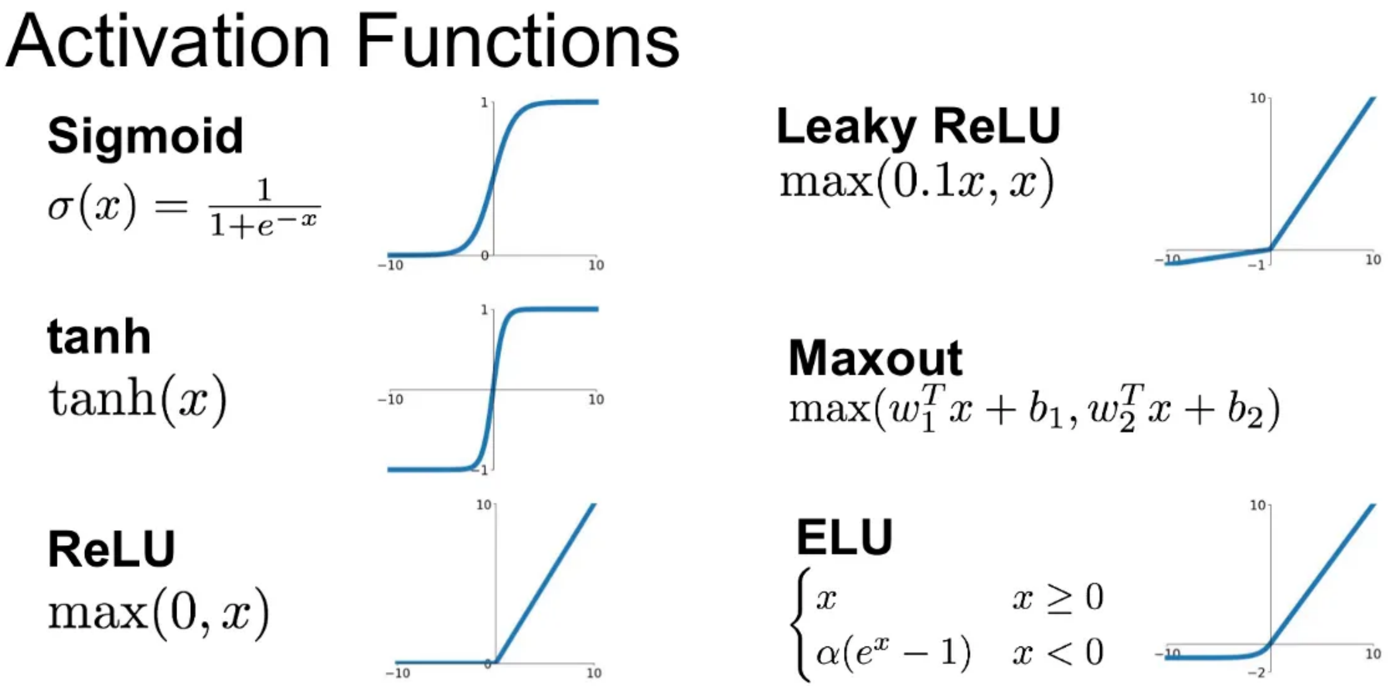
$$f_W(x) = g_2(W^{(2)} g_1(W^{(1)} g_0(W^{(0)} x)))$$

g_0, g_1, and g_2 are activation functions.

They must be non-linear since otherwise, adjacent layers would collapse into a single linear transformation.

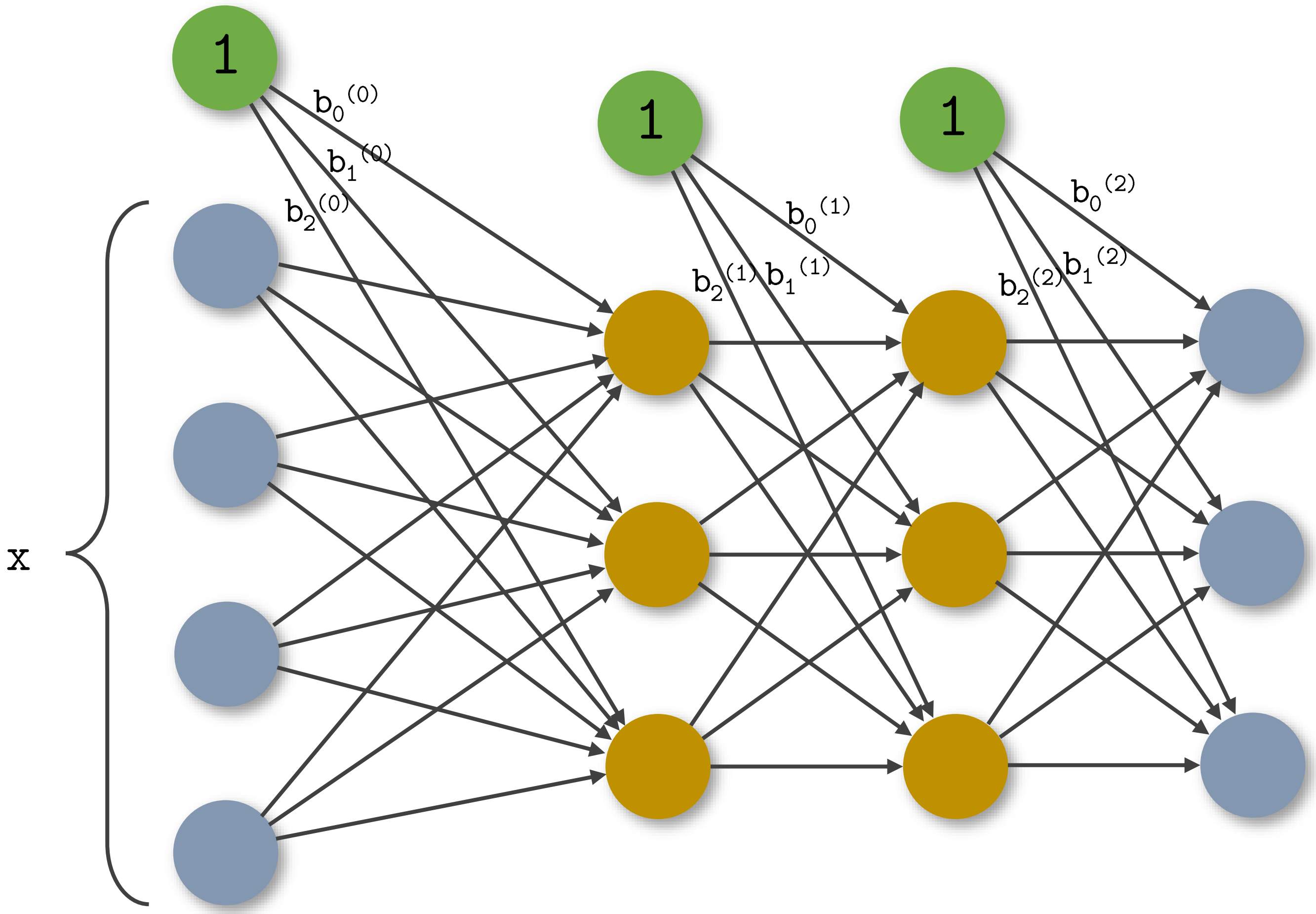
(compositions of linear functions are linear)

MULTI-LAYER PERCEPTRON



[Shruti Jadon, Introduction to Different Activation Functions for Deep Learning]

BIAS TERMS

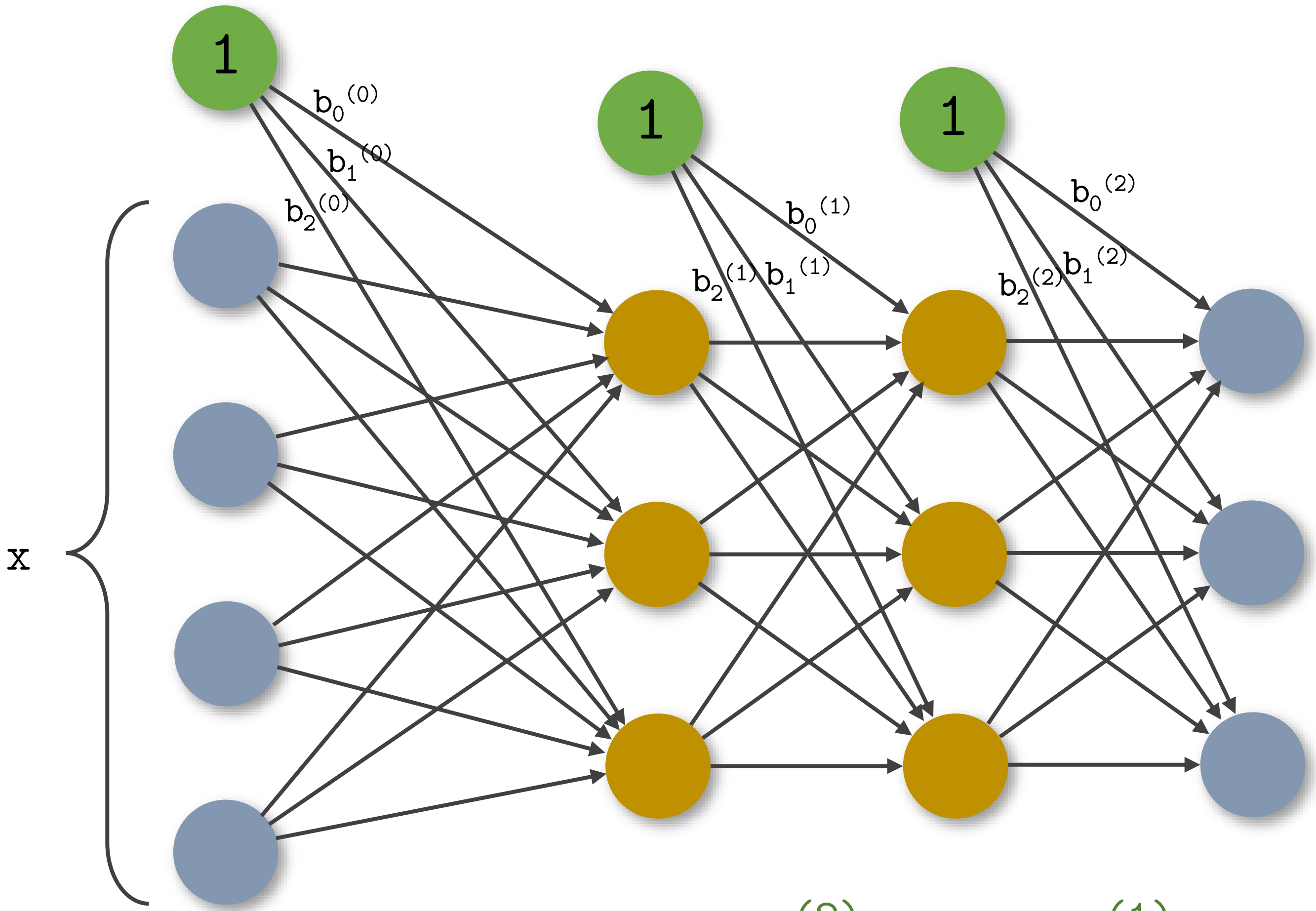


MLPs can have a bias term at each layer.

One perspective: There is an extra "bias neuron" in the previous layer with activation 1.

$$\begin{aligned} x_0^{(1)} &= g\left(\sum_{i=0}^{4} w_{i0}^{(0)} x_i^{(0)} + b_0^{(0)}\right) \\ &= g\left((w_{:0}^{(0)})^T x^{(0)} + b_0^{(0)}\right) \end{aligned}$$

BIAS TERMS



In terms of matrix multiplication:

$$\begin{bmatrix} x_0^{(1)} \\ x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = g\left(\begin{bmatrix} w_{00}^{(0)} & w_{10}^{(0)} & w_{20}^{(0)} & w_{30}^{(0)} \\ w_{01}^{(0)} & w_{11}^{(0)} & w_{21}^{(0)} & w_{31}^{(0)} \\ w_{02}^{(0)} & w_{12}^{(0)} & w_{22}^{(0)} & w_{32}^{(0)} \end{bmatrix} \begin{bmatrix} x_0^{(0)} \\ x_1^{(0)} \\ x_2^{(0)} \\ x_3^{(0)} \end{bmatrix} + \begin{bmatrix} b_0^{(0)} \\ b_1^{(0)} \\ b_2^{(0)} \end{bmatrix} \right)$$

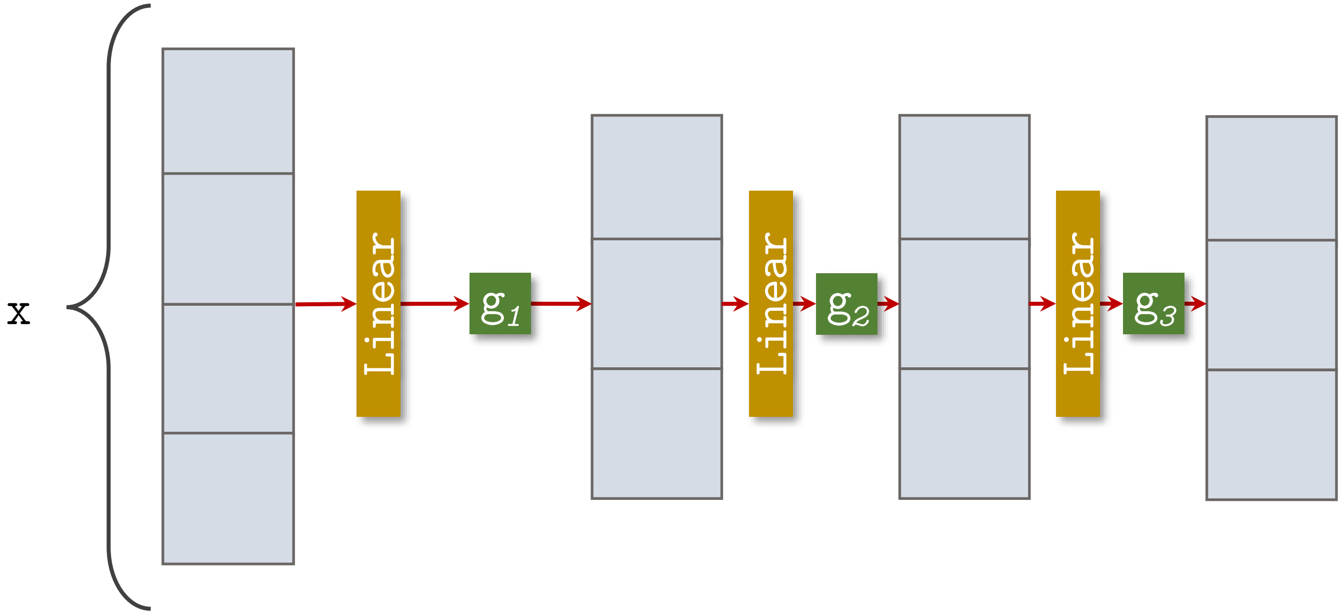
$$x^{(1)} = g(W^{(0)}x^{(0)} + b^{(0)})$$

Similarly, for the other layers:

$$x^{(2)} = g(W^{(1)}x^{(1)} + b^{(1)})$$
$$x^{(3)} = g(W^{(2)}x^{(2)} + b^{(2)})$$

$$f_W(x) = g_2(b^{(2)} + W^{(2)}g_1(b^{(1)} + W^{(1)}g_0(b^{(0)} + W^{(0)}x)))$$

MULTI-LAYER PERCEPTRON



Large neural networks are difficult to draw, especially if we draw every neuron/connection.

Instead, they are commonly drawn in a schematic style like in this example.

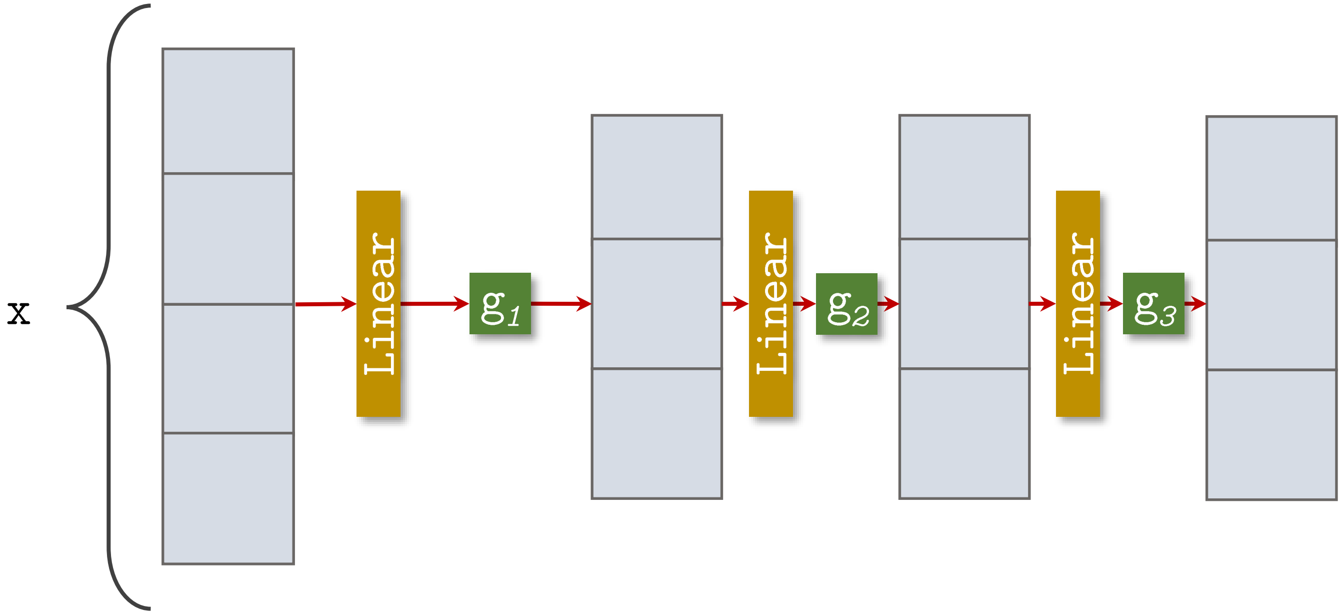
This style makes it easier to interpret the activations at each layer as a vector.

The operations (e.g., `Linear`, g_1) are drawn as blocks.

A linear layer is simply a matrix-vector product plus a bias: $\mathtt{Linear}(t) = \mathtt{W}t + \mathtt{b}$

$$f_{\mathtt{W}}(\mathtt{x}) = g_2(\mathtt{b}^{(2)} + \mathtt{W}^{(2)} g_1(\mathtt{b}^{(1)} + \mathtt{W}^{(1)} g_0(\mathtt{b}^{(0)} + \mathtt{W}^{(0)}\mathtt{x})))$$

MULTI-LAYER PERCEPTRON



A quick note on nomenclature:

MLPs are also sometimes called fully-connected neural networks,
or feedforward (FF) networks,
or fully-connected FF networks.

Linear layers are also sometimes referred to as fully-connected layers or FF layers.

$$f_W(x) = g_2(b^{(2)} + W^{(2)} g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x)))$$

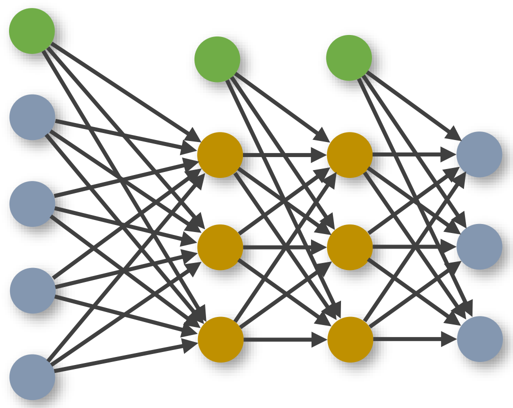
TRAINING MLPS

- How do we learn the weight matrices $\mathtt{W}$ and bias vectors $\mathtt{b}$, given a training dataset?
- We can use gradient descent!
  - We just need to compute the gradient of the loss function with respect to each weight and bias parameter.
- Suppose we have an MLP with n layers.

$$f_W(x) = g_n(b^{(n)} + W^{(n)} \ldots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x)))$$

- And we have a loss function $\mathtt{C}$, so the objective function we want to minimize is:

$$L(W) = C(g_n(b^{(n)} + W^{(n)} \ldots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x))))$$



TRAINING MLPS

- We want to minimize:

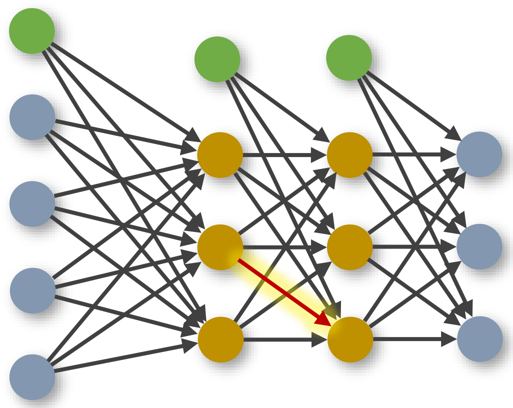
$$L(W) = C(g_n(b^{(n)} + W^{(n)} \dots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x))))$$

- Let's compute the gradient with respect to the weight of one connection in layer l: $w_{ij}^{(l)}$
- We know C is a function of the activations of the last layer $x^{(n)}$.
- So we can apply the chain rule:

$$\frac{dL}{dw_{ij}^{(l)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dw_{ij}^{(l)}}$$

- The last layer activations $x^{(n)}$ is a function of $w_{ij}^{(l)}$, so we apply the chain rule again:

$$\frac{dL}{dw_{ij}^{(l)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \frac{dx^{(n-1)}}{dw_{ij}^{(l)}}$$



C needs to be differentiable, and we also need to compute $x^{(n)}$ before we can compute this derivative.

TRAINING MLPS

- We want to minimize:

$$L(W) = C(g_n(b^{(n)} + W^{(n)} \dots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x))))$$

- We repeatedly apply the chain rule until we reach layer $l+1$:

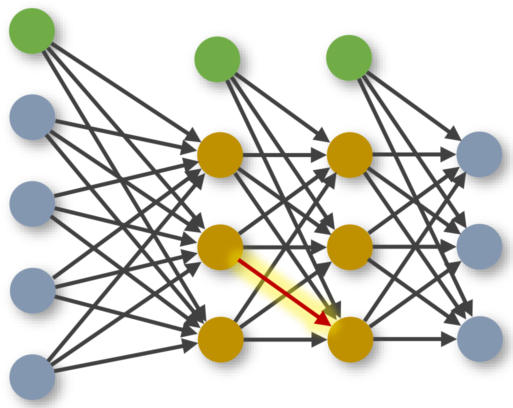
$$\frac{dL}{dw_{ij}^{(l)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \cdots \frac{dx^{(l+2)}}{dx^{(l+1)}} \frac{dx^{(l+1)}}{dw_{ij}^{(l)}}$$

- How do we compute $\frac{dx^{(l+1)}}{dw_{ij}^{(l)}}$?
- We can write $x^{(l+1)}$ as a function of $w_{ij}^{(l)}$:

$$x_j^{(l+1)} = g\left(\sum_{i=0}^{m} w_{ij}^{(l)} x_i^{(l)} + b_j^{(l)}\right)$$

$$\frac{dx_j^{(l+1)}}{dw_{ij}^{(l)}} = g'\left(\sum_{i=0}^{m} w_{ij}^{(l)} x_i^{(l)} + b_j^{(l)}\right) \cdot x_i^{(l)}$$

$$\frac{dx_k^{(l+1)}}{dw_{ij}^{(l)}} = 0 \quad \text{for any } k \neq j$$



We need to compute $x^{(l)}$ before we can compute this, and we need g to be differentiable.

So we can simply run a forward pass to compute all activations $x^{(l)}$ beforehand.

TRAINING MLPS

- We want to minimize:

$$L(W) = C(g_n(b^{(n)} + W^{(n)} \ldots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x))))$$

- We repeatedly apply the chain rule until we reach layer $l+1$:

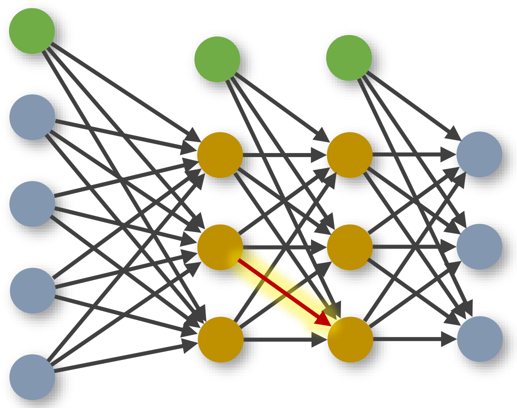
$$\frac{dL}{dw_{ij}^{(l)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \cdots \frac{dx^{(l+2)}}{dx^{(l+1)}} \frac{dx^{(l+1)}}{dw_{ij}^{(l)}}$$

- How do we compute $\frac{dx^{(l+1)}}{dw_{ij}^{(l)}}$?
- So the gradient $\frac{dx^{(l+1)}}{dw_{ij}^{(l)}}$ is: (notice there is no j subscript for $x^{(l+1)}$)

A vector with dimension equal to the number of neurons in layer $l+1$.

The vector contains all zeros except for the non-zero term at index j:

$$g'(\textstyle\sum_{i=0}^{m} w_{ij}^{(l)} x_i^{(l)} + b_j^{(l)}) \cdot x_i^{(l)}$$



TRAINING MLPS

- We want to minimize:

$$L(W) = C(g_n(b^{(n)} + W^{(n)} \dots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x))))$$

- We repeatedly apply the chain rule until we reach layer $l+1$:

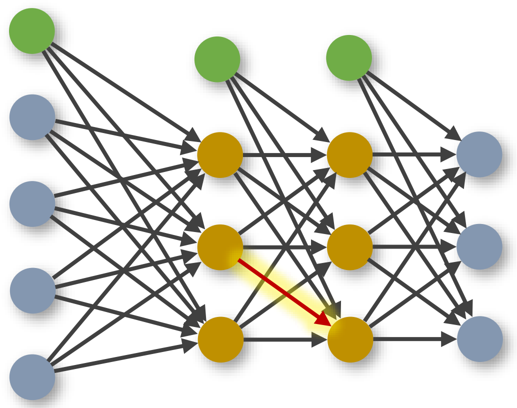
$$\frac{dL}{dw_{ij}^{(l)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \cdots \frac{dx^{(l+2)}}{dx^{(l+1)}} \frac{dx^{(l+1)}}{dw_{ij}^{(l)}}$$

- How do we compute $\frac{dx^{(n)}}{dx^{(n-1)}}$? (for any n, not just the last layer)
- We can write $x^{(n)}$ as a function of $x^{(n-1)}$:

$$x_j^{(n)} = g(\textstyle\sum_{k=0}^{m} w_{kj}^{(n-1)} x_k^{(n-1)} + b_j^{(n-1)})$$

$$\frac{dx_j^{(n)}}{dx_i^{(n-1)}} = g'(\textstyle\sum_{k=0}^{m} w_{kj}^{(n-1)} x_k^{(n-1)} + b_j^{(n-1)}) \cdot w_{ij}^{(n-1)}$$

- If we compute the above for each i and j, we get a matrix!



TRAINING MLPS

- We want to minimize:

$$L(W) = C(g_n(b^{(n)} + W^{(n)} \dots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x))))$$

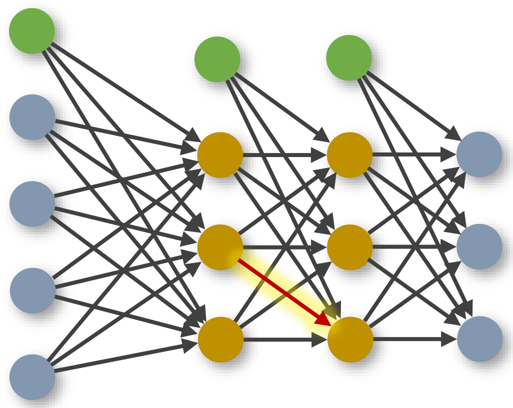
- We repeatedly apply the chain rule until we reach layer $l+1$:

$$\frac{dL}{dw_{ij}^{(l)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \cdots \frac{dx^{(l+2)}}{dx^{(l+1)}} \frac{dx^{(l+1)}}{dw_{ij}^{(l)}}$$

- How do we compute $\frac{dx^{(n)}}{dx^{(n-1)}}$?
- $\frac{dx^{(n)}}{dx^{(n-1)}}$ is a matrix where the element at i, j is:

$$\frac{dx_j^{(n)}}{dx_i^{(n-1)}} = g'(\textstyle\sum_{k=0}^{m} w_{kj}^{(n-1)} x_k^{(n-1)} + b_j^{(n-1)}) \cdot w_{ij}^{(n-1)}$$

- This matrix has the same dimensions as $W^{(n-1)}$.
  - The number of rows is the number of neurons in layer $n-1$.
  - The number of columns is the number of neurons in layer n.



TRAINING MLPS

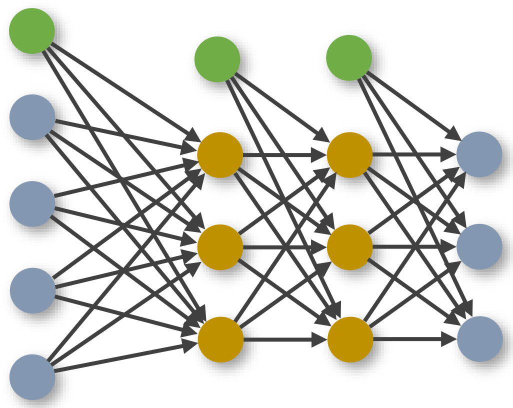
- We want to minimize:

$$L(W) = C(g_n(b^{(n)} + W^{(n)} \dots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x))))$$

- We repeatedly apply the chain rule until we reach layer $l+1$:

$$\frac{dL}{dw_{ij}^{(l)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \cdots \frac{dx^{(l+2)}}{dx^{(l+1)}} \frac{dx^{(l+1)}}{dw_{ij}^{(l)}}$$

- Now we have all the ingredients to compute the above expression.
- Note the first term is a vector, followed by a bunch of matrices, and the last term is a vector.
    - So their product is a scalar.
    - (whenever we have matrix products, it's usually a good idea to check their dimensions and make sure they're consistent)



TRAINING MLPS

- We want to minimize:

$$L(W) = C(g_n(b^{(n)} + W^{(n)} \dots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x))))$$

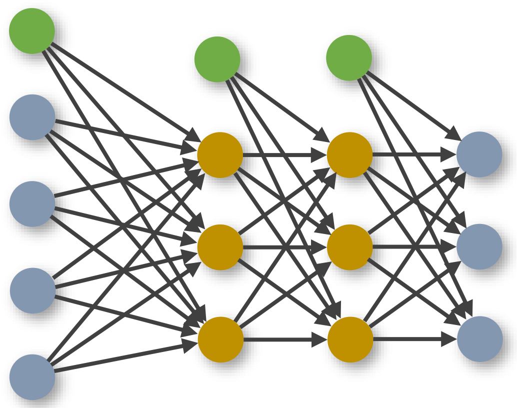
- We repeatedly apply the chain rule until we reach layer $l+1$:

$$\frac{dL}{dw_{ij}^{(l)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \cdots \frac{dx^{(l+2)}}{dx^{(l+1)}} \frac{dx^{(l+1)}}{dw_{ij}^{(l)}}$$

- Now we have all the ingredients to compute the above expression.
- Notice that for the derivative of the weight of a connection in any layer $<l$ will also contain the terms:

$$\frac{dx^{(n)}}{dx^{(n-1)}} \cdots \frac{dx^{(l+2)}}{dx^{(l+1)}}$$

- Repeatedly computing these matrix products would be redundant.
- How do we avoid this redundant computation?



TRAINING MLPS

- We want to minimize:

$$L(W) = C(g_n(b^{(n)} + W^{(n)} \ldots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)}x))))$$

- We repeatedly apply the chain rule until we reach layer $l+1$:

$$\frac{dL}{dw_{ij}^{(l)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \cdots \frac{dx^{(l+2)}}{dx^{(l+1)}} \frac{dx^{(l+1)}}{dw_{ij}^{(l)}}$$

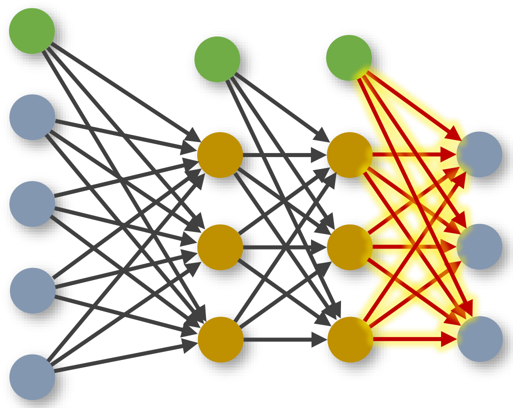
- We go backwards, starting with computing the gradients of the weights in the last layer.

$$\frac{dL}{dw_{ij}^{(n-1)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dw_{ij}^{(n-1)}}$$

- Next, compute the gradients of the weights in the second-to-last layer:

$$\frac{dL}{dw_{ij}^{(n-2)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \frac{dx^{(n-1)}}{dw_{ij}^{(n-2)}}$$

- Note that we only need to compute the product $C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}}$ once.



TRAINING MLPS

- We want to minimize:

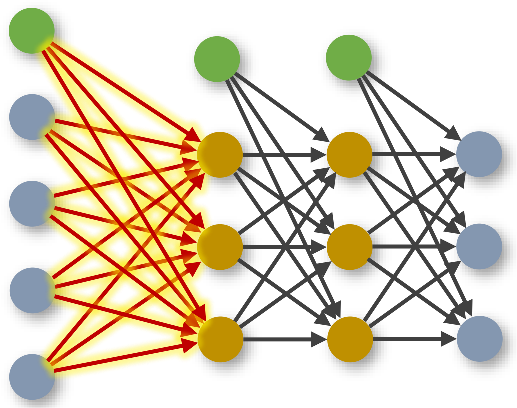
$$L(W) = C(g_n(b^{(n)} + W^{(n)} \dots g_1(b^{(1)} + W^{(1)} g_0(b^{(0)} + W^{(0)} x))))$$

- Next, compute the gradients of the weights in the third-to-last layer:

$$\frac{dL}{dw_{ij}^{(n-3)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \frac{dx^{(n-1)}}{dx^{(n-2)}} \frac{dx^{(n-2)}}{dw_{ij}^{(n-3)}}$$

- Multiply the quantity $C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}}$ with $\frac{dx^{(n-1)}}{dx^{(n-2)}}$ once and store it.
  - We re-use it to compute the gradient of all weights in this layer.
- Continue to the fourth-to-last layer:

$$\frac{dL}{dw_{ij}^{(n-4)}} = C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \frac{dx^{(n-1)}}{dx^{(n-2)}} \frac{dx^{(n-2)}}{dx^{(n-3)}} \frac{dx^{(n-3)}}{dw_{ij}^{(n-4)}}$$



- Multiply the quantity $C'(x^{(n)}) \frac{dx^{(n)}}{dx^{(n-1)}} \frac{dx^{(n-1)}}{dx^{(n-2)}}$ with $\frac{dx^{(n-2)}}{dx^{(n-3)}}$ once and store it.

BACKPROPAGATION

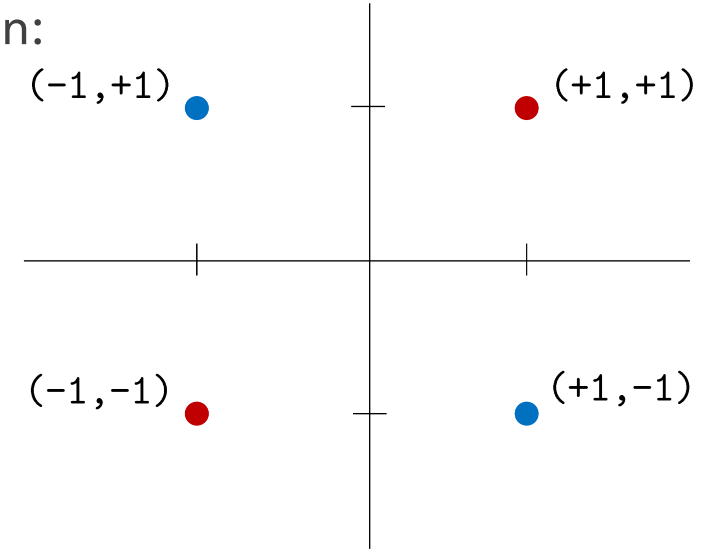
- Since we compute gradients backwards starting from the last layer, this algorithm is called backpropagation, or backprop.
- We first need to do a forward pass to compute the activations $\mathrm{x}^{(l)}$ at every layer l.
  - The forward pass starts from the first layer and progresses forwards.
- Then we perform a "backward pass" to compute the gradients at each layer.
  - The backward pass starts from the last layer and progresses backwards.
- The result is the gradient of the loss function with respect to every weight and bias.
- We can then perform a gradient step as in each iteration of gradient descent.

BACKPROPAGATION

- Minor detail: Our loss function only contained a single input $\mathtt{x}$.
$$\mathtt{L(W)} = \mathtt{C}(g_n(\mathtt{b}^{(n)} + \mathtt{W}^{(n)} \ldots g_1(\mathtt{b}^{(1)} + \mathtt{W}^{(1)} g_0(\mathtt{b}^{(0)} + \mathtt{W}^{(0)}\mathtt{x}))))$$
- In practice, the loss function typically contains a sum over multiple training examples.
$$\mathtt{L(W)} = \sum_{i=1}^{N} \mathtt{C}(g_n(\mathtt{b}^{(n)} + \mathtt{W}^{(n)} \ldots g_1(\mathtt{b}^{(1)} + \mathtt{W}^{(1)} g_0(\mathtt{b}^{(0)} + \mathtt{W}^{(0)}\mathtt{x}_i))))$$
- But the gradient/derivative is a linear operator, so we can easily move it inside the sum.
  - So we compute the gradient with respect to each training example, then compute the sum.
- Another idea: stack the training examples $\mathtt{x}_i$ into a matrix $\mathtt{X}$:
$$\mathtt{C}(g_n(\mathtt{b}^{(n)} + \mathtt{W}^{(n)} \ldots g_1(\mathtt{b}^{(1)} + \mathtt{W}^{(1)} g_0(\mathtt{b}^{(0)} + \mathtt{W}^{(0)}\mathtt{X}))))$$
- This approach is taken by most machine learning frameworks like `PyTorch`.

EXPRESSIVENESS OF MLPS

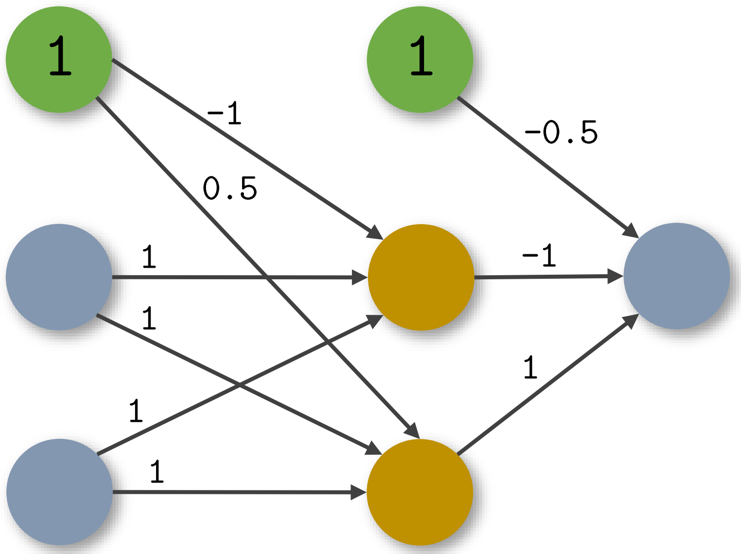
- MLPs, like all neural networks, can learn nonlinear decision boundaries, and can be very expressive (especially if there are many neurons/layers).
- But they need more data to train (and to avoid overfitting) than simpler models.
- Let's consider the XOR function:

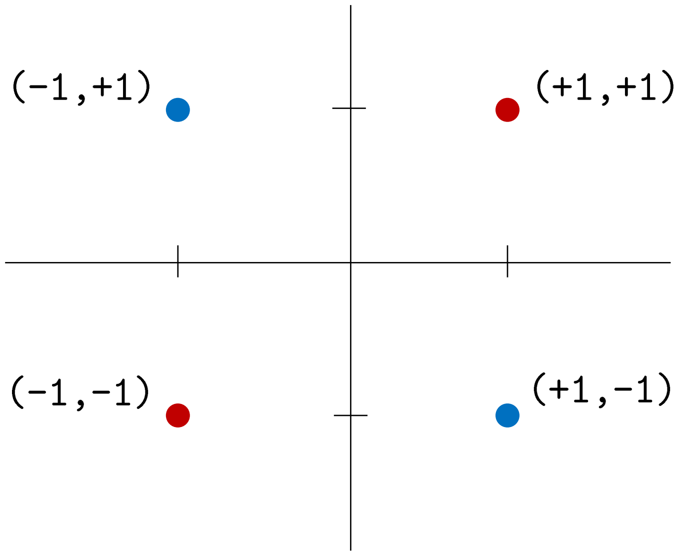


- We previously saw that linear classifiers are not able to perfectly classify this data.
  - Linear classifiers are unable to learn the XOR function.
  - (unless you use a feature function such as $\Phi(x,y)=(x,y,xy)$)

EXPRESSIVENESS OF MLPS

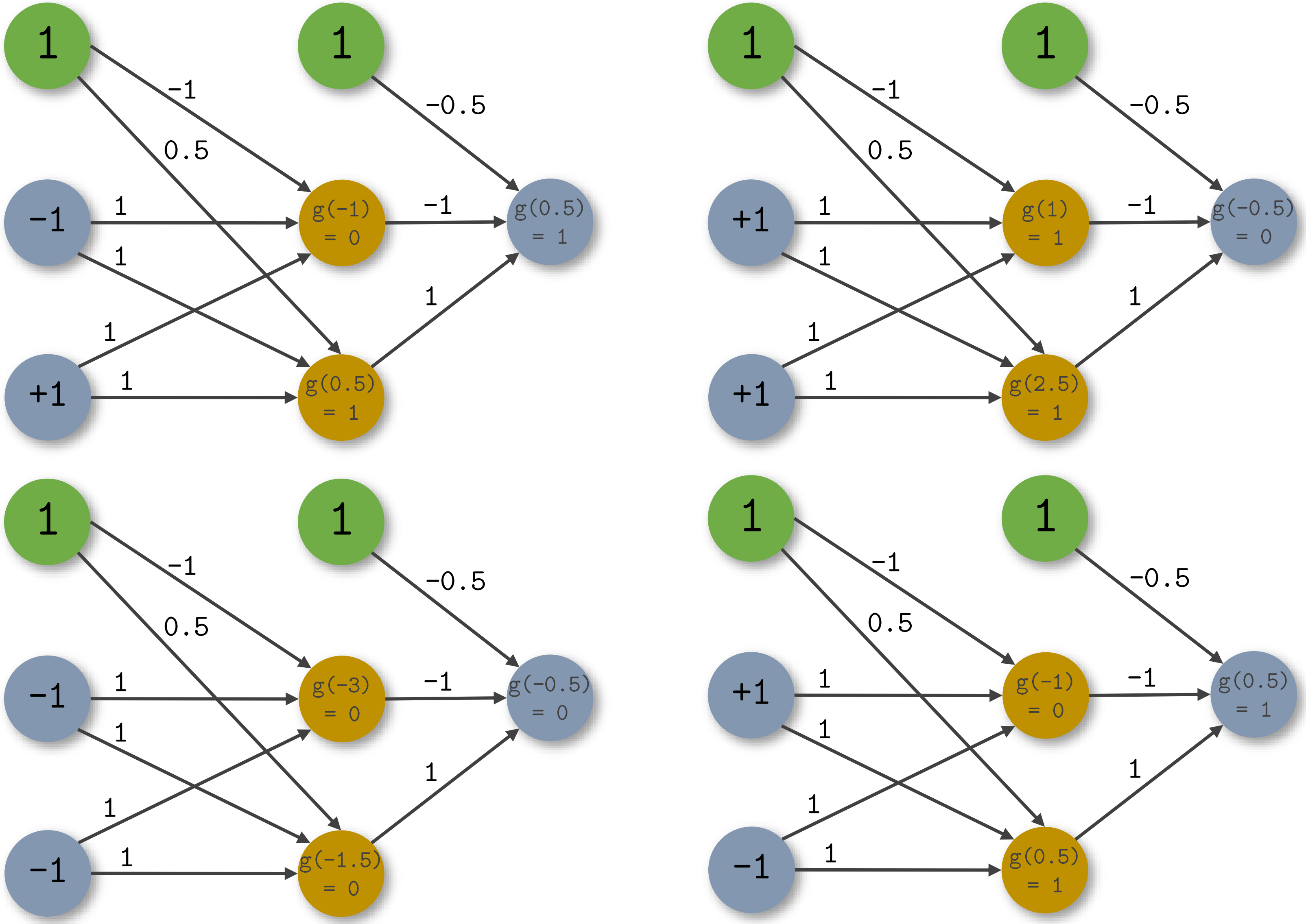
- An MLP can easily express/represent this function.
- Let's construct one that computes XOR:
- The input has two dimensions so the input layer has two neurons.
- We are doing binary classification, so we can have one output neuron.

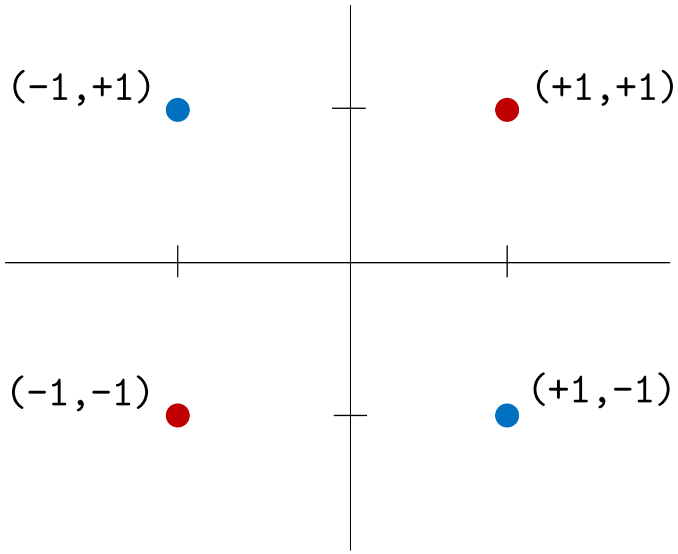




The activation function is:
$g(t) = \mathbb{1}\{t > 0\}$

EXPRESSIVENESS OF MLPS





EXPRESSIVENESS OF MLPS

- Using a more expressive model can enable it to learn more complex functions.
- This is the core idea underlying representation learning:
  - Instead of designing and specifying features manually and using a linear model,
  - Use a non-linear model.
  - If trained properly, the non-linear model will *learn* the features automatically.
  - I.e., the non-linear model will hopefully learn a good “representation” of the input data.
- For example, the XOR network in the last slide has learned two useful “detectors”:
  - The first hidden neuron fires when both inputs are +1.
  - The second hidden neuron fires when either input is +1.
- The representation is computed in the hidden layer.

EXPRESSIVENESS OF MLPS

- Universal approximation theorem: MLPs with one hidden layer and non-polynomial activation functions can approximate *any function*, with sufficiently many neurons in the hidden layer.
  - This theorem doesn't say how many neurons you need.
  - For some functions, you may need a lot of neurons.
- There are similar theorems that look at the arbitrary-depth case.
- But keep in mind expressiveness doesn't imply learnability.
  - Just because a machine learning model can express a function does not mean that it can easily learn it from data.

PROBABILISTIC PREDICTIONS

- Suppose we want an MLP to output a probability.
  - So there is only one output neuron.
  - How can we guarantee that the output neuron's value is between 0 and 1?
- We can set the activation function of the last neuron to be the logistic (sigmoid) function.
- But what if we have a multi-class classification task?
  - For each input, we want the model to output the probability distribution over output classes.
- E.g., consider the sentiment analysis task:
  - Given a user review of a product, the task is to classify whether the review is `positive`, `negative`, or `neutral`.
  - In this case, we want the model to output the probability that the review is `positive`, `negative`, or `neutral`.
  - The three predicted probabilities should sum to 1.

PROBABILISTIC PREDICTIONS

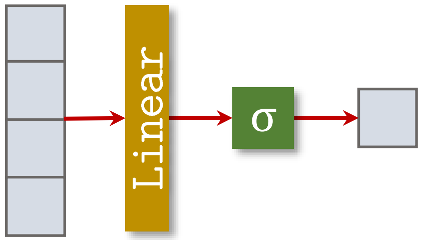
- Idea: Make the output layer have K neurons (where K is the number of output classes).
  - Compute the exponential of each output neuron,
  - And then normalize so they sum to 1.
- Suppose the activations of the last layer is x,
- The probability that the output y is k is:

$$p(y = k) = \frac{\exp(x_k)}{\sum_{j=1}^{K} \exp(x_j)}$$

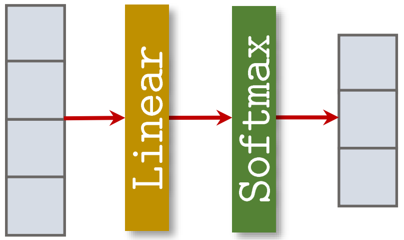
- Note that $f(\alpha) = \frac{\exp(\alpha_k)}{\sum_{j=1}^{K} \exp(\alpha_j)}$ is called the softmax function.
  - This is a multidimensional generalization of the logistic/sigmoid function.
- This is the same basic idea as multi-class logistic regression.

LOGISTIC REGRESSION AS MLP

- Neat connection: Suppose we have an MLP with an input layer, an output layer with 1 neuron, and zero hidden layers.
  - Let the activation function be the sigmoid function.
  - The resulting function is equivalent to logistic regression!



- If we increase the size of the output layer and replace the activation function with a softmax operation,
- The resulting network is equivalent to multi-class logistic regression.



CROSS-ENTROPY LOSS

- How do we train multi-class probabilistic models?
  - You could maximize the log likelihood.
- We can also minimize the cross-entropy loss:

$$L(w) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} p(y_i=k) \log f_w(x_i)_k$$

- If we know the ground truth labels y_i, then $p(y_i=k) = 1$ if the ground truth label for the i-th example is k, and 0 otherwise.

$$L(w) = -\frac{1}{N}\sum_{i=1}^{N} \log f_w(x_i)_{y_i}$$

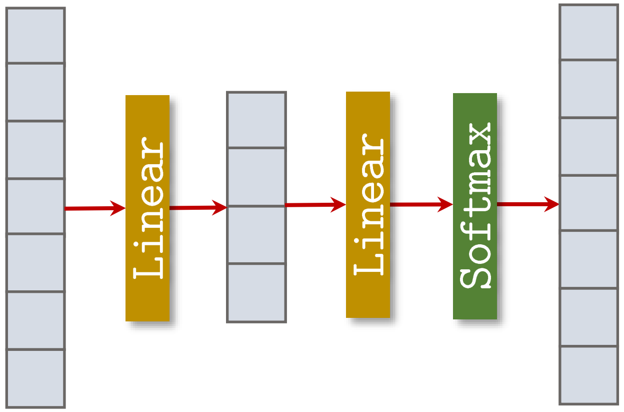
- Note that $f_w(x_i)_{y_i}$ is the model's prediction of the probability that the i-th example has label y_i.
  - This is the likelihood of the i-th example.
  - When learning, we are minimizing the loss, which is equivalent to the negative log likelihood.
  - Therefore, minimizing the cross-entropy loss is equivalent to maximizing the likelihood.

LEARNING REPRESENTATIONS OF WORDS

- Let's apply the ideas of representation learning that we discussed to learn good representations of words.
- Let's train a model to predict a word using information about the surrounding words.
- For example, pick some "window size", say 2.
- Given an input '`the cat sat on the mat`', and a target word '`sat`'.
  - We want the model to predict '`sat`' given the 2 preceding and 2 succeeding words: '`the cat ___ on the`'
- Use a bag-of-words embedding of the surrounding words.
- E.g., suppose our vocabulary consists of {`the, cat, sat, on, mat`}.
  - The bag-of-words embedding of the above input is: `[2, 1, 0, 1, 0]`

LEARNING REPRESENTATIONS OF WORDS

- Let's use the following neural network architecture:



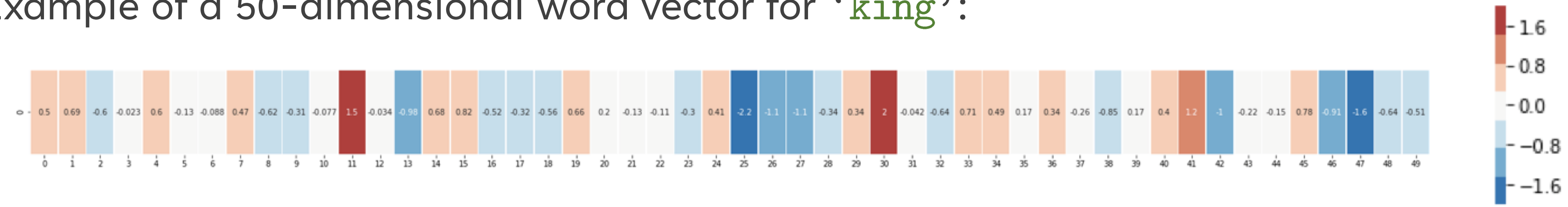
- The input is the bag-of-words encoding of the neighboring words.
- The output is the probability of the "masked" word (in the middle of the window).
- There is one hidden layer that is smaller than the input and output layers.
- This model is called the continuous bag-of-words (CBoW) model.

LEARNING REPRESENTATIONS OF WORDS

- In 2013, Mikolov et al. trained this model on data from Google News containing 1.6 billion words.
  - They used a window size of 4.
  - The vocab size is 1 million.
  - The hidden layer size (also called hidden dimension) is 600.
  - They minimized cross-entropy loss.
- After training the model, we can take any 1-hot embedding of a word and run the trained model on this input.
  - The 600-dimensional activations in the hidden layer provide a vector representation of that word.
- If we do this for every word, we obtain a mapping from every word to a vector.
  - This is called `word2vec`.
  - There are other ways to obtain vector representations of words (more generally called word vectors).

WORD VECTORS

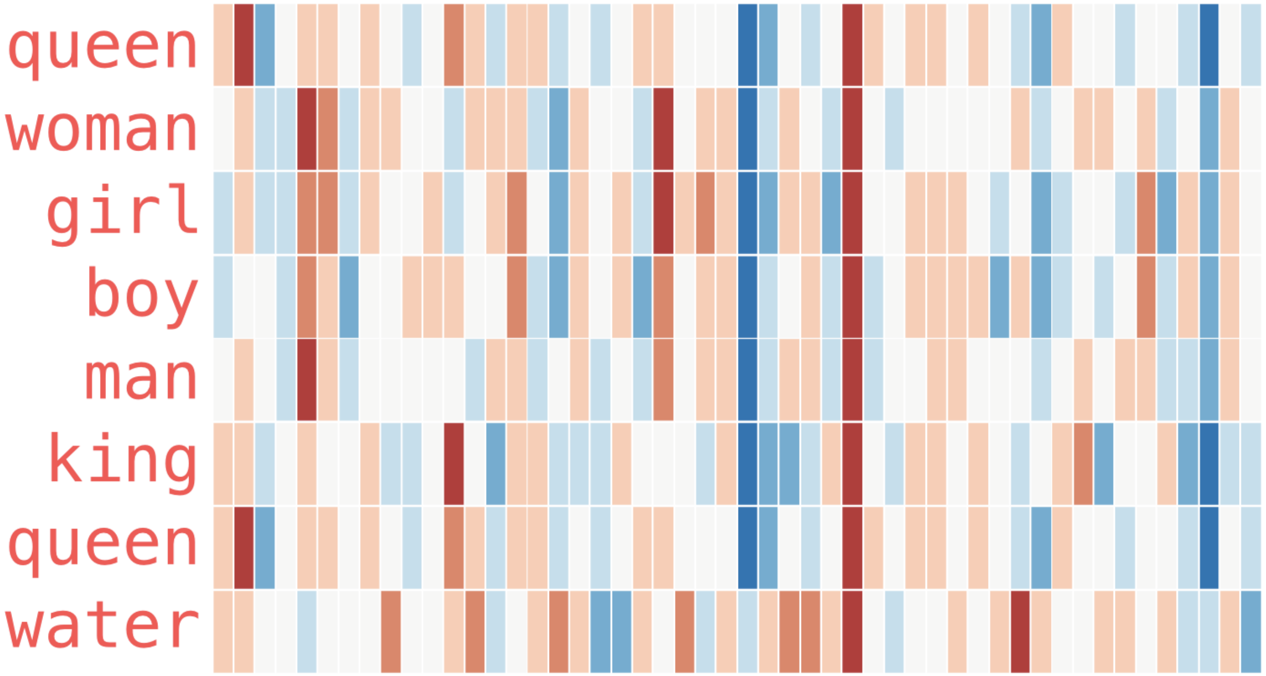
- How do we measure whether a learned representation is good?
  - I.e., How do we measure the quality of word vectors?
- What do word vectors look like?
- Example of a 50-dimensional word vector for ‘`king`’:



[Alammar, 2019, The Illustrated Word2vec]

WORD VECTORS

- How do we measure whether a learned representation is good?
  - I.e., How do we measure the quality of word vectors?
- What do word vectors look like?
- More examples:



[Alammar, 2019, The Illustrated Word2vec]

WORD VECTORS

- Examples of other learned relationships from `word2vec`:

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

- These are qualitative, not quantitative results, we have to interpret them with a grain of salt.
  - The authors may have cherry-picked them.
- There are some mistakes: '`France: tapas`', '`uranium: plutonium`', '`Google: Yahoo`'

[Mikolov et al. 2019]

WORD VECTORS

- Mikolov et al. (2019) did provide some quantitative results as well.

Dimensionality / Training words	24M	49M	98M	196M	391M	783M
50	13.4	15.7	18.6	19.1	22.5	23.2
100	19.4	23.1	27.8	28.7	33.4	32.2
300	23.2	29.2	35.3	38.6	43.7	45.9
600	24.0	30.1	36.5	40.8	46.6	50.4

- The table shows accuracies.
- The rows correspond to different hidden layer sizes.
- The columns correspond to different training set sizes.
- Evidently, more data and larger models do better.
- But the largest model is far from perfect.
- Word vectors can still be useful! (e.g., as embeddings in a larger model)

QUESTIONS?