CS 457/557: Machine Learning

Lecture 07-*:
Neural Networks

UNIVERSITY of WISCONSIN
LA CROSSE
Fall 2024

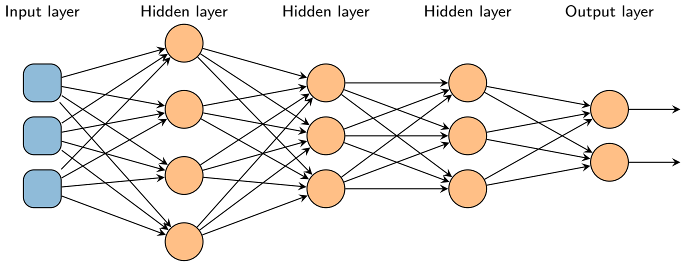
Prof. Jason Sauppe
jsauppe@uwlax.edu
https://cs.uwlax.edu/~jsauppe/

Lecture 07-1a: Neural Networks

Neural Learning Methods

Definition

A **neural network** is a computational model inspired by the structure of the human brain which consists of a network of simple information processing units called **neurons** (or **units**).

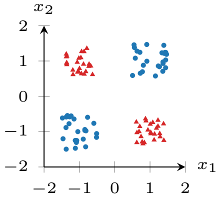


Can be used for classification and regression, plus other uses as well!

History of Neural Networks

A brief history:

- 1943: Early work by McCulloch and Pitts on modeling of neurons and network of connections that allow animals to learn
- 1957: Rosenblatt's perceptron algorithm was a single-layer neural network using the threshold activation function
- 1969: Minksy and Papert's book "Perceptrons" pointed out limitations (e.g., XOR) and led to the "AI Winter"



- Data is not linearly separable, so the perceptron algorithm cannot handle this!
- This problem can be solved using **multiple neurons**, but at the time it wasn't known how such networks could be trained

History of Neural Networks (continued)

1980s: The Connectionism Era: Connectionism is the idea that intelligent behavior can emerge from interactions between simple units.

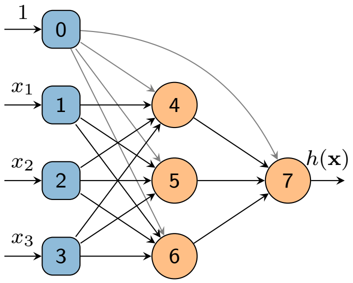
- 1986: The Parallel Distributed Processing (PDP) research group popularized the backpropagation algorithm for training a multi-layer neural network with logistic activation functions
- 1991: Vanishing gradient problem noted for deeper networks
- 1990s: Support vector machines gained in popularity due to ease of training compared to neural networks, leading to decreased interest in neural networks

2000s: Deep Learning Era

- Hinton et al. introduce greedy layer-wise pre-training process
- Use of rectified linear activation function (ReLU)
- Virtuous cycle: Better algorithms, faster hardware, bigger data

Lecture 07-1b: Anatomy of a Neuron

Network Topology and Notation



A **network** $\mathcal{N} = (V, A)$ consists of:

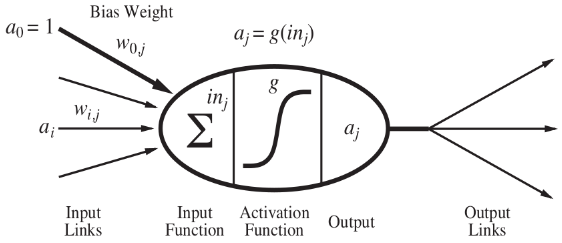
- a set of **vertices** (**nodes**) V, and
- a set of **directed edges** (**arcs**) $A \subseteq V \times V$

Neurons are labeled from 0 to n.

(Sometimes neurons are also indexed by layer, typically using a superscript, but we'll avoid this here to keep notation simple.)

- **Input neurons** are in blue, **processing neurons** are in yellow.
- The number of input neurons matches the dimensionality of a feature vector $\mathbf{x}$, plus the dummy neuron 0 (e.g., here $\mathbf{x} \in \mathbb{R}^3$)
- Neuron j produces an output a_j
- Input neuron j receives input x_j and transmits this as its output a_j (input neuron 0 receives and transmits $x_0 = 1$)
- Processing neuron j receives input from its incoming edges and produces some function of this input as its output a_j, which it transmits on its outgoing edges
- An edge (i, j) indicates a connection from neuron i to neuron j
- Input neuron 0 connects to all processing neurons (usually these edges are omitted)

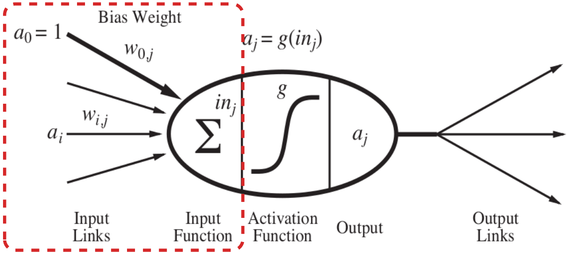
The Basic Neuron Model



For a processing neuron j:

- Input comes in from other neurons i with $(i, j) \in A$
- Some internal processing is done using the inputs
- Output transmitted to other neurons or used as network output

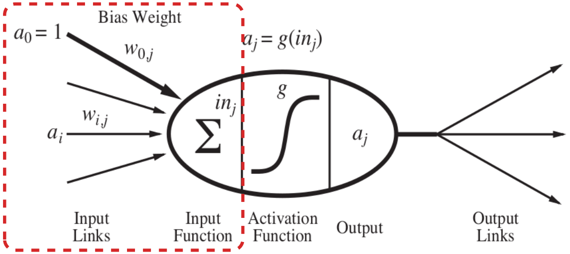
Analyzing A Neuron's Inputs



- Each input a_i from neuron i to neuron j weighted by w_{ij}
- Input $a_0 = 1$ from dummy neuron 0 is weighted by w_{0j}
- The input function is then a weighted linear sum:

$$in_j = \sum_{i:(i,j)\in A} w_{ij} a_i$$

Linear Transformations of Inputs

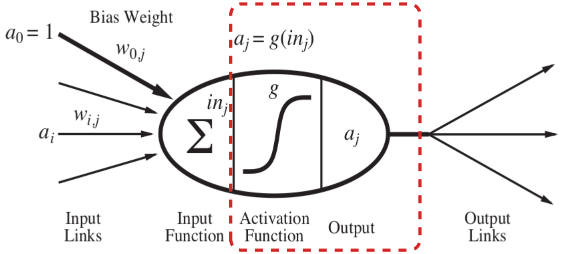


The weighted linear sum of inputs combined with $a_0 = 1$ is just like what gets done in other classifiers such as the perceptron!

$$in_j = \sum_{i:(i,j)\in A} w_{ij} a_i = \sum_{i=0}^{n} w_{ij} a_i = \mathbf{w}_j \cdot \mathbf{a}$$

where this last part abuses notation slightly using $w_{ij} = 0$ for $(i,j) \notin A$.

Neuron Output Functions



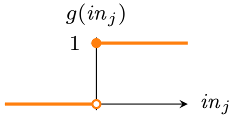
- After computing in_j, neuron j computes its output as $a_j = g(in_j)$, where g is an **activation function** that is generally **nonlinear**.
- The introduction of nonlinearity through g is crucial to allowing neural networks to **learn nonlinear functions**!

Different Activation Functions

Over the years, many different activation functions have been used:

The **binary threshold** function:

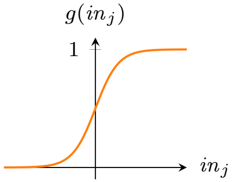
$$g(in_j) = \begin{cases} 1 & \text{if } in_j \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



- Provides a hard activation threshold
- Used by perceptron algorithm
- Not differentiable at 0; derivatives elsewhere are 0

The **logistic function**:

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



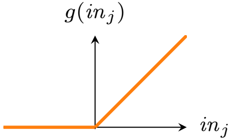
- Provides a soft activation threshold
- Differentiable everywhere, with $g'(in_j) = g(in_j)(1 - g(in_j))$
- Limits output to $[0, 1]$

Different Activation Functions (continued)

Over the years, many different activation functions have been used:

The **ReLU** function:

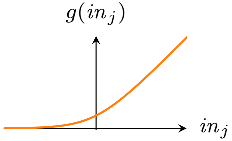
$$\begin{aligned} g(in_j) &= \mathsf{ReLU}(in_j) \\ &= \max\{0, in_j\} \end{aligned}$$



- ReLU is **rectified linear unit**
- Also known as a “ramp” function
- Popular in many modern applications
- Not differentiable at 0; derivative of 1 for positives, 0 for negatives

The **softplus** function:

$$\begin{aligned} g(in_j) &= \mathsf{softplus}(in_j) \\ &= \log\left(1 + e^{in_j}\right) \end{aligned}$$



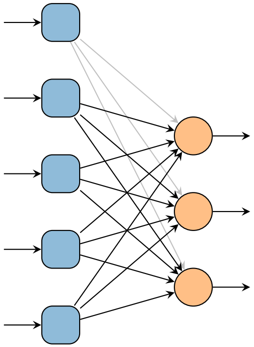
- Smoothed version of ReLU
- Differentiable everywhere, with $g'(in_j) = \sigma(in_j)$

Lecture 07-1c:
Single-Layer Perceptron Networks

Single-Layer Perceptron Networks

Definition

A **single-layer perceptron network** consists of an input layer and an output layer, with no hidden layers.

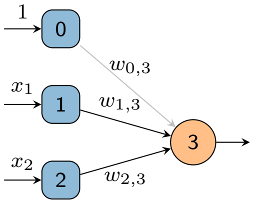


- Input neurons are connected directly to output neurons
- Each output neuron provides the probability of belonging to a particular class (e.g., for **multi-class classification**)
- Rosenblatt's perceptron algorithm is a special case with one output node using the binary threshold activation function

A Simple Example

Consider a network with a single processing neuron in the output layer.

With a logistic activation function, we have:



$$\begin{aligned} in_3 &= w_{0,3}a_0 + w_{1,3}a_1 + w_{2,3}a_2 \\ &= w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2 \end{aligned}$$

$$\begin{aligned} a_3 &= g(in_3) = \sigma(in_3) \\ &= \frac{1}{1 + e^{-(w_{0,3}+w_{1,3}x_1+w_{2,3}x_2)}} \end{aligned}$$

This is **logistic regression!**

Training is the process of finding optimal weights $\mathbf{w}^*$. This requires:

- Loss function
- Cost function
- Fitting method (optimization algorithm)

Some Loss Functions

Some loss functions that we have seen so far (treating $y_i \in \{0,1\}$ for classification):

$$\ell_2\left(y, h_{\mathbf{w}}(\mathbf{x})\right) = \left(y - h_{\mathbf{w}}(\mathbf{x})\right)^2$$

$$\ell_{0/1}\left(y, h_{\mathbf{w}}(\mathbf{x})\right) = \begin{cases} 0 & \text{if } y = h(\mathbf{x}) \\ 1 & \text{otherwise} \end{cases}$$

$$\ell_{\pi}\left(y, h_{\mathbf{w}}(\mathbf{x})\right) = \max\{0, (1-2y)\mathbf{w}\cdot\mathbf{x}\}$$

$$\ell_{\log}\left(y, h_{\mathbf{w}}(\mathbf{x})\right) = -1\left[y \log\left(h_{\mathbf{w}}(\mathbf{x})\right) + (1-y)\log\left(1 - h_{\mathbf{w}}(\mathbf{x})\right)\right]$$

To keep things simple, we'll use ℓ_2 loss for now for both regression and classification.

- Recall: ℓ_2 loss leads to a **non-convex** cost function for logistic regression
- However, for networks with more than one layer, most cost functions are non-convex regardless of the loss function that is used

The Cost Function and Fitting Process

We'll use the cost function $J(\mathbf{w})$ as the average loss over the training set:

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell_2\left(y_i, h_{\mathbf{w}}(\mathbf{x}_i)\right).$$

For fitting, we'll use gradient descent, specifically **stochastic gradient descent** (SGD), which has appealing properties for neural network training. Recall:

- Gradient descent converges to a local minimum
- If cost function is **convex**, all local minima are global minima
- If cost function is **non-convex**, gradient descent may not find the global optimum

SGD can avoid getting stuck in "shallow" local minima during search.

SGD Weight Updates

With stochastic gradient descent, each weight update examines a **single example** in the training set. For convenience, we'll denote this training example as $(\mathbf{x}, y)$ (no subscripts), and we'll treat the cost function $J(\mathbf{w})$ as only using this training example, i.e.:

$$J(\mathbf{w}) = \ell_2\left(y, h_{\mathbf{w}}(\mathbf{x})\right).$$

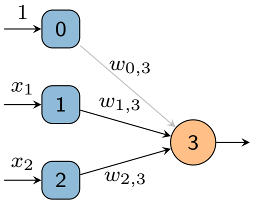
In SGD, the weight updates require the partial derivatives of $J(\mathbf{w})$ with respect to each weight w_{ij}:

$$w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \frac{\partial}{\partial w_{ij}} J(\mathbf{w}).$$

To compute $\frac{\partial}{\partial w_{ij}} J(\mathbf{w})$, we'll make use of the **chain rule** from calculus:

$$\frac{\partial}{\partial x} g(f(x)) = g'(f(x)) \frac{\partial}{\partial x} f(x) = g'(f(x)) f'(x).$$

Computing the Partial Derivatives



For the network on the left, we have:

$$\begin{aligned} J(\mathbf{w}) &= \ell_2\left(y, h_{\mathbf{w}}(\mathbf{x})\right) \\ &= \left(y - h_{\mathbf{w}}(\mathbf{x})\right)^2 \\ &= \left(y - a_3\right)^2 \\ &= \left(y - g(in_3)\right)^2 \\ in_3 &= w_{0,3}a_0 + w_{1,3}a_1 + w_{2,3}a_2 \\ &= w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2 \end{aligned}$$

So:

$$\begin{aligned} \frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} \left(y - g(in_3)\right)^2 && \textit{[By above derivation]} \\ &= 2\left(y - g(in_3)\right) \frac{\partial}{\partial w_{ij}} \left(y - g(in_3)\right) && \textit{[By chain rule]} \\ &= -2\left(y - g(in_3)\right) \frac{\partial}{\partial w_{ij}} g(in_3) \\ &= -2\left(y - g(in_3)\right) g'(in_3) \frac{\partial}{\partial w_{ij}} in_3 && \textit{[By chain rule]} \\ &= -2\left(y - g(in_3)\right) g'(in_3) \frac{\partial}{\partial w_{ij}} \left(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2\right) && \textit{[Defn. of } in_3\textit{]} \end{aligned}$$

Computing the Partial Derivatives

From the previous slide, we had

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = -2\,(y - g(in_3))\, g'(in_3) \frac{\partial}{\partial w_{ij}} \left(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2\right),$$

so

$$\begin{aligned}
\frac{\partial}{\partial w_{0,3}} J(\mathbf{w}) &= -2(y - g(in_3))g'(in_3) \\
\frac{\partial}{\partial w_{1,3}} J(\mathbf{w}) &= -2(y - g(in_3))g'(in_3)x_1 \\
\frac{\partial}{\partial w_{2,3}} J(\mathbf{w}) &= -2(y - g(in_3))g'(in_3)x_2.
\end{aligned}$$

- $y - g(in_3)$ is the **error** in prediction given example $(\mathbf{x}, y)$
- $g'(in_3)$ is the derivative of the activation function evaluated at in_3; if g is the logistic function σ, then $g'(t) = g(t)(1 - g(t))$ for all $t \in \mathbb{R}$.

Putting It All Together

Using the logistic activation function, the gradient descent updates are:

$$w_{0,3}^{(t+1)} \leftarrow w_{0,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3))$$
$$w_{1,3}^{(t+1)} \leftarrow w_{1,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3))x_1$$
$$w_{2,3}^{(t+1)} \leftarrow w_{2,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3))x_2.$$

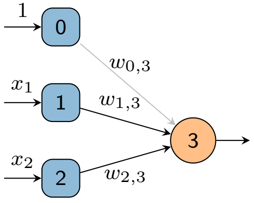
where $in_3 = w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2$.

This allows us to train a neural network with no hidden layers using SGD:

1. Select an example $(\mathbf{x}, y)$ from the training set
2. Compute in_3 and $g(in_3)$ using $\mathbf{x}$
3. Update weights as above
4. Repeat until convergence

Lecture 07-2a: Training a Simple Neural Network: Example SGD Iteration

Quick Recap: Computing Partial Derivatives



Gradient descent updates the weights using:

$$w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \frac{\partial}{\partial w_{ij}} J(\mathbf{w})$$

To make this work, we need the **partial derivatives** of the cost function $J(\mathbf{w})$!

We saw that:

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = -2(y - g(in_3))g'(in_3) \frac{\partial}{\partial w_{ij}} (w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2)$$

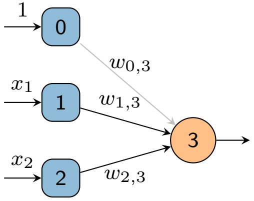
So for the particular weights in our network, we have:

$$\frac{\partial}{\partial w_{0,3}} J(\mathbf{w}) = -2(y - g(in_3))g'(in_3)$$

$$\frac{\partial}{\partial w_{1,3}} J(\mathbf{w}) = -2(y - g(in_3))g'(in_3)x_1$$

$$\frac{\partial}{\partial w_{2,3}} J(\mathbf{w}) = -2(y - g(in_3))g'(in_3)x_2$$

Example SGD Iteration



Suppose at iteration t we have:

$$w_{0,3}^{(t)} = 0.1$$
$$w_{1,3}^{(t)} = 0.1$$
$$w_{2,3}^{(t)} = 0.1$$

1. We select example $(\mathbf{x}, y)$ for training with $\mathbf{x} = (0.4, 0.5)$ and $y = 1$.
2. We compute:

$$\begin{aligned} in_3 &= w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2 \\ &= 0.1 + (0.1)(0.4) + (0.1)(0.5) \\ &= 0.19 \end{aligned}$$

$$\begin{aligned} g(in_3) &= \frac{1}{1 + e^{-0.19}} \\ &\approx 0.5474 \end{aligned}$$

Updating the Weights

3. Update the weights:

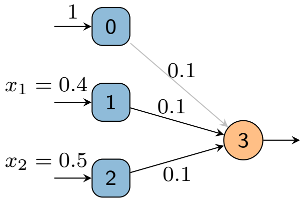
$$(y - g(in_3)) = (1 - 0.5474) = 0.4526$$
$$g'(in_3) = g(in_3)(1 - g(in_3)) = 0.5474(1 - 0.5474) \approx 0.2478$$

Using $\alpha = 0.5$ for simplicity:

$$\begin{aligned}
w_{0,3}^{(t+1)} &\leftarrow w_{0,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3)) \\
&= 0.1 + (0.4526)(0.2478) \\
&\approx 0.2122 \\
w_{1,3}^{(t+1)} &\leftarrow w_{1,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3))x_1 \\
&= 0.1 + (0.4526)(0.2478)(0.4) \\
&\approx 0.1449 \\
w_{2,3}^{(t+1)} &\leftarrow w_{2,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3))x_2 \\
&= 0.1 + (0.4526)(0.2478)(0.5) \\
&\approx 0.1561
\end{aligned}$$

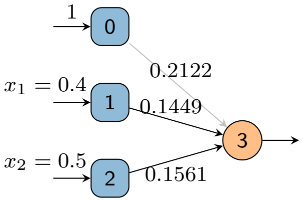
Evaluating the Change in Weights

At iteration t:



$$\begin{aligned} in_3 &= 0.1 + (0.1)(0.4) + (0.1)(0.5) \\ &= 0.19 \\ g(in_3) &\approx 0.5474 \end{aligned}$$

At iteration $t+1$:



$$\begin{aligned} in_3 &= 0.2122 + (0.1449)(0.4) + (0.1561)(0.5) \\ &= 0.34821 \\ g(in_3) &\approx 0.5862 \end{aligned}$$

After the weight update, the neuron connections have been **strengthened**:

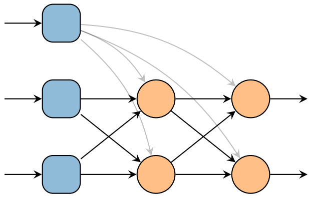
- Each weight increased in proportion to the error made as well as the input to that connection
- This causes the value computed by the output neuron to increase as well, reducing (but not eliminating) the error on this training example

Lecture 07-2b: Feedforward Neural Networks

Feedforward Neural Networks

Definition

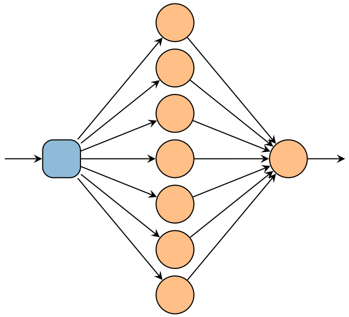
A **feedforward neural network** (**FFNN**, **multilayer perceptron**) consists of an input layer, an output layer, and one or more hidden layers. All node connections go from a layer to the subsequent layer.



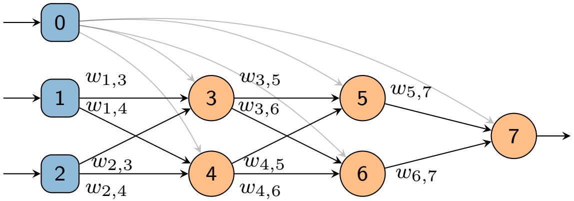
- Using hidden layers overcomes XOR problem of single-layer perceptrons
- The layout of the network is called the **architecture**; different architectures are suitable for different types of learning; fully-connected FFNN is a reasonable default
- Multilayer networks do feature engineering **automatically!**

Usefulness of Hidden Layers

The *Universal Approximation Theorem* states that under certain mild conditions, **any** continuous function can be approximated by a neural network with a **single hidden layer** and a large enough number of neurons.



The Effects of Hidden Layers



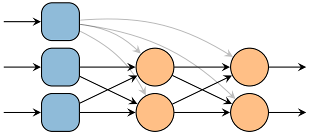
Output neurons process inputs **indirectly** via values that come from the hidden layers:

$$\begin{aligned}
a_7 &= g\left(w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6\right) \\
&= g\Big(w_{0,7} + w_{5,7}g\left(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4\right) + w_{6,7}g\left(w_{0,6} + w_{3,6}a_3 + w_{4,6}a_4\right)\Big) \\
&= g\Big(w_{0,7} + w_{5,7}g\big(w_{0,5} + w_{3,5}g(w_{0,3} + w_{1,3}a_1 + w_{2,3}a_2) \\
&\qquad\qquad\qquad + w_{4,5}g(w_{0,4} + w_{1,4}a_1 + w_{2,4}a_2)\big) \\
&\qquad + w_{6,7}g\big(w_{0,6} + w_{3,6}g(w_{0,3} + w_{1,3}a_1 + w_{2,3}a_2) \\
&\qquad\qquad\qquad + w_{4,6}g(w_{0,4} + w_{1,4}a_1 + w_{2,4}a_2)\big)\Big)
\end{aligned}$$

Lecture 07-2c:
Forward Propagation

Forward Propagation

To make predictions for an example $\mathbf{x}$ using a feedforward neural network with weights $\mathbf{w}$, we “push” the attribute values in $\mathbf{x}$ through the network!

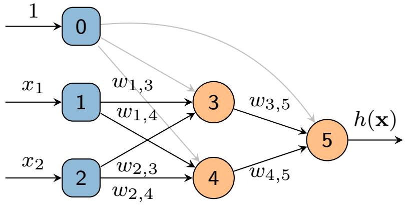


The Forward Propagation Algorithm

```
procedure ForwardPropagate($\mathbf{x}$; $\mathcal{N} = (V, A)$; $\mathbf{w}$)
    for each neuron $j$ in input layer do
        $a_j \leftarrow x_j$
    for each layer $\ell$ from $2$ to $L$ do
        for each neuron $j$ in layer $\ell$ do
            $in_j \leftarrow \sum_{i:(i,j)\in A} w_{ij} a_i$
            $a_j \leftarrow g(in_j)$
    return $a_j$ value(s) for every neuron $j$ in the output layer
```

Example: Making Predictions Using Forward Propagation

Let's make a prediction for the network below using input $\mathbf{x} = (3, 4)$, the weights given at the right, and the logistic activation function.



Parameter values:

$w_{0,3} = 5$	$w_{0,4} = 3$	$w_{0,5} = -1$
$w_{1,3} = 1$	$w_{1,4} = -4$	$w_{3,5} = 4$
$w_{2,3} = -2$	$w_{2,4} = 2$	$w_{4,5} = 2$

Input layer:

$$a_0 = 1 \qquad a_1 = x_1 = 3$$
$$a_2 = x_2 = 4$$

Hidden layer:

$$\begin{aligned} in_3 &= w_{0,3}a_0 + w_{1,3}a_1 + w_{2,3}a_2 \\ &= 5 \cdot 1 + 1 \cdot 3 - 2 \cdot 4 = 0 \\ a_3 &= g(in_3) = 0.5 \end{aligned}$$

$$\begin{aligned} in_4 &= w_{0,4}a_0 + w_{1,4}a_1 + w_{2,4}a_2 \\ &= 3 \cdot 1 - 4 \cdot 3 + 2 \cdot 4 = -1 \\ a_4 &= g(in_4) \approx 0.27 \end{aligned}$$

Output layer:

$$\begin{aligned} in_5 &= w_{0,5}a_0 + w_{3,5}a_3 + w_{4,5}a_4 \\ &= -1 \cdot 1 + 4 \cdot 0.5 + 2 \cdot 0.27 = 1.54 \\ a_5 &= g(in_5) \approx 0.82 \end{aligned}$$

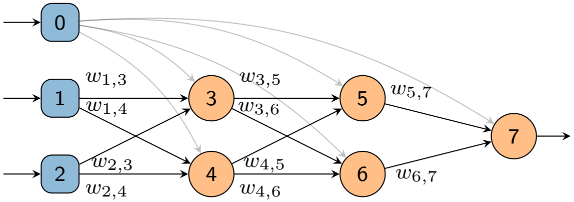
$$h(\mathbf{x}) = a_5 \approx 0.82$$

Compute node outputs in order, one at a time!

Lecture 07-2d:
Training a Feedforward Neural Network

Training a Feedforward Neural Network

Consider the following feedforward neural network:



To **train** it, we need to find optimal (or near-optimal) **weights** w^*_{ij} for each edge (i, j) in the network!

As before, this will require knowing the **partial derivatives** of the cost function $J(\mathbf{w})$.

Computing Partial Derivatives in a Multi-Layer Network

For the network on the previous slide, let's try to compute some partial derivatives of $J(\mathbf{w})$ given training example $(\mathbf{x}, y)$.

$$\begin{aligned}
\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} \left(y - a_7\right)^2 \\
&= \frac{\partial}{\partial w_{ij}} \left(y - g(in_7)\right)^2 \\
&= 2\left(y - g(in_7)\right) \frac{\partial}{\partial w_{ij}} \left(y - g(in_7)\right) \\
&= -2\left(y - g(in_7)\right) \frac{\partial}{\partial w_{ij}} g(in_7) \\
&= -2\left(y - g(in_7)\right) g'(in_7) \frac{\partial}{\partial w_{ij}} in_7 \\
&= -2\left(y - g(in_7)\right) g'(in_7) \frac{\partial}{\partial w_{ij}} \left(w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6\right)
\end{aligned}$$

Computing Partial Derivatives: Weights on Edges to Output Layer

On the previous slide, we saw that:

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = -2\left(y - g(in_7)\right) g'(in_7) \frac{\partial}{\partial w_{ij}} \left(w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6\right)$$

So for the weights on edges to neuron 7 in the output layer, we have:

$$\begin{aligned}
\frac{\partial}{\partial w_{0,7}} J(\mathbf{w}) &= -2\left(y - g(in_7)\right) g'(in_7) \\
\frac{\partial}{\partial w_{5,7}} J(\mathbf{w}) &= -2\left(y - g(in_7)\right) g'(in_7) a_5 \\
\frac{\partial}{\partial w_{6,7}} J(\mathbf{w}) &= -2\left(y - g(in_7)\right) g'(in_7) a_6.
\end{aligned}$$

This looks **similar** to what we had earlier with no hidden layers!

Computing Partial Derivatives: Weights on Edges to Last Hidden Layer

For weights on edges going into the last hidden layer, we have:

$$\begin{aligned}
\frac{\partial}{\partial w_{3,5}} J(\mathbf{w}) &= -2\,(y - g(in_7))\, g'(in_7)\ \frac{\partial}{\partial w_{3,5}}\,(w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6) && [\textit{Where is } w_{3,5}\,?] \\
&= -2\,(y - g(in_7))\, g'(in_7)\ \frac{\partial}{\partial w_{3,5}} w_{5,7}a_5 && [\textit{Buried in } a_5\,!] \\
&= -2\,(y - g(in_7))\, g'(in_7)\ w_{5,7}\ \frac{\partial}{\partial w_{3,5}} g(in_5) \\
&= -2\,(y - g(in_7))\, g'(in_7)\ w_{5,7}g'(in_5)\ \frac{\partial}{\partial w_{3,5}} in_5 \\
&= -2\,(y - g(in_7))\, g'(in_7)\ w_{5,7}g'(in_5)\ \frac{\partial}{\partial w_{3,5}}\,(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\
&= -2\,(y - g(in_7))\, g'(in_7)\ w_{5,7}g'(in_5)\ a_3
\end{aligned}$$

$$\frac{\partial}{\partial w_{w,6}} J(\mathbf{w}) = -2\,(y - g(in_7))\, g'(in_7)\ w_{6,7}g'(in_6)\ a_3$$

Computing Partial Derivatives: Weights on Edges to First Hidden Layer

For $w_{1,3}$, we have:

$$\begin{aligned}
\frac{\partial}{\partial w_{1,3}} J(\mathbf{w}) &= -2\,(y - g(in_7))\, g'(in_7)\ \frac{\partial}{\partial w_{1,3}} \left(w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6\right) \quad \textit{[Where is } w_{1,3}\textit{?]} \\
&= -2\,(y - g(in_7))\, g'(in_7)\ \frac{\partial}{\partial w_{1,3}} \left(w_{5,7}a_5 + w_{6,7}a_6\right) \quad \textit{[Buried in both } a_5 \textit{ and } a_6\textit{!]} \\
&= -2\,(y - g(in_7))\, g'(in_7) \left(w_{5,7} \frac{\partial}{\partial w_{1,3}} g(in_5) + w_{6,7} \frac{\partial}{\partial w_{1,3}} g(in_6)\right) \\
&= -2\,(y - g(in_7))\, g'(in_7) \left(w_{5,7} g'(in_5) \frac{\partial}{\partial w_{1,3}} in_5 + w_{6,7} g'(in_6) \frac{\partial}{\partial w_{1,3}} in_6\right) \\
&= \ \ldots
\end{aligned}$$

This takes **more work**, but **in principle** we can do this partial derivative calculation for any weight w_{ij}.

- We see some repeated terms showing up though – we can exploit this to make the partial derivative calculations easier!

Lecture 07-3a: Computing Arbitrary Partial Derivatives

The Chain Rule Revisited

Chain Rule

Suppose $z = g(f(x))$ for some functions f and g, and let $y = f(x)$. Then

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}.$$

Intuitively:

change in z per **change in** x = **change in** z per **change in** y
$\times$ **change in** y per **change in** x.

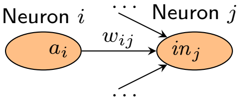
We can use this to **reformulate** our partial derivative calculations!

- We know that w_{ij} is a variable which influences $J(\mathbf{w})$.
- However, in_j is **another variable** that also influences $J(\mathbf{w})$, and in_j **depends on** w_{ij} because w_{ij} is used to compute in_j!

Reframing Partial Derivatives

For any $(i, j) \in A$, we have:

$$\begin{aligned}
\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \left[\frac{\partial}{\partial in_j} J(\mathbf{w})\right]\left[\frac{\partial}{\partial w_{ij}} in_j\right] \\
&= \left[\frac{\partial}{\partial in_j} J(\mathbf{w})\right]\left[\frac{\partial}{\partial w_{ij}}\left(\sum_{i':(i',j)\in A} w_{i'j} a_{i'}\right)\right] \\
&= \left[\frac{\partial}{\partial in_j} J(\mathbf{w})\right] a_i
\end{aligned}$$



- a_i: Output of neuron i
- in_j: Input to neuron j

Intuitively,

change in $J(\mathbf{w})$ per **change in** w_{ij} = **change in** $J(\mathbf{w})$ per **change in** in_j $\times$ **change in** in_j per **change in** w_{ij}.

Now we just need to calculate $\frac{\partial}{\partial in_j} J(\mathbf{w})$!

Computing Partial Derivatives With Respect To in_j: Base Case

To make notation easier, for any neuron $j \in V$, let

$$\Delta_j = \frac{\partial}{\partial in_j} J(\mathbf{w}).$$

If neuron j is in the **output layer** (the **base case**), then with squared error for loss we have:

$$\begin{aligned}
\Delta_j = \frac{\partial}{\partial in_j} J(\mathbf{w}) &= \frac{\partial}{\partial in_j} \left(y - a_j\right)^2 \\
&= 2\left(y - a_j\right) \frac{\partial}{\partial in_j} \left(y - g(in_j)\right) \\
&= -2\left(y - a_j\right) g'(in_j) \frac{\partial}{\partial in_j} in_j \\
&= -2\left(y - a_j\right) g'(in_j).
\end{aligned}$$

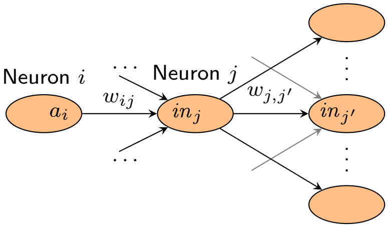
(This matches what we had seen earlier for the multi-layer network examples.)

Computing Partial Derivatives With Respect To in_j: Recursive Case

If neuron j is in a **hidden layer**, then we can use the chain rule to reformulate our computation for Δ_j as:

$$\Delta_j = \frac{\partial}{\partial in_j} J(\mathbf{w}) = \sum_{j' : (j,j') \in A} \left[\frac{\partial}{\partial in_{j'}} J(\mathbf{w})\right] \left[\frac{\partial}{\partial in_j} in_{j'}\right]$$

$$= \sum_{j' : (j,j') \in A} \Delta_{j'} \left[\frac{\partial}{\partial in_j} in_{j'}\right].$$



More Work

Now we need to compute $\frac{\partial}{\partial in_j} in_{j'}$:

$$\begin{aligned}
\frac{\partial}{\partial in_j} in_{j'} &= \frac{\partial}{\partial in_j} \sum_{i':(i',j')\in A} w_{i'j'} a_{i'} && \textit{[Defn. of } in_{j'}\textit{]} \\
&= \frac{\partial}{\partial in_j} \sum_{i':(i',j')\in A} w_{i'j'} g(in_{i'}) && \textit{[Defn. of } a_{i'}\textit{]} \\
&= \frac{\partial}{\partial in_j} w_{j,j'} g(in_j) \\
&= w_{j,j'} g'(in_j) \frac{\partial}{\partial in_j} in_j \\
&= w_{j,j'} g'(in_j)
\end{aligned}$$

More Work

From the previous slide, we found that

$$\frac{\partial}{\partial in_j} in_{j'} = w_{j,j'} g'(in_j).$$

We then use this to rewrite the calculation for Δ_j as

$$\begin{aligned}
\Delta_j &= \sum_{j':(j,j')\in A} \Delta_{j'} \left[\frac{\partial}{\partial in_j} in_{j'}\right] \\
&= \sum_{j':(j,j')\in A} \Delta_{j'} w_{j,j'} g'(in_j) \\
&= g'(in_j) \sum_{j':(j,j')\in A} w_{j,j'} \Delta_{j'}.
\end{aligned}$$

Putting It All Together

For any $(i, j) \in A$, we have:

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \left[\frac{\partial}{\partial in_j} J(\mathbf{w})\right] a_i = \Delta_j a_i.$$

We also saw that

$$\Delta_j = \begin{cases} -2(y - a_j)g'(in_j) & \text{if } j \text{ belongs to output layer} \\ g'(in_j) \displaystyle\sum_{j' : (j,j') \in A} w_{j,j'} \Delta_{j'} & \text{if } j \text{ belongs to hidden layer} \end{cases}$$

We can compute these Δ_j values in a "backwards" fashion, starting from the neurons in the output layer and moving towards the input layer.

- Using a different loss function will change the above expression for Δ_j for the output layer, but not for the hidden layers

Lecture 07-3b:
The Backpropagation Algorithm

Backpropagation Algorithm

The Backpropagation Algorithm

```
procedure BACKPROPAGATE($(\mathbf{x}, y)$; $\mathcal{N} = (V, A)$; $\mathbf{w}$)
    /* Forward Propagation */
    for each neuron $j$ in input layer do
        $a_j \leftarrow x_j$
    for each layer $\ell$ from $2$ to $L$ do
        for each neuron $j$ in layer $\ell$ do
            $in_j \leftarrow \sum_{i:(i,j)\in A} w_{ij} a_i$
            $a_j \leftarrow g(in_j)$
    /* Backpropagation */
    for each neuron $j$ in the output layer do
        $\Delta_j \leftarrow g'(in_j) \cdot (-2(y - a_j))$        ▷ Assumes squared error is the loss function
    for each layer $\ell$ from $L-1$ down to $2$ do
        for each neuron $j$ in layer $\ell$ do
            $\Delta_j \leftarrow g'(in_j) \sum_{j':(j,j')\in A} w_{j,j'} \Delta_{j'}$
```

- Forward propagation computes the in_j and a_j values for every neuron
- Backward propagation computes the Δ_j values for every neuron

Weight Updates

The **backpropagation algorithm** computes the in_j, a_j and Δ_j values for all neurons j in our network given a training example $(\mathbf{x}, y)$.

These values allow us to compute (estimate) the partial derivatives of J with respect to any weight w_{ij} using

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \left[\frac{\partial}{\partial in_j} J(\mathbf{w})\right] \left[\frac{\partial}{\partial w_{ij}} in_j\right] = \Delta_j a_i.$$

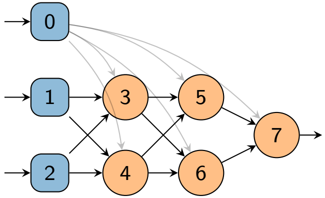
Then our **weight update** in stochastic gradient descent becomes

$$w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \left(\Delta_j a_i\right),$$

where α is the step size (learning rate).

Lecture 07-3c: Example Calculations

Some Calculations



$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \Delta_j a_i$$

$$\Delta_j = \begin{cases} -2(y - a_j) g'(in_j) & \text{if } j \text{ in output layer} \\ g'(in_j) \sum_{j' : (j,j') \in A} w_{j,j'} \Delta_{j'} & \text{if } j \text{ in hidden layer} \end{cases}$$

Computing some partial derivatives:

$$\frac{\partial}{\partial w_{5,7}} J(\mathbf{w}) = \Delta_7 a_5$$

$$= -2(y - a_7) g'(in_7)\ a_5$$

$$\frac{\partial}{\partial w_{3,5}} J(\mathbf{w}) = \Delta_5 a_3$$

$$= \Delta_7\ w_{5,7} g'(in_5)\ a_3$$

$$= -2(y - a_7) g'(in_7)\ w_{5,7} g'(in_5)\ a_3$$

Computing some Δ_j's:

$$\Delta_7 = -2(y - a_7) g'(in_7)$$

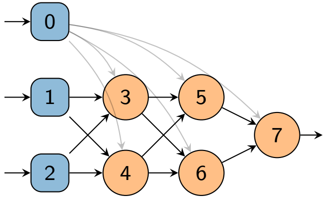
$$\Delta_5 = g'(in_5) \sum_{j' : (5,j') \in A} \Delta_{j'} w_{5,j'}$$

$$= \Delta_7\ w_{5,7} g'(in_5)$$

$$\Delta_6 = g'(in_6) \sum_{j' : (6,j') \in A} \Delta_{j'} w_{6,j'}$$

$$= \Delta_7\ w_{6,7} g'(in_6)$$

Some More Calculations



$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \Delta_j a_i$$

$$\Delta_j = \begin{cases} -2(y - a_j)g'(in_j) & \text{if } j \text{ in output layer} \\ g'(in_j) \displaystyle\sum_{j':(j,j')\in A} w_{j,j'}\Delta_{j'} & \text{if } j \text{ in hidden layer} \end{cases}$$

Computing some partial derivatives:

$$\frac{\partial}{\partial w_{1,3}} J(\mathbf{w}) = \Delta_3 a_1$$

Computing some Δ_j's:

$$\begin{aligned} \Delta_7 &= -2(y - a_7)g'(in_7) \\ \Delta_5 &= \Delta_7\ w_{5,7} g'(in_5) \\ \Delta_6 &= \Delta_7\ w_{6,7} g'(in_6) \\ \Delta_3 &= g'(in_3) \sum_{j':(3,j')\in A} w_{3,j'}\Delta_{j'} \\ &= g'(in_3)\left(w_{3,5}\Delta_5 + w_{3,6}\Delta_6\right) \\ &= g'(in_3)\big(w_{3,5}\Delta_7 w_{5,7} g'(in_5) \\ &\qquad + w_{3,6}\Delta_7 w_{6,7} g'(in_6)\big) \end{aligned}$$

Lecture 07-3d:
The Vanishing Gradient Problem

Quick Recap: Partial Derivatives and Backpropagation

We've seen how to compute the partial derivatives of our cost function $J(\mathbf{w})$ (involving the loss on a single training example) with respect to any weight w_{ij} in the network:

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \left[\frac{\partial}{\partial in_j} J(\mathbf{w})\right] a_i = \Delta_j a_i.$$

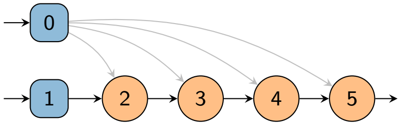
We also saw that

$$\Delta_j = \begin{cases} g'(in_j)(-2(y - a_j)) & \text{if } j \text{ belongs to output layer} \\ g'(in_j) \displaystyle\sum_{j' : (j,j') \in A} w_{j,j'} \Delta_{j'} & \text{if } j \text{ belongs to hidden layer} \end{cases}$$

The **backpropagation algorithm** provides an efficient way to calculate all a_i and Δ_j values. Let's take a closer look at these formulas.

Computing Δ_j Values

Consider the network below:



For this network, we have:

$$\Delta_j = \begin{cases} g'(in_j)(-2(y - a_j)) & \text{if } j = 5 \\ g'(in_j) w_{j,j+1} \Delta_{j+1} & \text{otherwise} \end{cases}$$

Then:

$$\begin{aligned} \Delta_2 &= \left[g'(in_2)w_{2,3}\right] \Delta_3 \\ &= \left[g'(in_2)w_{2,3}\right] \left[g'(in_3)w_{3,4}\right] \Delta_4 \\ &= \left[g'(in_2)w_{2,3}\right] \left[g'(in_3)w_{3,4}\right] \left[g'(in_4)w_{4,5}\right] \Delta_5 \\ &= \left[g'(in_2)w_{2,3}\right] \left[g'(in_3)w_{3,4}\right] \left[g'(in_4)w_{4,5}\right] \left[g'(in_5)(-2(y - a_5))\right] \end{aligned}$$

There are a lot of $g'(\cdot)$ evaluations here!

The Vanishing Gradient Problem

From previous slide, we have

$$\Delta_2 = \left[g'(in_2)w_{2,3}\right]\left[g'(in_3)w_{3,4}\right]\left[g'(in_4)w_{4,5}\right]\left[g'(in_5)(-2(y-a_5))\right].$$

With a logistic activation function, $g(t) = \sigma(t) = 1/\left(1+e^{-t}\right)$. Then:

- We have $g'(t) = g(t)(1-g(t))$.
- Recall: $g(t) \in [0,1]$ always. Therefore, $\max_{t\in\mathbb{R}} g'(t) = 0.25$.
- Repeated multiplication yields values **closer and closer to zero**
- As $\frac{\partial}{\partial w_{1,2}} J(\mathbf{w}) = \Delta_2 a_1$, the partial derivative will be very close to 0 $\Rightarrow$ we only make **very small changes** to $w_{1,2}$

With a ReLU activation function, $g(t) = \max\{0, t\}$. Then:

- We have $g'(t) = 1$ if $t > 0$, 0 if $t < 0$, and undefined if $t = 0$
- Might seem problematic if any $in_j < 0$ anywhere along the chain, but most networks aren't chains, and instead have Δ_j being a sum of $\Delta_{j'}$ values in the next layer

Lecture 07-4a: Stochastic Gradient Descent for Neural Networks

Quick Recap: Partial Derivatives and Backpropagation

For a cost function $J(\mathbf{w})$ involving squared error (loss) on a **single** training example, we have:

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \left[\frac{\partial}{\partial in_j} J(\mathbf{w})\right] \left[\frac{\partial}{\partial w_{ij}} in_j\right] = \Delta_j a_i$$

for any weight w_{ij}, where

$$\Delta_j = \begin{cases} g'(in_j)(-2(y - a_j)) & \text{if } j \text{ belongs to output layer} \\ g'(in_j) \displaystyle\sum_{j' : (j,j') \in A} w_{j,j'} \Delta_{j'} & \text{if } j \text{ belongs to hidden layer} \end{cases}$$

Forward propagation calculates the in_j and a_j values; **Backward propagation** provides an efficient way to calculate the Δ_j values.

Let's see how we can use this to **train** a neural network!

Neural Network Training

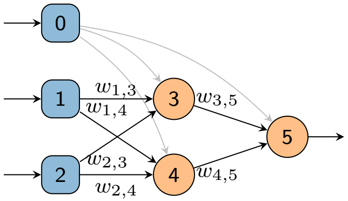
Training a Neural Network with Stochastic Gradient Descent

```
procedure NeuralNetTrain($\{(\mathbf{x}_i, y_i)\}_{i=1}^n$; $\mathcal{N} = (V, A)$; learning rate $\alpha$)
    for each edge $(i, j) \in A$ do
        $w_{ij}^{(0)} \leftarrow \text{RAND}(-\epsilon, \epsilon)$          ▷ Initialize each weight to a small random value
    $e \leftarrow 0$, $t \leftarrow 0$          ▷ Initialize epoch and iteration counters
    while stopping conditions not met do
        Create random permutation $L$ of $\{1, 2, \ldots, n\}$
        for each $i$ in $L$ do
            Run Backpropagate($(\mathbf{x}_i, y_i), \mathcal{N}, \mathbf{w}^{(t)}$) to obtain all $in_j$, $a_j$, $\Delta_j$ values
            for each edge $(i, j) \in A$ do
                $w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \Delta_j a_i$          ▷ Update each edge weight
            $t \leftarrow t + 1$
        $e \leftarrow e + 1$          ▷ an epoch ends once we process all training examples once
```

- Using the same value for all weights is problematic, so randomize!
- Typical stopping conditions are reaching an epoch limit, getting errors close to zero for all training examples, and/or having minimal change in cost across an epoch

The Need for Randomizing the Initial Weights

Suppose we tried to initialize each weight to 0 as we did in linear regression. What could go wrong?



If all of the weights are initialized to 0, then we have:

$$in_3 = w_{0,3}^{(0)} + w_{1,3}^{(0)}x_1 + w_{2,3}^{(0)}x_2 = 0$$
$$in_4 = w_{0,4}^{(0)} + w_{1,4}^{(0)}x_2 + w_{2,4}^{(0)}x_2 = 0$$

More generally, if $w_{0,3}^{(0)} = w_{0,4}^{(0)}$, $w_{1,3}^{(0)} = w_{1,4}^{(0)}$, and $w_{2,3}^{(0)} = w_{2,4}^{(0)}$, then neurons 3 and 4 see the **same** inputs and produce the **same** output.

Okay, but maybe this is just a temporary issue that would get resolved as the network trains...

Quick Aside: The Problem with Equal Weights

Suppose that at the start of iteration t, the weights satisfy:

$$w_{0,3}^{(t)} = w_{0,4}^{(t)} \qquad w_{1,3}^{(t)} = w_{1,4}^{(t)} \qquad w_{2,3}^{(t)} = w_{2,4}^{(t)} \qquad w_{3,5}^{(t)} = w_{4,5}^{(t)}$$

Forward propagation on $(\mathbf{x}, y)$ computes

$$in_3 = w_{0,3}^{(t)} + w_{1,3}^{(t)} x_1 + w_{2,3}^{(t)} x_2 \qquad in_4 = w_{0,4}^{(t)} + w_{1,4}^{(t)} x_1 + w_{2,4}^{(t)} x_2$$

so $in_3 = in_4$, which means that $a_3 = a_4$ as well. Further forward propagation and backward propagation computes

$$\begin{aligned} in_5 &= w_{0,5}^{(t)} + w_{3,5}^{(t)} a_3 + w_{4,5}^{(t)} a_4 \\ \Delta_5 &= g'(in_5)(-2(y - a_5)) \\ \Delta_3 &= g'(in_3) w_{3,5}^{(t)} \Delta_5 \\ \Delta_4 &= g'(in_4) w_{4,5}^{(t)} \Delta_5 \end{aligned}$$

Because $in_3 = in_4$ and $w_{3,5}^{(t)} = w_{4,5}^{(t)}$, we have $\Delta_3 = \Delta_4$.

Quick Aside: The Problem with Equal Weights (continued)

For $w_{0,3}$ and $w_{0,4}$, we have:

$$w_{0,3}^{(t+1)} \leftarrow w_{0,3}^{(t)} - \Delta_3 a_0$$
$$w_{0,4}^{(t+1)} \leftarrow w_{0,4}^{(t)} - \Delta_4 a_0$$

Because $w_{0,3}^{(t)} = w_{0,4}^{(t)}$ and $\Delta_3 = \Delta_4$, we have $w_{0,3}^{(t+1)} = w_{0,4}^{(t+1)}$.

For $w_{1,3}$ and $w_{1,4}$, we have:

$$w_{1,3}^{(t+1)} \leftarrow w_{1,3}^{(t)} - \Delta_3 a_1$$
$$w_{1,4}^{(t+1)} \leftarrow w_{1,4}^{(t)} - \Delta_4 a_1$$

Because $w_{1,3}^{(t)} = w_{1,4}^{(t)}$ and $\Delta_3 = \Delta_4$, we have $w_{1,3}^{(t+1)} = w_{1,4}^{(t+1)}$.

For $w_{2,3}$ and $w_{2,4}$, we have:

$$w_{2,3}^{(t+1)} \leftarrow w_{2,3}^{(t)} - \Delta_3 a_2$$
$$w_{2,4}^{(t+1)} \leftarrow w_{2,4}^{(t)} - \Delta_4 a_2$$

Because $w_{2,3}^{(t)} = w_{3,4}^{(t)}$ and $\Delta_3 = \Delta_4$, we have $w_{2,3}^{(t+1)} = w_{2,4}^{(t+1)}$.

For $w_{3,5}$ and $w_{4,5}$, we have:

$$w_{3,5}^{(t+1)} \leftarrow w_{3,5}^{(t)} - \Delta_5 a_3$$
$$w_{4,5}^{(t+1)} \leftarrow w_{4,5}^{(t)} - \Delta_5 a_4$$

Because $w_{3,5}^{(t)} = w_{4,5}^{(t)}$ and $a_3 = a_4$, we have $w_{3,5}^{(t+1)} = w_{4,5}^{(t+1)}$.

So at iteration $t + 1$, we still have the **same** equality relationships across the weights!

Keeping the Initial Weights Small

As we just saw, picking the same initial value for each weight is **bad**, so we should use a different initial value for each weight.

An easy way to do this is via **randomization**! We pick random values with a **small magnitude** because otherwise neurons tend to get **saturated**, meaning that their outputs are very close to 0 or 1 (when using a logistic activation function).

Recall that $in_j = \sum_{i:(i,j)\in A} w_{ij} a_i$:

- If $in_j \geq 10$, then $a_j = g(in_j) \geq \sigma(10) \approx 0.99995$
- If $in_j \leq -10$, then $a_j = g(in_j) \leq \sigma(-10) \approx 0.00005$

Additionally, $g'(in_j) = \sigma(in_j)(1 - \sigma(in_j)) \approx 0$, which makes training **slow**.

This is also why we should **normalize** the input attributes x_j prior to training, e.g., by rescaling values to the range $[-1, 1]$ via **min-max normalization**.

Notes About Stochastic Gradient Descent

When we have **many** training examples, our cost function $J(\mathbf{w})$ is typically the average loss:

$$J(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(y_i, h_{\mathbf{w}}(\mathbf{x}_i)\right).$$

In **stochastic gradient descent**, we **estimate** the gradient of $J(\mathbf{w})$ instead of computing it exactly.

- SGD weight updates are faster (relative to full batch GD)
- Not all steps we make are directions of descent, though!
  - This may seem **bad**, but it actually allows SGD to potentially **escape from shallow local minima**
  - Neural network cost functions are generally **non-convex** with lots of local minima
- SGD tends to have poor convergence properties though...

Lecture 07-4b: Mini-Batch Gradient Descent for Neural Networks

Mini-Batch Gradient Descent

Recall that **mini-batch gradient descent** offers a compromise between computing the gradient of $J(\mathbf{w})$ exactly versus estimating it using a single training point.

To use mini-batch gradient descent, we need to revisit our partial derivative calculations.

In SGD, we use a single training example $(\mathbf{x}, y)$ with:

$$J(\mathbf{w}) \approx \ell\left(y, h_{\mathbf{w}}(\mathbf{x})\right)$$

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} \ell\left(y, h_{\mathbf{w}}(\mathbf{x})\right) \\ &= \Delta_j a_i\end{aligned}$$

where Δ_j and a_i are computed by running backpropagation on training example $(\mathbf{x}, y)$.

In mini-batch GD, we use a subset of training examples with indices in a set B:

$$J(\mathbf{w}) \approx \frac{1}{|B|} \sum_{n \in B} \ell\left(y_n, h_{\mathbf{w}}(\mathbf{x}_n)\right)$$

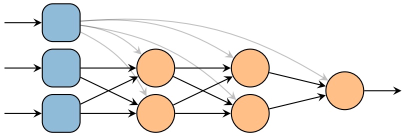
$$\begin{aligned}\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} \frac{1}{|B|} \sum_{e \in B} \ell\left(y_e, h_{\mathbf{w}}(\mathbf{x}_e)\right) \\ &= \frac{1}{|B|} \sum_{e \in B} \frac{\partial}{\partial w_{ij}} \ell\left(y_e, h_{\mathbf{w}}(\mathbf{x}_e)\right) \\ &= \frac{1}{|B|} \sum_{e \in B} \left(\Delta_j^{(e)} a_i^{(e)}\right)\end{aligned}$$

where $\Delta_j^{(e)}$ and $a_i^{(e)}$ are the Δ_j and a_i values computed during backpropagation using training example $(\mathbf{x}_e, y_e)$, for each example index $e \in B$. (So we have to run backpropagation $|B|$ times!)

Lecture 07-4c:
Regression and Multi-Class Classification

Neural Networks for Regression

So far we've been thinking about a neural network in terms of binary classification (just like **logistic regression**):



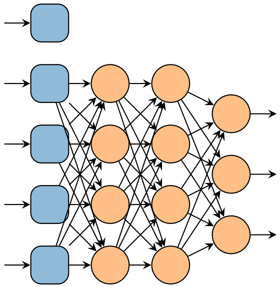
Given example $\mathbf{x}$, the predicted output $h(\mathbf{x})$ is the output value $a_j = g(in_j)$ from the one neuron j in the output layer.

- With $g = \sigma$, we have $a_j \in [0, 1]$, which is interpreted as the **probability** that $\mathbf{x}$ belongs to the positive class

What if we want to do **regression**? Just **remove** the activation function g from the neuron in the output layer, so that $a_j = in_j = \sum_{i:(i,j)\in A} w_{ij} a_i$ (and update the partial derivative calculations appropriately).

Multi-Class Classification

For **multi-class classification** with $K > 2$ classes, we typically use one output neuron per class.



For any class $k \in \{1, 2, \ldots, K\}$, the output neuron associated with k is responsible for:

- "Firing" (i.e., producing a value close to 1) if input $\mathbf{x}$ looks like it belongs to class k
- "Not firing" (i.e., producing a value close to 0) if input $\mathbf{x}$ looks like it does not belong to class k

We can also use this configuration for $K = 2$ classes (which can be more convenient). However, with this setup, we need to make a few modifications to our inputs and network outputs.

Multi-Class Classification: Network Outputs

With multiple output neurons, the hypothesis function $h_{\mathbf{w}}$ for a neural network produces **multiple** output values for an input $\mathbf{x} \in \mathbb{R}^p$.

We can think of these outputs as a **vector** of predictions $\hat{\mathbf{y}}$. For example, if $K = 3$, we might find

$$\hat{\mathbf{y}} = h_{\mathbf{w}}(\mathbf{x}) = \begin{pmatrix} 0.03 \\ 0.57 \\ 0.24 \end{pmatrix}.$$

The kth component of $\hat{\mathbf{y}}$ can **almost** be interpreted as the probability that $\mathbf{x}$ belongs to class k; the difficulty is that there is no guarantee that the components of $\hat{\mathbf{y}}$ add up to 1 (but this should happen for probabilities).

- If we **rescale** each component of $\hat{\mathbf{y}}$ by dividing it by the sum of the values in $\hat{\mathbf{y}}$, then the rescaled values can be interpreted as probabilities.
- For classification purposes, we can just predict the label that corresponds to the component of $\hat{\mathbf{y}}$ with the **highest value**.

Multi-Class Classification: Network Inputs

In a multi-class classification context, the output y is usually a **class label** in the range $\{1, 2, \ldots, K\}$.

To compare a single label y with a vector of predictions $\hat{\mathbf{y}}$, we apply **one-hot encoding** on the label y. For example, if $K = 3$, we would **one-hot encode** the three different labels as the following **vectors**:

$$y = 1 \;\Rightarrow\; \mathbf{y} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad y = 2 \;\Rightarrow\; \mathbf{y} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad y = 3 \;\Rightarrow\; \mathbf{y} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

The squared error loss function $\ell_2(\mathbf{y}, \hat{\mathbf{y}})$ is then a sum of squared errors across the classes:

$$\ell_2(\mathbf{y}, \hat{\mathbf{y}}) = \ell_2(\mathbf{y}, h_{\mathbf{w}}(\mathbf{x})) = \sum_{k=1}^{K} (y_k - a_k)^2 .$$

where a_k comes from the output neuron corresponding to class k.

Lecture 07-4d:
Regularization in Neural Networks

Generalization and Overfitting

Remember the **ultimate goal** of learning a hypothesis function: we want to **do well on out-of-sample data** (i.e., test data), not just on in-sample training data.

With linear regression, we saw that:

- Using a model that was **too simple** resulted in **underfitting**
- Using a model that was **too complex** resulted in **overfitting**

Having more parameters (i.e., weights) makes it easier to **overfit**.

A fully connected feed-forward neural network can have **lots** of parameters: with p features and p neurons in the first hidden layer, there are p^2 weights just between these first two layers alone!

Because of this, neural networks are **prone to overfitting** if there is insufficient training data available.

Corrections for Overfitting

Some remedies to overfitting:

- Get more training data!
  - Adjusting parameters to "memorize the noise" on a single training example is likely to increase error on other training examples, so the learning algorithm won't do this!
  - This is why "big data" has helped renew interest in neural networks.

- Add **regularization** to the cost function:

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell\left(y_i, h_{\mathbf{w}}(\mathbf{x}_i)\right) + \lambda \sum_{(i,j)\in A} w_{ij}^2$$

Recall: λ is a **model hyperparameter** that needs to be picked ahead of time (we can use **cross-validation** to help pick λ).

Regularization in Neural Networks: Weight Decay

In stochastic gradient descent with training example $(\mathbf{x}, y)$, the partial derivatives of the regularized cost function are given by:

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} \left[\ell\left(y, h_{\mathbf{w}}(\mathbf{x})\right) + \lambda \sum_{(i',j') \in A} w_{i'j'}^2 \right] \\ &= \left[\frac{\partial}{\partial w_{ij}} \ell\left(y, h_{\mathbf{w}}(\mathbf{x})\right) \right] + \left[\lambda \sum_{(i',j') \in A} \frac{\partial}{\partial w_{ij}} w_{i'j'}^2 \right] \\ &= \Delta_j a_i + 2\lambda w_{ij}.\end{aligned}$$

Then the weight update is:

$$w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \Delta_j a_i - 2\alpha\lambda w_{ij}.$$

This is called **weight decay**, because the weights tend to move **towards zero** unless pushed back by $\Delta_j a_i$.

Lecture 07-5a:
More on Regularization

Regularization in Neural Networks

With regularization, our cost function is

$$J(\mathbf{w}) = \sum_{i=1}^{n} L\left(y_i, h_{\mathbf{w}}(\mathbf{x}_i)\right) + \lambda \sum_{(i,j)\in A} w_{ij}^2.$$

Regularization tends to keep weights small. With small weights, in_j doesn't get too far from 0:

$$in_j = \sum_{i:(i,j)\in A} w_{ij} a_i$$

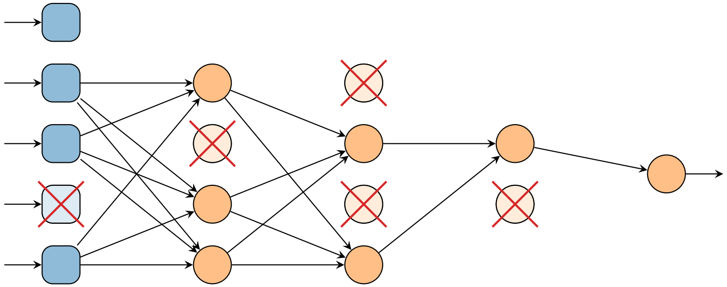
- With a logistic activation function, this helps ensure that $g(in_j)$ is not too close to either 0 or 1, and hence that $g'(in_j)$ isn't close to zero

So **weight decay** helps **mitigate** the **vanishing gradient problem**!

Dropout Regularization

Dropout regularization is another way to reduce overfitting:

- At the start of each SGD iteration, temporarily drop some nodes in the network (chosen at random)
- Run the backprop update process as before, but rescale the outputs from the surviving neurons in each layer to compensate for the loss of output from the dropped neurons



Benefits of Dropout Regularization

Dropout regularization prevents any neuron from relying too heavily on any one input connection (because it might disappear in the future), and thus makes it such that the weights get distributed across incoming connections.

Key benefits:

- Avoids overemphasis on single features
- Hidden units are forced to work well with others
- Dropout introduces noise which forces the network to be robust to it
- Approximates creating an ensemble of networks instead of just a single one (this idea is more general than this, and is **quite useful**!)

Lecture 07-5b:
Some Black Box Magic

Training Improvement: Batch Normalization

Batch normalization is a technique that can be used to improve the training of neural networks with mini-batch gradient descent. It involves **normalizing** the outputs from each layer of the network before feeding them into the next layer.

For mini-batch B, let $a_j^{(e)}$ be output of neuron j on example e. Then batch normalization updates $a_j^{(e)}$ as

$$\hat{a}_j^{(e)} = \gamma \frac{a_j^{(e)} - \mu}{\sqrt{\epsilon + \sigma^2}} + \beta$$

- μ and σ are the mean and standard deviation of the $a_j^{(e)}$ values within B
- ϵ is a small positive constant to prevent division by zero
- γ and β are parameters that need to be learned during training

Researchers don't fully understand **why** this tends to work well, though!

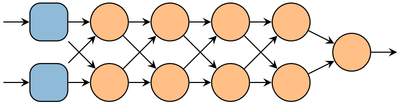
Lecture 07-5c:
Neural Net Architectures

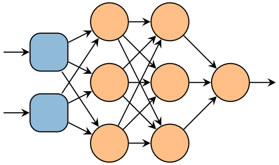
Designing a Network

Definition

The layout of a network is called the **network architecture**.

There are **lots** of different ways to structure a network!





- Both networks have 18 weights to learn (excluding the bias weights)
- For a fixed number of weights, **deep but narrow** tends to do better than **shallow but wide**

There is not yet a complete understanding for when and/or why some architectures work better than others.

In Search of Good Network Architecture

Figuring out the “right” architecture is kind of like **hyperparameter tuning**, where the number of layers, neurons per layer, and connections serve as hyperparameters.

Lots of performance improvements have come from experimentation with these hyperparamters (a process that is often called GSD for *graduate student descent*).

Definition

Neural architecture search is the process of automatically searching for a good network architecture.

Doing an **exhaustive search** through the hyperparameter values (e.g., via grid search) is infeasible because of the sheer number of different combinations of parameters that are available, so metaheuristics are often used instead.

Neural Architecture Search

Neural architecture search has been conducted using:

- Genetic algorithms and other metaheuristics
- Reinforcement learning approaches
- Gradient descent in network architecture space

Most options require evaluating any given architecture's performance, which generally requires training the architecture first! Training is slow, so various ways to speed it up have been proposed:

- Use smaller training data
- Stop training early and try to predict performance improvements based on trend in cost over epochs
- Use a reduced version of the architecture and hope its performance is comparable to a scaled up version
- Learn a heuristic evaluation function to map network architectures to performance estimates

Lecture 07-5d: Other Ideas

For Further Exploration

Network architectures have also been tailored for specific applications. Two prominent examples are:

- **Convolutional neural networks** (CNNs) are designed for image data, which is very high-dimensional (each pixel is a feature)
- **Recurrent neural networks** (RNNs) have back edges and feedback loops in the network, which make them suitable for natural language processing and similar tasks

For additional reading:

- *Deep Learning* by Goodfellow et al.
- *Dive into Deep Learning* by Zhang et al.
- Deep Learning Specialization course on Coursera

This looks fun too: `https://playground.tensorflow.org/`