CPE 490 590: Machine Learning for Engineering Applications

09 Neural Network

Rahul Bhadani
Electrical & Computer Engineering, The University of Alabama in Huntsville

Outline

Announcement

Quiz

- Next quiz will open on Friday 21, 2025.
- Around 1 hour.
- Will cover everything until Logistics Regression.
- May ask you to run notebooks

Motivation

Lecture Overview

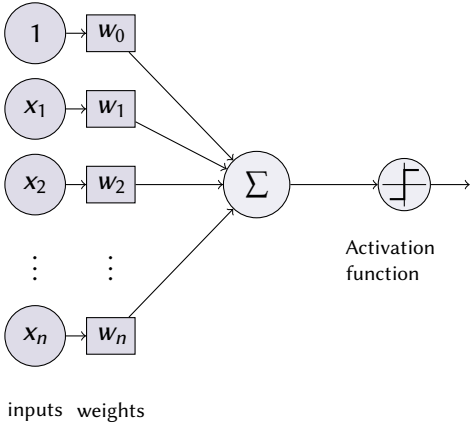
- Our goal is to develop an understanding of how neural networks work and how they can be trained.
- Neural networks are universal approximators. The universal approximation theorem implies that neural networks can represent a wide variety of functions when given appropriate weights. They typically do not provide a construction for the weights, but merely state that such a construction is possible.

A Brief History of Neural Networks

A Brief History of Neural Networks

Perceptron

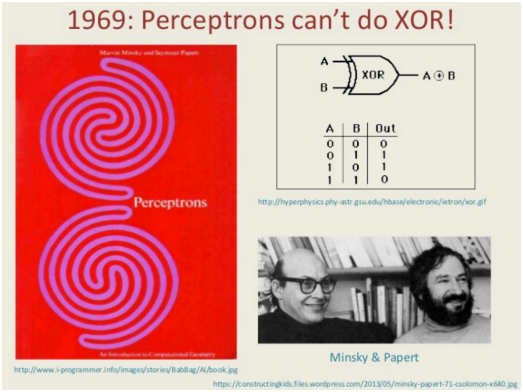
- The percepton is the first single layer network that has an "artificial" neuron, which was developed by Frank Rozenblat (1957).
- Designed for binary classification tasks.
- Optimization does not converge for nonlinearly separable datasets.



A Brief History of Neural Networks

Minsky & Papert (1969)

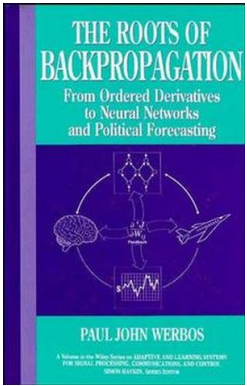
- Minsky & Papert published a work that was the "kiss of death" for neural network research.
- Claim: Perceptrons cannot solve the XOR problem (arguably the easiest nonlinear task)!
- Neural network research came to a halt for nearly 15 years.



A Brief History of Neural Networks

Backpropagation

- Werbos (1974), Rummelhart and Hinton (1986) were major contributors to the backpropagation algorithm, which is how we train neural networks.
- Werbos' 1974 PhD thesis from Harvard University presented the backpropagation algorithm. It wasn't until the 1980's that neural network research picked back up.
- Backprop even goes back to Kelley's work in 1960 on continuous-time control theory.



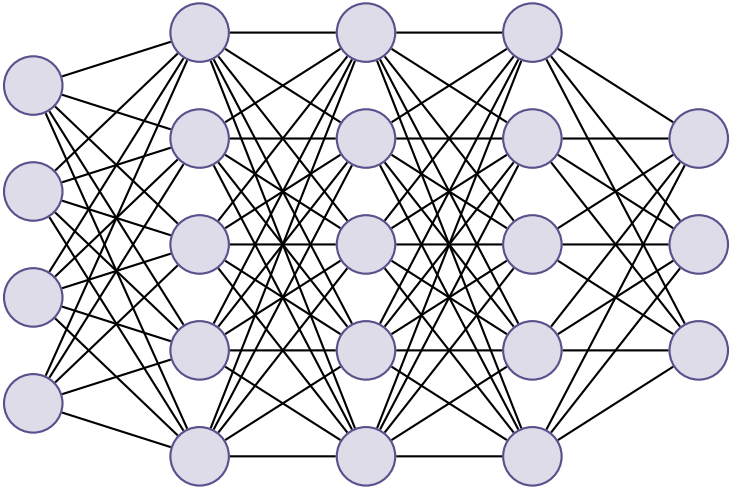
The Multilayer Perceptron and Backpropagation

Definitions

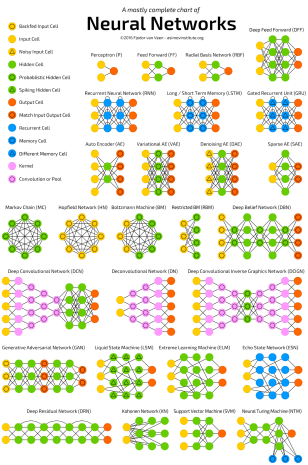
From Microsoft Encarta

A neural network is a highly interconnected network of information-processing elements that mimics the connectivity and functioning of the human brain. One of the most significant strengths of neural networks is their ability to learn from a limited set of examples.

A Neural Network



A Tour of Neural Networks



Definitions

- We assume a supervised learning setting where we're provided a training data set $\mathcal{D} := \{\mathbf{x}(n), \mathbf{d}(n)\}_{n=1}^{N}$, where $\mathbf{x}(n)$ is a feature vector and $\mathbf{d}(n)$ is the desired output for the nth sample.
  - Notation such as $d_j(n)$ is used to refer to the jth output node of the desired signal
  - Neuron and node are used interchangeably
- The first and last layers of the network are referred to as the input and output layer, respectively. All layers between the input and output are referred to as *hidden layers*.
- For the moment, we're only going to exam a neural network with one hidden layer; however, we can generalize the results to multiple hidden layers.
  - Training a very deep network can be challenging due to issues we cover in this lecture and future lectures will address training a deep network.

An Example

The Credit Assignment Problem

- The credit assignment problem is one where we seek to credit or blame for overall outcomes to each of the internal decisions made by the hidden computational units.
  - These hidden units are the first to "blame" for an error made at the output because the data are processed internally before ever reaching the output.
  - How much "weight" or "credit" do we assign a neuron when an error is made at the output?
- The *backpropagation algorithm* solves the credit assignment problem in an elegant manner.

Setting up the Problem

Defining the error

A data $\mathbf{x}(n)$ is passed through the network and the measured output of the neural network is $\mathbf{y}(n)$.

Defining the error

A data $\mathbf{x}(n)$ is passed through the network and the measured output of the neural network is $\mathbf{y}(n)$. The error signal, $e_j(n)$

$$e_j(n) = d_j(n) - y_j(n),$$

Defining the error

A data $\mathbf{x}(n)$ is passed through the network and the measured output of the neural network is $\mathbf{y}(n)$. The error signal, $e_j(n)$, instantaneous error energy, $E_j(n)$

$$e_j(n) = d_j(n) - y_j(n), \quad E_j(n) = \frac{1}{2} e_j^2(n),$$

Defining the error

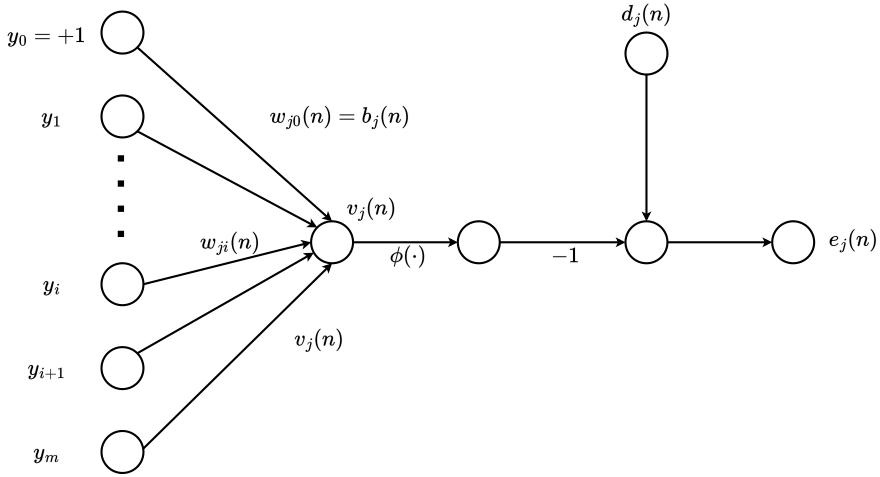
A data $\mathbf{x}(n)$ is passed through the network and the measured output of the neural network is $\mathbf{y}(n)$. The error signal, $e_j(n)$, instantaneous error energy, $E_j(n)$, and total error, $E(n)$, is given by:

$$e_j(n) = d_j(n) - y_j(n), \quad E_j(n) = \frac{1}{2}e_j^2(n), \quad E(n) = \sum_{j=1}^{C} E_j(n) = \frac{1}{2}\sum_{j=1}^{C} e_j^2(n)$$

The Total Error Overall Samples

$$E_{av}(N) = \frac{1}{N}\sum_{n=1}^{N} E(n) = \frac{1}{2N}\sum_{n=1}^{N}\sum_{j=1}^{C} e_j^2(n)$$

Signal Flow at a Neuron



Signal Flow at a Neuron

The *induced local field*, $v_j(n)$ is produced at the input of the activation associated with neuron j, which is given by:

$$v_j(n) = \sum_{i=1}^{m} w_{ji}(n) y_i(n)$$

Signal Flow at a Neuron

The *induced local field*, $v_j(n)$ is produced at the input of the activation associated with neuron j, which is given by:

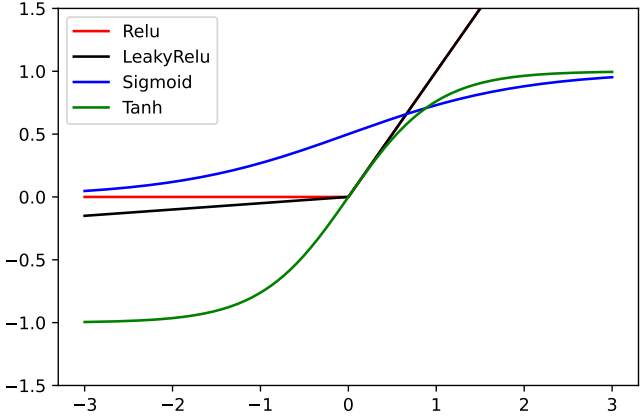
$$v_j(n) = \sum_{i=1}^{m} w_{ji}(n) y_i(n)$$

The signal at the neuron is

$$y_j(n) = \phi(v_j(n))$$

where $\phi(\cdot)$ is a nonlinear *activation function*. We refer to $y_j(n)$ and $v_j(n)$ as EQ3 and EQ4, respectively.

Example Activation Functions



Setting up gradient descent

Keeping our eye on the prize

Out goal is to determine the update rule for the weights of the neural network using gradient descent. Therefore, we need to find the gradient of the error w.r.t. the parameters $\mathbf{w}$. Recall that the update for gradient descent is given by:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t) = \mathbf{w}(t) - \eta \frac{\partial E}{\partial \mathbf{w}}$$

Finding the Gradient is Challenging

The term $\mathbf{w}$ does not show up in the expression for E, which makes finding $\frac{\partial E}{\partial \mathbf{w}}$ more challenging. Finding the gradient is going to be very difficult unless we look at the problem differently.

Back to the basics of calculus

Calculus 101: The Chain Rule

We need to find an expression for $\partial E(n)/\partial w_{ji}(n)$. One way to find this expression is to use the chain rule of calculus. The partial $\partial E(n)/\partial w_{ji}(n)$ represents the *sensitivity factor* that determines the direction of the search weight space for the synaptic weight $w_{ji}(n)$.

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \cdot \frac{\partial e_j(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

Our goal is to work with derivatives above to find the gradient updates when we're at an output neuron.

Filling in the Missing Pieces of $\partial E(n)/\partial w_{ji}(n)$

$$E(n) = \frac{1}{2}\sum e_j^2(n)$$

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n)$$

Filling in the Missing Pieces of $\partial E(n)/\partial w_{ji}(n)$

$$e_j(n) = d_j(n) - y_j(n)$$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1$$

Filling in the Missing Pieces of $\partial E(n)/\partial w_{ji}(n)$

$$y_j(n) = \phi(v_j(n))$$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \phi'(v_j(n))$$

Filling in the Missing Pieces of $\partial E(n)/\partial w_{ji}(n)$

$$v_j(n) = \sum w_{ji}(n) y_i(n)$$

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

Filling in the Missing Pieces of $\partial E(n)/\partial w_{ji}(n)$

Putting it all together

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \cdot \frac{\partial e_j(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)} = -e_j(n)\phi'(v_j(n))y_i(n)$$

The Local Gradient

Definition

We're going to define new term that will be useful for the rest of the discussion on backpropagation. The *local gradient* at neuron j is given by:

$$\delta_j(n) = \frac{\partial E(n)}{\partial v_j(n)} = e_j(n)\phi'(v_j(n))$$

The Local Gradient

Case 1: Neuron j is an output node

When neuron j is located in the output layer of the network, it is supplied with a desired response. We can use EQ1 to compute the error $e_j(n)$ associated with this neuron. Having determined $e_j(n)$ we can easily calculate the local gradient $\delta_j(n)$.

The Local Gradient

Case 1: Neuron j is an output node

When neuron j is located in the output layer of the network, it is supplied with a desired response. We can use EQ1 to compute the error $e_j(n)$ associated with this neuron. Having determined $e_j(n)$ we can easily calculate the local gradient $\delta_j(n)$.

Case 2: Neuron j is a hidden node

When neuron j is located in the hidden layer of the network, there is not a desired response. The error signal for a hidden neuron would have to be determined recursively and working backwards in terms of the error of all the neurons to which that hidden neuron is directly connected. *This is where backpropogation gets interesting!*

Redefining the Local Gradient

New definitions of the local gradient and error at the output

Let j denote a neuron at a hidden layer (i.e., j is no longer used for an arbitrary neuron). Then the local gradient at hidden neuron j is given by

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \phi'(v_j(n))$$

Redefining the Local Gradient

New definitions of the local gradient and error at the output

Let j denote a neuron at a hidden layer (i.e., j is no longer used for an arbitrary neuron). Then the local gradient at hidden neuron j is given by

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \phi'(v_j(n))$$

The error gradient at this neuron is given by $\frac{\partial E(n)}{\partial y_j(n)}$ and we now use k to denote neurons at the output layer.

$$E(n) = \frac{1}{2} \sum_{k=1}^{C} e_k^2(n),$$

Redefining the Local Gradient

New definitions of the local gradient and error at the output

Let j denote a neuron at a hidden layer (i.e., j is no longer used for an arbitrary neuron). Then the local gradient at hidden neuron j is given by

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \phi'(v_j(n))$$

The error gradient at this neuron is given by $\frac{\partial E(n)}{\partial y_j(n)}$ and we now use k to denote neurons at the output layer.

$$E(n) = \frac{1}{2}\sum_{k=1}^{C} e_k^2(n), \qquad \frac{\partial E(n)}{\partial y_j(n)} = \sum_{k=1}^{C} e_k(n)\frac{\partial e_k(n)}{\partial y_j(n)}$$

Redefining the Local Gradient

New definitions of the local gradient and error at the output

Let j denote a neuron at a hidden layer (i.e., j is no longer used for an arbitrary neuron). Then the local gradient at hidden neuron j is given by

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \phi'(v_j(n))$$

The error gradient at this neuron is given by $\frac{\partial E(n)}{\partial y_j(n)}$ and we now use k to denote neurons at the output layer.

$$E(n) = \frac{1}{2}\sum_{k=1}^{C} e_k^2(n), \qquad \frac{\partial E(n)}{\partial y_j(n)} = \sum_{k=1}^{C} e_k(n)\frac{\partial e_k(n)}{\partial y_j(n)} = \sum_{k=1}^{C} e_k(n)\frac{\partial e_k(n)}{\partial v_k(n)}\frac{\partial v_k(n)}{\partial y_j(n)}$$

Filling in the missing pieces

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_{k=1}^{C} e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

We have $e_k(n) = d_k(n) - y_k(n)$

$$\frac{\partial e_k(n)}{\partial v_k(n)} = \frac{\partial}{\partial v_k(n)} \{d_k(n) - y_k(n)\} = \frac{\partial}{\partial v_k(n)} \{d_k(n) - \phi_k(v_k(n))\} = -\phi_k(v_k(n))$$

$$\frac{\partial v_k(n)}{\partial y_j(n)} = \frac{\partial}{\partial y_j(n)} \left\{ \sum_{j=1}^{m} w_{kj}(n) y_j(n) \right\} = w_{kj}(n)$$

Filling in the missing pieces

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_{k=1}^{C} e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} = -\sum_{k=1}^{C} \underbrace{e_k(n)\phi_k(v_k(n))}_{\text{Seen this before? } \delta_k(n)} w_{kj}(n)$$

We have $e_k(n) = d_k(n) - y_k(n)$

$$\frac{\partial e_k(n)}{\partial v_k(n)} = \frac{\partial}{\partial v_k(n)} \{d_k(n) - y_k(n)\} = \frac{\partial}{\partial v_k(n)} \{d_k(n) - \phi_k(v_k(n))\} = -\phi_k(v_k(n))$$

$$\frac{\partial v_k(n)}{\partial y_j(n)} = \frac{\partial}{\partial y_j(n)} \left\{ \sum_{j=1}^{m} w_{kj}(n) y_j(n) \right\} = w_{kj}(n)$$

The Local Gradient

New definition of the local gradient, $\delta_j(n)$

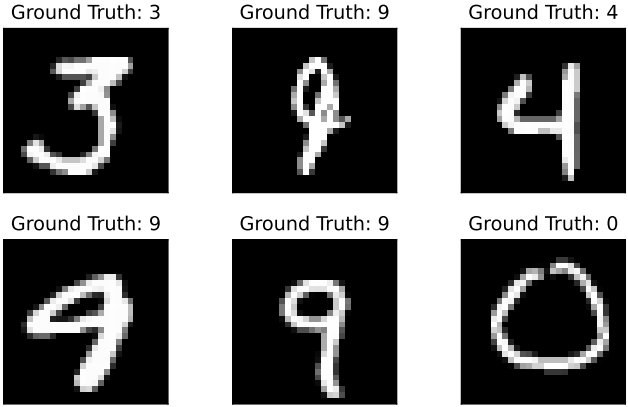
Let j denote a neuron at a hidden layer (i.e., j is no longer used for an arbitrary neuron). Then the local gradient at hidden neuron j is given by

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \phi'(v_j(n)) = \phi'(v_j(n)) \sum_{k=1}^{C} \delta_k(n) w_{kj}(n)$$

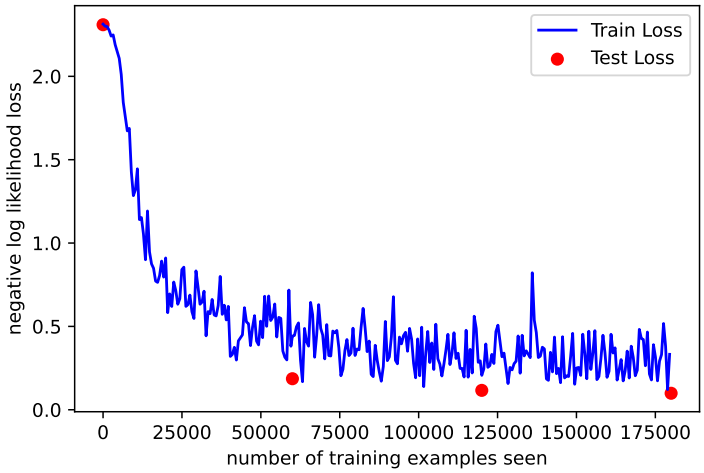
Update rule for any neuron

$$\begin{pmatrix} \text{weight} \\ \text{correction} \\ \Delta w_{ij}(n) \end{pmatrix} = \eta \begin{pmatrix} \text{local} \\ \text{gradient} \\ \delta_j(n) \end{pmatrix} \begin{pmatrix} \text{Input Signal} \\ \text{at neuron } i \\ y_i(n) \end{pmatrix}$$

Example: MNIST



Example: MNIST



The Vanishing Gradient

- Notice that ϕ' is a term in the front of the local gradient, which are multiplied by other local gradients.
- If the nonlinear activation function saturates then we might run into issues if we're multiplying a bunch on numbers on $(-1, 1)$

The Vanishing Gradient

- Notice that ϕ' is a term in the front of the local gradient, which are multiplied by other local gradients.
- If the nonlinear activation function saturates then we might run into issues if we're multiplying a bunch on numbers on $(-1, 1)$

The Derivative of the Logistic Function

$$\frac{\mathrm{d}}{\mathrm{d}x}\left\{\frac{1}{1+\mathrm{e}^{-x}}\right\} = \frac{\mathrm{e}^{-x}}{(1+\mathrm{e}^{-x})^2}$$

The Vanishing Gradient

- Notice that ϕ' is a term in the front of the local gradient, which are multiplied by other local gradients.
- If the nonlinear activation function saturates then we might run into issues if we're multiplying a bunch on numbers on $(-1, 1)$

The Derivative of the Logistic Function

$$\frac{\mathrm{d}}{\mathrm{d}x}\left\{\frac{1}{1+\mathrm{e}^{-x}}\right\} = \frac{1-1+\mathrm{e}^{-x}}{(1+\mathrm{e}^{-x})^2}$$

The Vanishing Gradient

- Notice that ϕ' is a term in the front of the local gradient, which are multiplied by other local gradients.
- If the nonlinear activation function saturates then we might run into issues if we're multiplying a bunch on numbers on $(-1, 1)$

The Derivative of the Logistic Function

$$\frac{\mathrm{d}}{\mathrm{d}x}\left\{\frac{1}{1+\mathrm{e}^{-x}}\right\} = \frac{1-1+\mathrm{e}^{-x}}{(1+\mathrm{e}^{-x})^2} = \frac{1+\mathrm{e}^{-x}}{(1+\mathrm{e}^{-x})^2} - \frac{1}{(1+\mathrm{e}^{-x})^2}$$

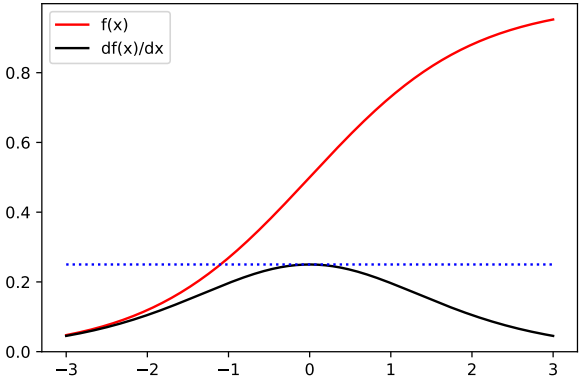
The Vanishing Gradient

- Notice that ϕ' is a term in the front of the local gradient, which are multiplied by other local gradients.
- If the nonlinear activation function saturates then we might run into issues if we're multiplying a bunch on numbers on $(-1, 1)$

The Derivative of the Logistic Function

$$\frac{\mathrm{d}}{\mathrm{d}x}\left\{\frac{1}{1+\mathrm{e}^{-x}}\right\} = \frac{1-1+\mathrm{e}^{-x}}{(1+\mathrm{e}^{-x})^2} = \frac{1+\mathrm{e}^{-x}}{(1+\mathrm{e}^{-x})^2} - \frac{1}{(1+\mathrm{e}^{-x})^2}$$
$$= \phi(x)(1-\phi(x)) \leq \frac{1}{4}$$

The Sigmoid



The function $f(x)$ is a sigmoid. The derivative has a maximum value of $1/4$.

Consequences of the Vanishing Gradient

- The multiplication of the local gradients (and ones coming from the previous layer) can cause the gradient to effectively vanish by the time the updates are done at layers close to the input.
  - Using backpropagation alone is not enough to train a deep multilayer perceptron. We're going to need a different view of training *deep neural network* with an MLP.
  - Relu-like activation functions can help; however, changing the activation function is not going to be sufficient to *effectively* train a deep net.
- Future lectures cover how to effectively train deep nets, but the main take away here is that backpropagation works well at training shallow nets; however, not deep nets.

Tips to Training Networks

- The inputs of your data should be scaled appropriately (e.g., normalization, standardization, etc.)
- Outputs of a classification net should be encoded as one-hot vectors; however, we can use *sparse representation* and *sparse cross-entropy* to avoid directly converting to one-hot.
- Training a net is a nonconvex task so different initialization of the weights will lead to different nets.
- Choosing the "right" number of layers and nodes at a layer is a *heuristic* process. Use the homework as an opportunity to gain experience with the sensitivity of a net to these parameters

Tips to Training Networks

- On choosing the learning rate: too large, too small, just right...

$$\eta_{\text{opt}} = \left(\frac{\partial^2 E}{\partial w^2} \right)^{-1}$$

  In practice, this terms is tedious to compute every step.
- Anneal the learning rate over time. Choose a larger η initially then reduce the value of η as the epochs increase
- Use early stopping by monitoring the validation error. If the validation error goes up after k successive increases in the loss then stop!
- There is more to put in this slide but you're going to need to get some practice to learn these challenges.

Remember Regularization?

Estimation for ℓ^1-norm (left) and ℓ^2-norm (right) regularization on $\mathbf{w}$. We see the contours of the error function and the regularization constraint on $\|\mathbf{w}\|_1 \leq \tau$ and $\|\mathbf{w}\|_2^2 \leq \tau^2$.

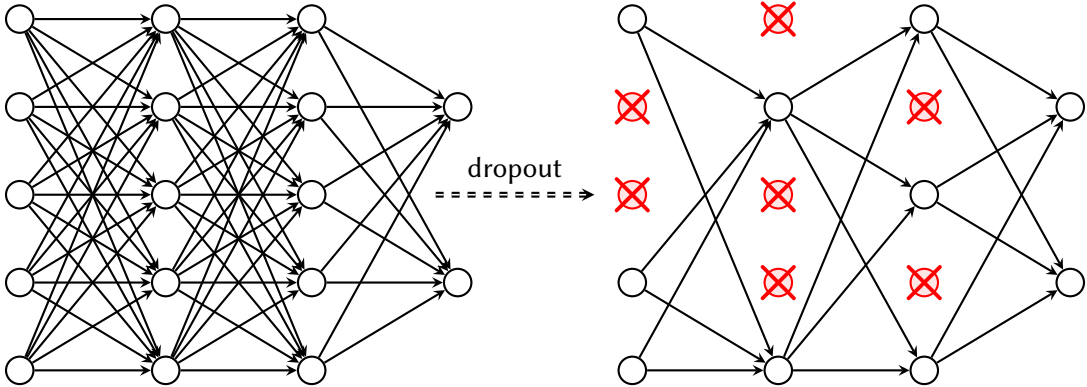
Dropout

Dropout: A new way to regularize a network

- The central idea behind *dropout* is to randomly dropout nodes in the neural network with a probability p. After the nodes are "dropped out" the remain network is a subnetwork of the original.
- Dropout has been shown to work quite well at preventing overfitting and allowing the network to find a good local minimum.
- **Recommendation**: Use dropout!

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

Dropout in a Neural Network



Code Availability and Python Notebook

Code used for this chapter is available at
https://github.com/rahulbhadani/CPE490_590_Sp2025/blob/master/Code/Chapter_09_Neural_Network.ipynb

The End