Introduction to Data Science Topic-10

- Instructor: Professor Henry Horng-Shing Lu,
  Institute of Statistics, National Yang Ming Chiao Tung University, Taiwan
  Email: henryhslu@nycu.edu.tw

- WWW: http://misg.stat.nctu.edu.tw/hslu/course/DataScience.htm
- Classroom: ED B27 (新竹市大學路1001號工程四館B27教室)
- References:
  M. A. Pathak, Beginning Data Science with R, 2014, Springer-Verlag.
  K.-T. Tsai, Machine Learning for Knowledge Discovery with R: Methodologies for Modeling, Inference, and Prediction, 2021, Chapman and Hall/CRC.
- Evaluation: Homework: 70%, Term Project: 30%
- Office hours: By appointment

Course Outline

10 Topics and 10 Homeworks:

- **Introduction of Data Science**
- **Introduction of R and Python**
- **Cleaning Data into R and Python**
- **Data Visualization**
- **Exploratory Data Analysis**
- **Regression (Supervised Learning)**
- **Classification (Supervised Learning)**
- **Text Mining**
- **Clustering (Unsupervised Learning)**
- **Neural Network and Deep Learning**

Neural Network with R

Link: https://www.kaggle.com/code/royshih23/topic-10-mlp/notebook

Date: https://www.kaggle.com/competitions/digit-recognizer

Reference: https://www.kaggle.com/code/mauriciofigueiredo/mlp-simples-com-keras-para-iniciantes

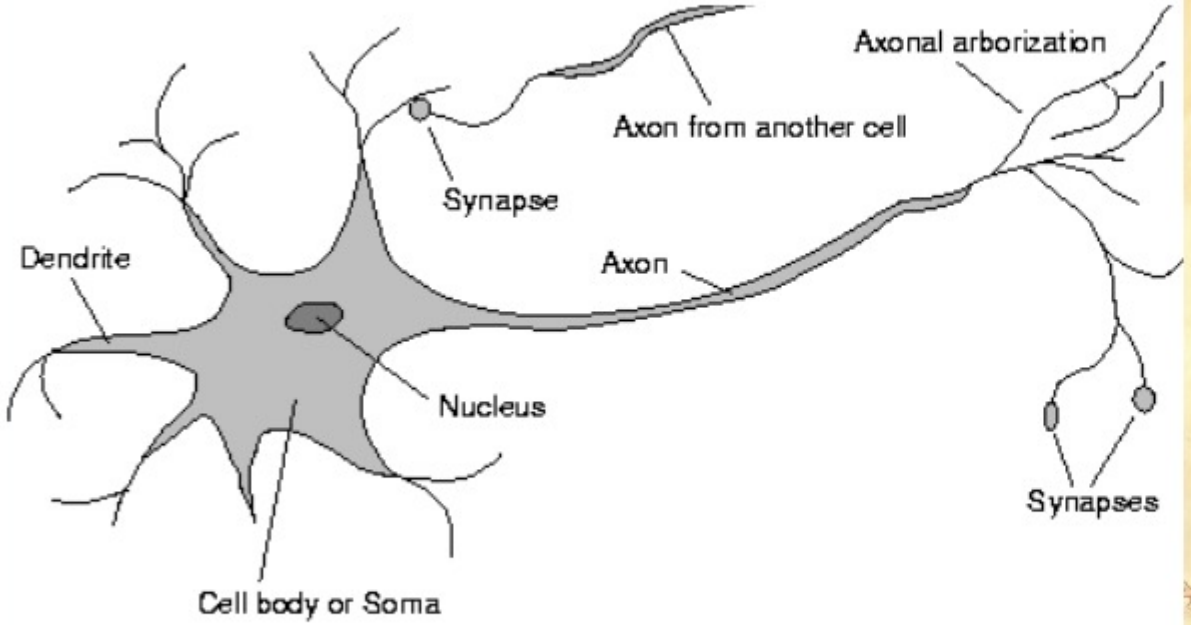
Neural network

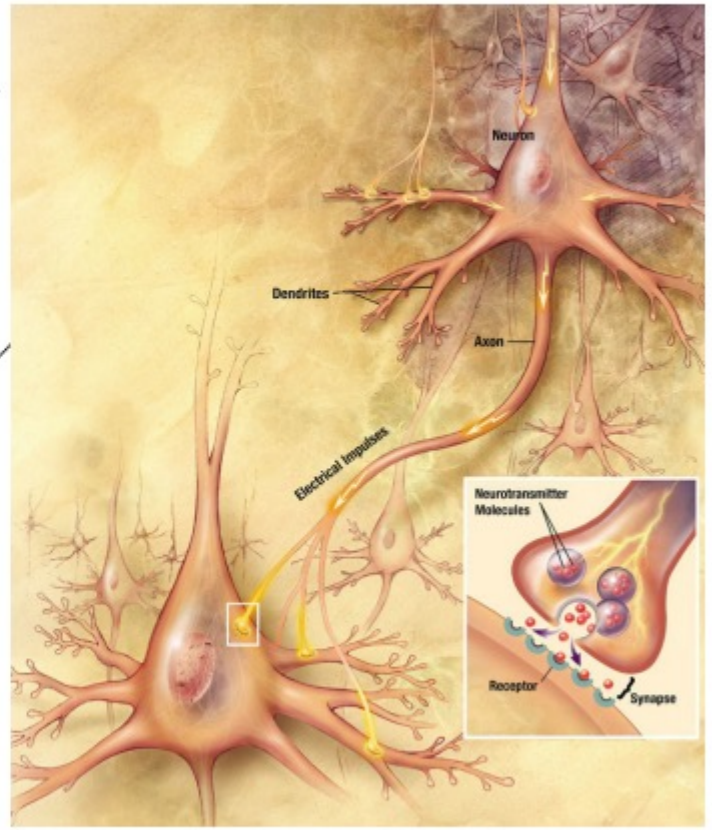
Neural Function

- Brain function (thought) occurs as the result of the firing of **neurons**
- Neurons connect to each other through **synapses**, which propagate **action potential** (electrical impulses) by releasing **neurotransmitters**
  - Synapses can be **excitatory** (potential-increasing) or **inhibitory** (potential-decreasing), and have varying **activation thresholds**
  - Learning occurs as a result of the synapses' **plasticicity**: They exhibit long-term changes in connection strength
- There are about 10^{11} neurons and about 10^{14} synapses in the human brain!

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Biology of a Neuron





https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Brain Structure

- Different areas of the brain have different functions
  - Some areas seem to have the same function in all humans (e.g., Broca's region for motor speech); the overall layout is generally consistent
  - Some areas are more plastic, and vary in their function; also, the lower-level structure and function vary greatly
- We don't know how different functions are "assigned" or acquired
  - Partly the result of the physical layout / connection to inputs (sensors) and outputs (effectors)
  - Partly the result of experience (learning)
- We *really* don't understand how this neural structure leads to what we perceive as "consciousness" or "thought"

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Neural network

Origins: Algorithms that try to mimic the brain.

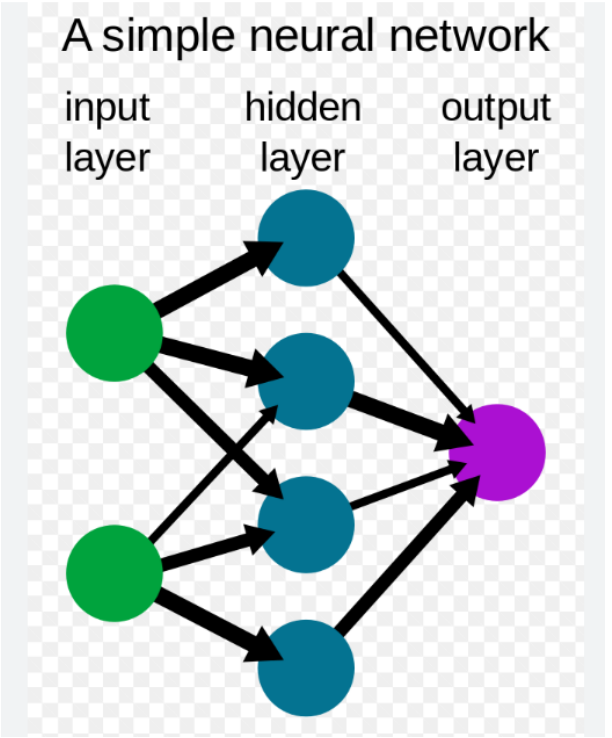
- Very widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence: State-of-the-art technique for many applications
- Artificial neural networks are not nearly as complex or intricate as the actual brain structure

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Neural network

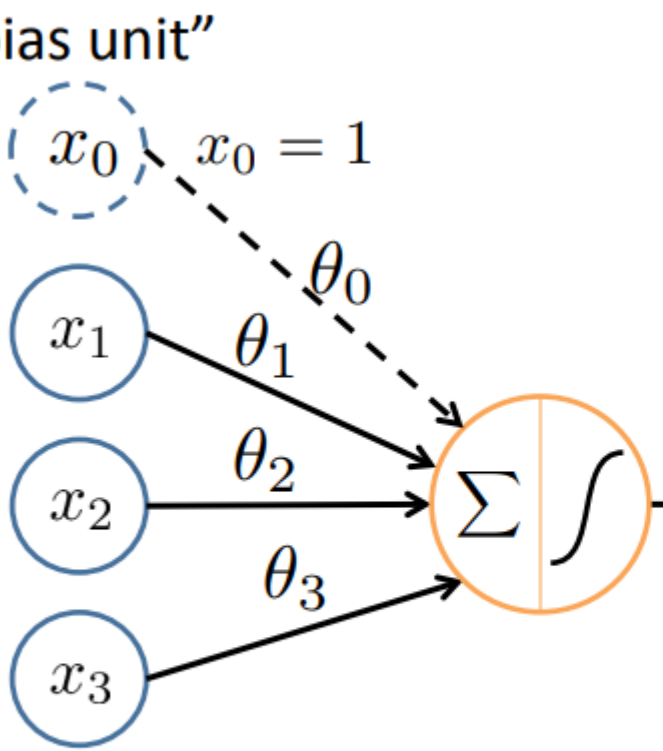
- Neural networks are made up of nodes or units, connected by links.
- Each link has an associated weight and activation level.
- Each node has an input function (typically summing over weighted inputs), an activation function, and an output.



Single hidden layer feed-forward network

https://en.wikipedia.org/wiki/Neural_network
https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Neuron Model: Logis1c Unit



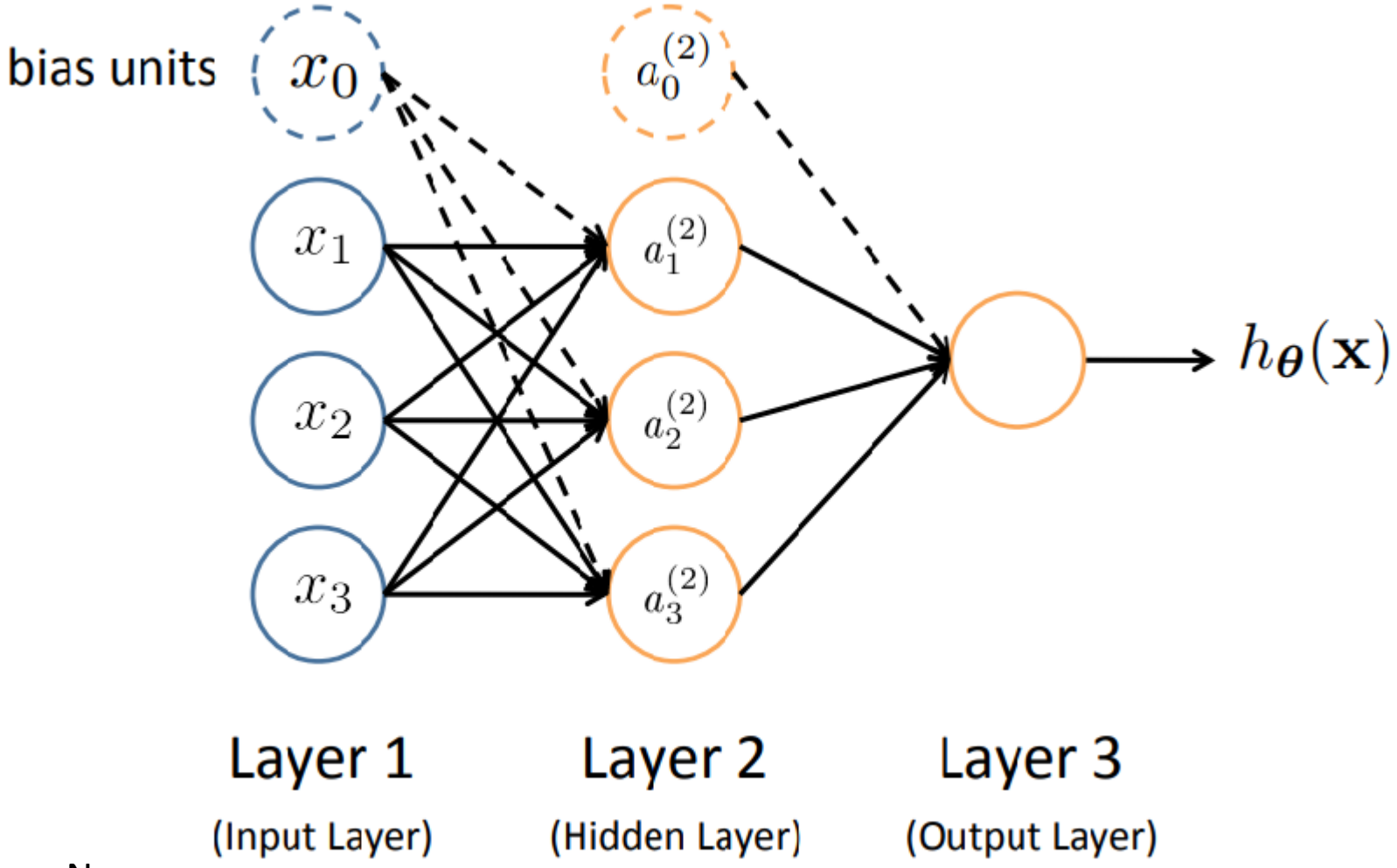
$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = g\left(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}\right) = \frac{1}{1 + e^{-\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}}}$$

Sigmoid (logistic) activation function: $g(z) = \dfrac{1}{1 + e^{-z}}$

Based on slide by Andrew Ng

Neuron Network



Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Neuron Network

1. Feed-Forward Process

2. Back-propagation

Feed-Forward Process

- Input layer units are set by some exterior function (think of these as sensors), which causes their output links to be activated at the specified level
- Working forward through the network, the input function of each unit is applied to compute the input value.
  - Usually, this is just the weighted sum of the ac1va1on on the links feeding into this node
- The activation function transforms this input function into a final value
  - Typically this is a nonlinear function, often a sigmoid function corresponding to the “threshold” of that node

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

KS

what is Feed-Forward Process in Neural Network?

The feed-forward process is a fundamental component of a neural network, which is a machine learning model inspired by the structure and function of biological neural networks. In a neural network, the feed-forward process refers to the flow of information through the network from the input layer to the output layer, without any feedback loops or recurrent connections.

Here's a step-by-step overview of the feed-forward process in a typical neural network:

1. Input Layer: The neural network receives input data, which could be numerical values, images, text, or any other form of data. Each input feature corresponds to a neuron in the input layer.
2. Neuron Activation: Each neuron in the input layer takes the input data and applies an activation function, typically a nonlinear function such as the sigmoid or ReLU (Rectified Linear Unit) function. The activation function determines whether the neuron fires (produces an output signal) based on the weighted sum of its inputs.
3. Hidden Layers: The output of the neurons in the input layer is passed on to one or more hidden layers in the neural network. A hidden layer consists of multiple neurons, each applying an activation function to its inputs and producing an output.

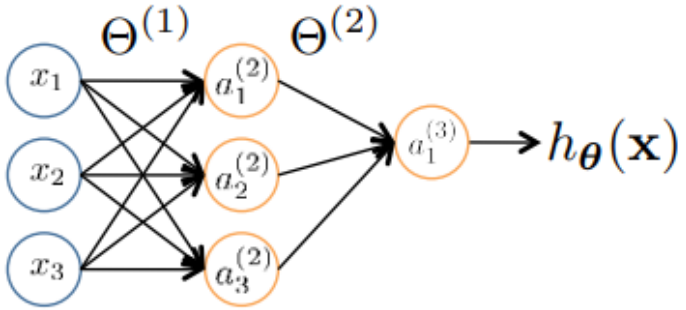
https://chat.openai.com

4. Weighted Connections: Each connection between neurons in adjacent layers is associated with a weight. The output of a neuron is multiplied by its corresponding weights before being passed as inputs to the neurons in the next layer. These weights represent the learned parameters of the neural network and are adjusted during the training process.
5. Output Layer: The final hidden layer's outputs are fed into the output layer, which produces the network's predictions or outputs. The activation function used in the output layer depends on the nature of the problem being solved. For example, a regression task may use a linear activation function, while a binary classification task may use a sigmoid function.
6. Output Generation: The output layer's activation values are interpreted as the predictions or outputs of the neural network for the given input.

During the feed-forward process, information flows from the input layer through the hidden layers to the output layer, with each layer transforming and processing the data. This process is called feed-forward because the information flows only in one direction, without any feedback or loops. The weights and biases of the network are typically adjusted through a process called backpropagation, which involves comparing the network's predictions with the desired outputs and updating the weights accordingly, aiming to minimize the prediction error.

https://chat.openai.com

Neuron Network



$a_i^{(j)}$ = "activation" of unit i in layer j

$\Theta^{(j)}$ = weight matrix controlling function mapping from layer j to layer $j+1$

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

If network has s_j units in layer j *and* s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ has dimension $s_{j+1} \times (s_j+1)$.

$$\Theta^{(1)} \in \mathbb{R}^{3\times 4} \qquad \Theta^{(2)} \in \mathbb{R}^{1\times 4}$$

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

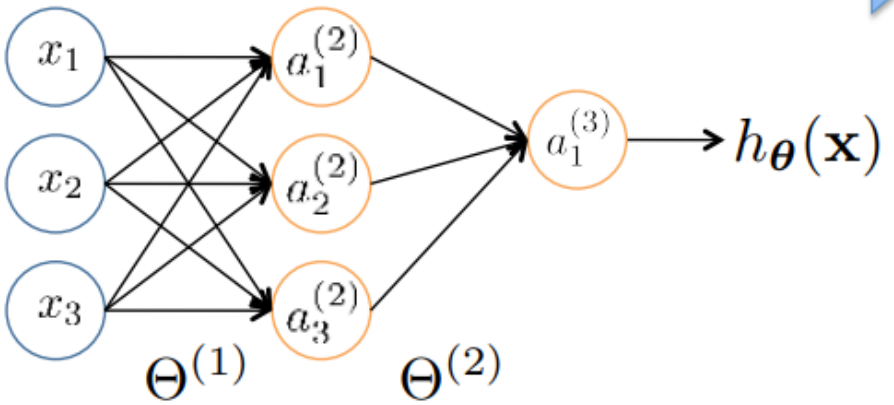
Vectorization

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right) = g\left(z_1^{(2)}\right)$$

$$a_2^{(2)} = g\left(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\right) = g\left(z_2^{(2)}\right)$$

$$a_3^{(2)} = g\left(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\right) = g\left(z_3^{(2)}\right)$$

$$h_\Theta(\mathbf{x}) = g\left(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}\right) = g\left(z_1^{(3)}\right)$$



Feed-Forward Steps:

$$\mathbf{z}^{(2)} = \Theta^{(1)}\mathbf{x}$$

$$\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$$

Add $a_0^{(2)} = 1$

$$\mathbf{z}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}$$

$$h_\Theta(\mathbf{x}) = \mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$$

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

KS

what is Vectorization in Neural Network?

Vectorization in neural networks refers to the process of performing operations on entire vectors or matrices rather than individual elements. It involves leveraging the mathematical properties of linear algebra and parallel computing to perform computations more efficiently. Vectorization plays a crucial role in optimizing the performance of neural networks, especially when dealing with large datasets.

In traditional programming, operations are typically performed element-wise, meaning each element of a vector or matrix is processed individually in a loop. However, this approach can be computationally expensive and time-consuming, especially when dealing with large datasets. Vectorization, on the other hand, allows for performing operations on entire vectors or matrices with a single instruction, taking advantage of parallel computing and optimized libraries.

In the context of neural networks, vectorization is used in various aspects, including:

https://chat.openai.com

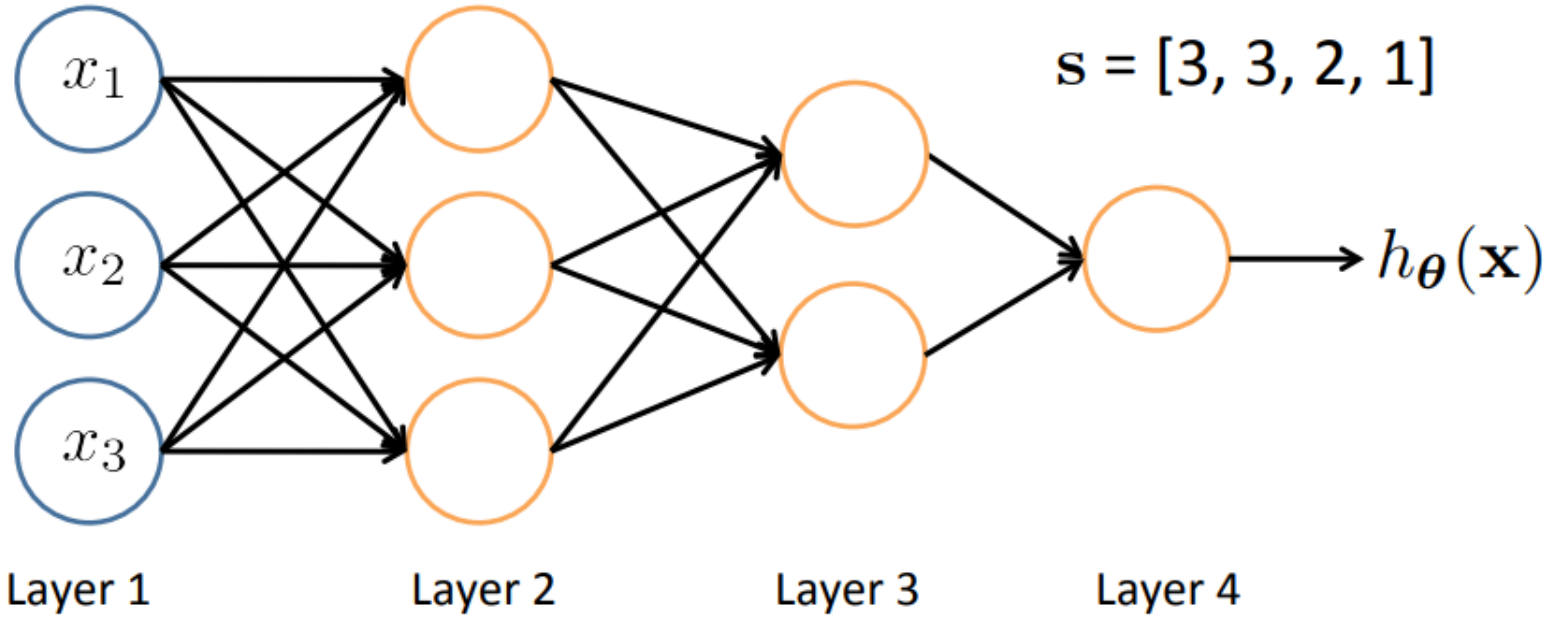
In the context of neural networks, vectorization is used in various aspects, including:

1. Forward Propagation: During the feed-forward process, vectorization allows for performing computations for multiple inputs simultaneously. Instead of processing one input at a time, the input data is organized as a matrix, where each row represents an input example. By utilizing matrix operations, such as matrix multiplication, the computations can be efficiently performed in parallel for all input examples.
2. Backpropagation: In the backpropagation algorithm, which is used to update the weights of a neural network during training, vectorization helps in performing the calculations efficiently. The gradient computations can be vectorized, allowing for the simultaneous update of weights for multiple training examples. This leads to faster training times and improved performance.
3. Optimization Algorithms: Many optimization algorithms used in neural networks, such as gradient descent and its variants, can benefit from vectorization. The computation of gradients, updates to the weights, and other operations involved in optimization can be vectorized to process multiple training examples or network parameters simultaneously.

By utilizing vectorization techniques, neural networks can take advantage of efficient linear algebra operations provided by hardware acceleration (e.g., GPUs) and optimized libraries (e.g., NumPy, TensorFlow, PyTorch). This can lead to significant improvements in computational efficiency, allowing for faster training and inference times, especially when working with large datasets and complex network architectures.

https://chat.openai.com

Other Network Architectures



L denotes the number of layers

$\mathbf{s} \in \mathbb{N}^{+L}$ contains the numbers of nodes at each layer

- Not counting bias units
- Typically, $s_0 = d$ (# input features) and $s_{L-1} = K$ (# classes)

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

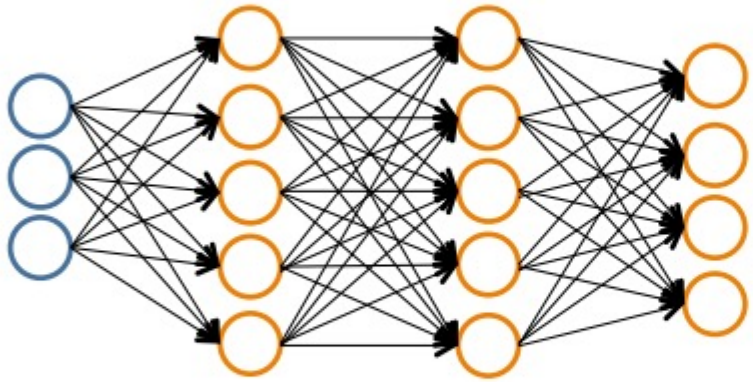
Multiple Output Units: One-vs-Rest

Pedestrian

Car

Motorcycle

Truck



$$h_\Theta(\mathbf{x}) \in \mathbb{R}^K$$

We want:

$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

when pedestrian

$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

when car

$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

when motorcycle

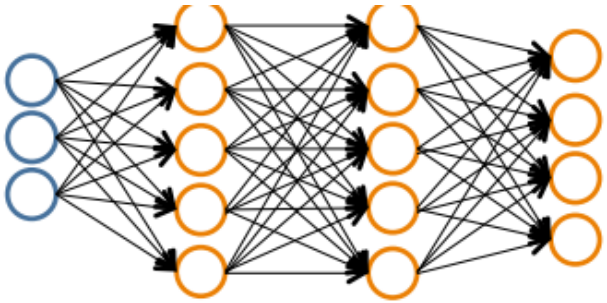
$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

when truck

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Multiple Output Units: One-vs-Rest



$h_\Theta(\mathbf{x}) \in \mathbb{R}^K$

We want:

$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ when pedestrian

$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ when car

$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when motorcycle

$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ when truck

---

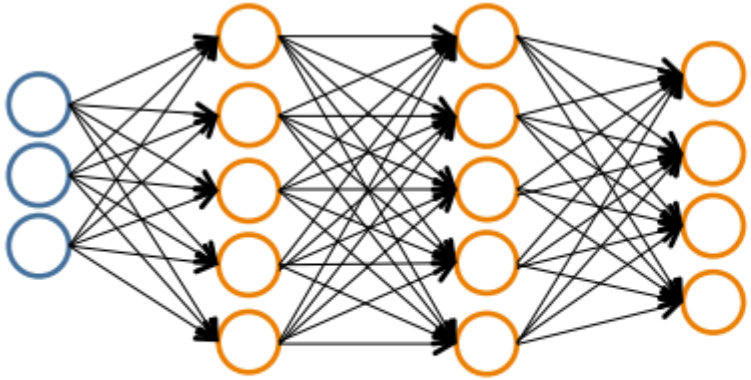
- Given $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$
- Must convert labels to 1-of-K representation
  - e.g., $\mathbf{y}_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when motorcycle, $\mathbf{y}_i = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ when car, etc.

ased on slide by Andrew Ng

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Neural Network Classification



Given:

$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$

$\mathbf{s} \in \mathbb{N}^{+L}$ contains # nodes at each layer

- $s_0 = d$ (# features)

Binary classification

y = 0 or 1

1 output unit $(s_{L-1} = 1)$

Multi-class classification (K classes)

$\mathbf{y} \in \mathbb{R}^K$ e.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

K output units $(s_{L-1} = K)$

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

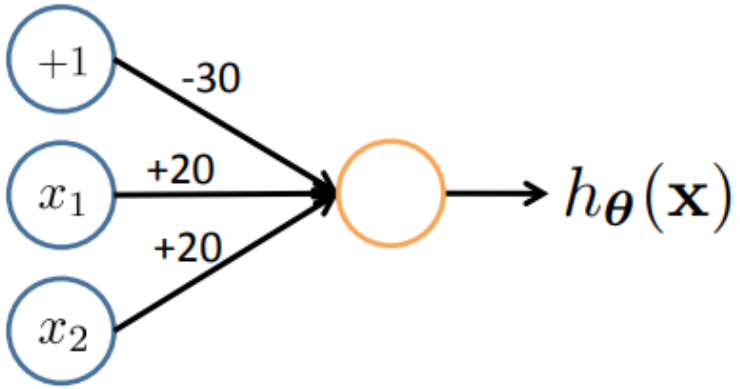
Understanding Representations of Neural network

Representing Boolean Functions

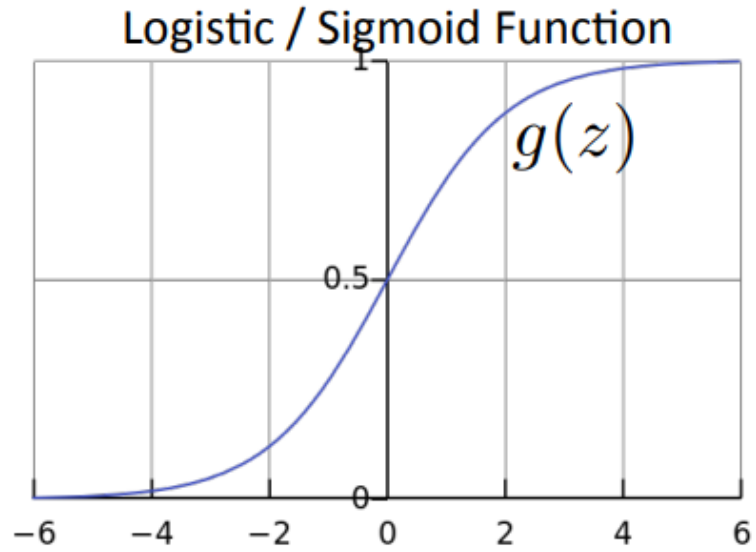
Simple example: AND

$x_1, x_2 \in \{0, 1\}$

$y = x_1 \text{ AND } x_2$



$$\mathrm{h}_\Theta(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$

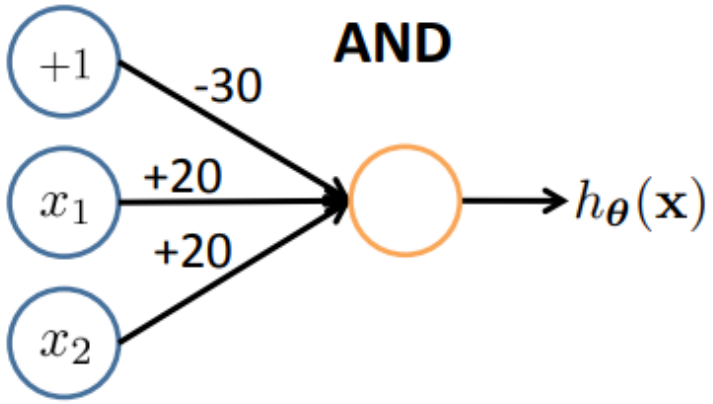


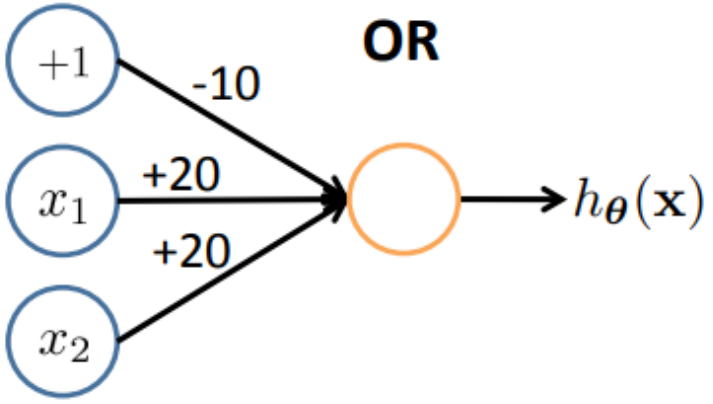
x_1	x_2	$\mathrm{h}_\Theta(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

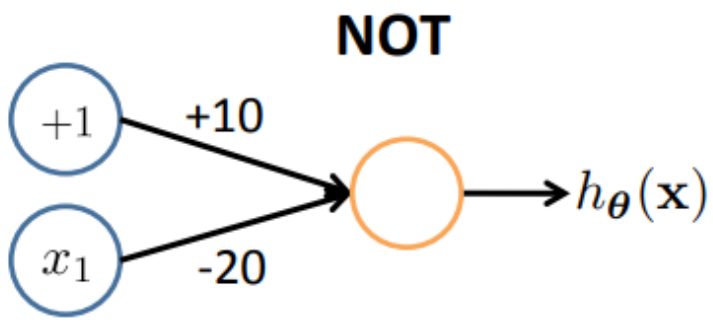
Based on slide by Andrew Ng

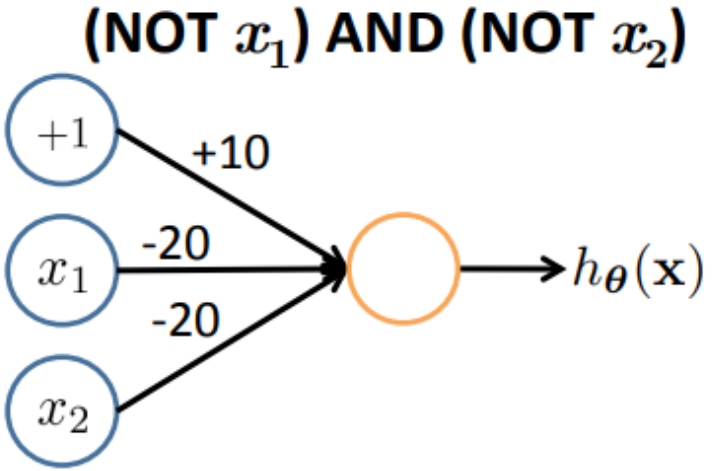
https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Representing Boolean Functions



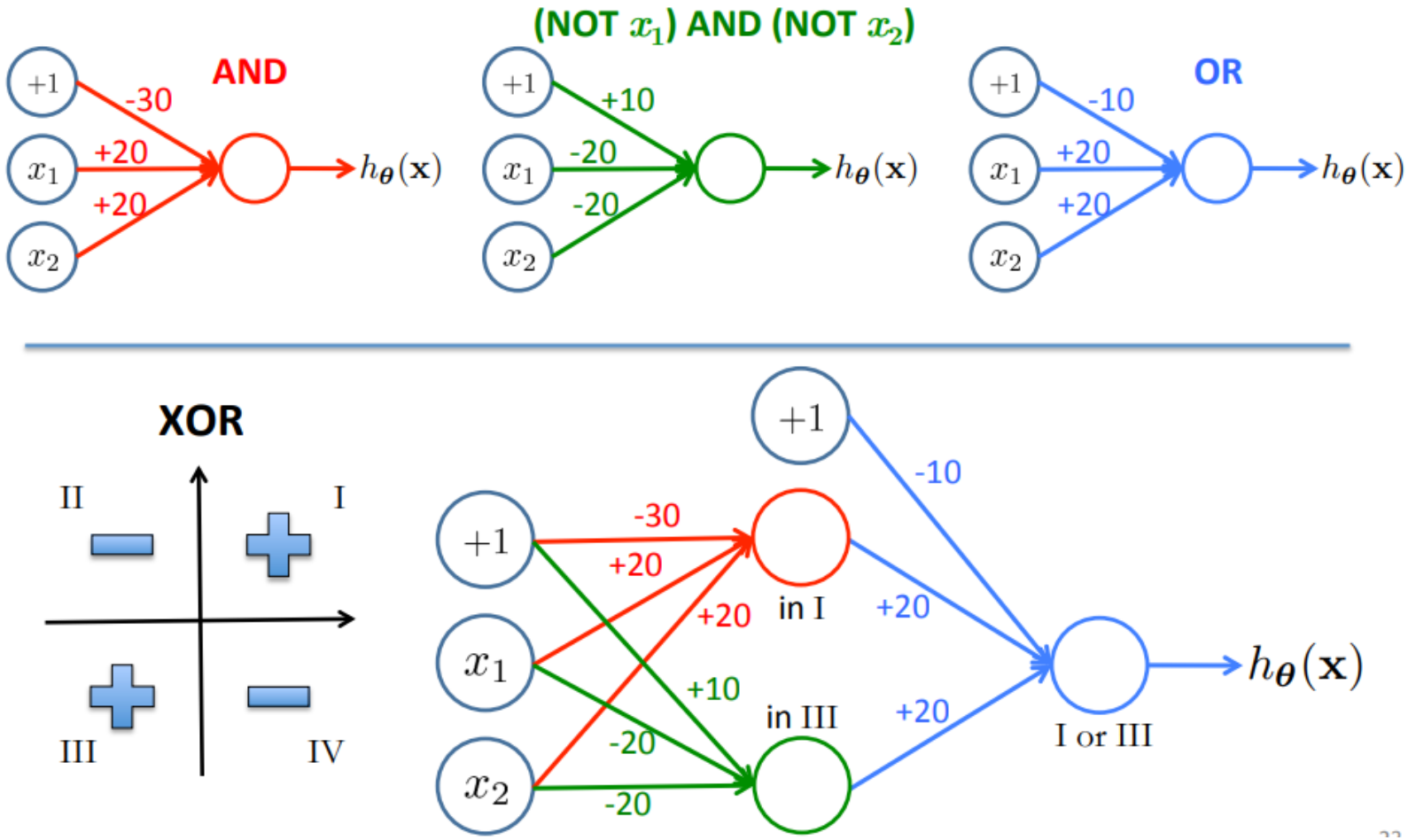






https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

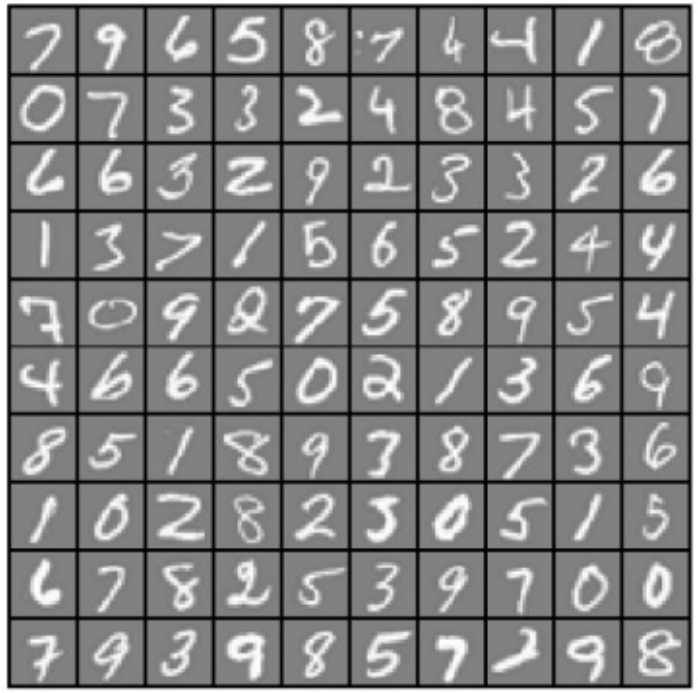
Combining Representations to Create Non-Linear Functions

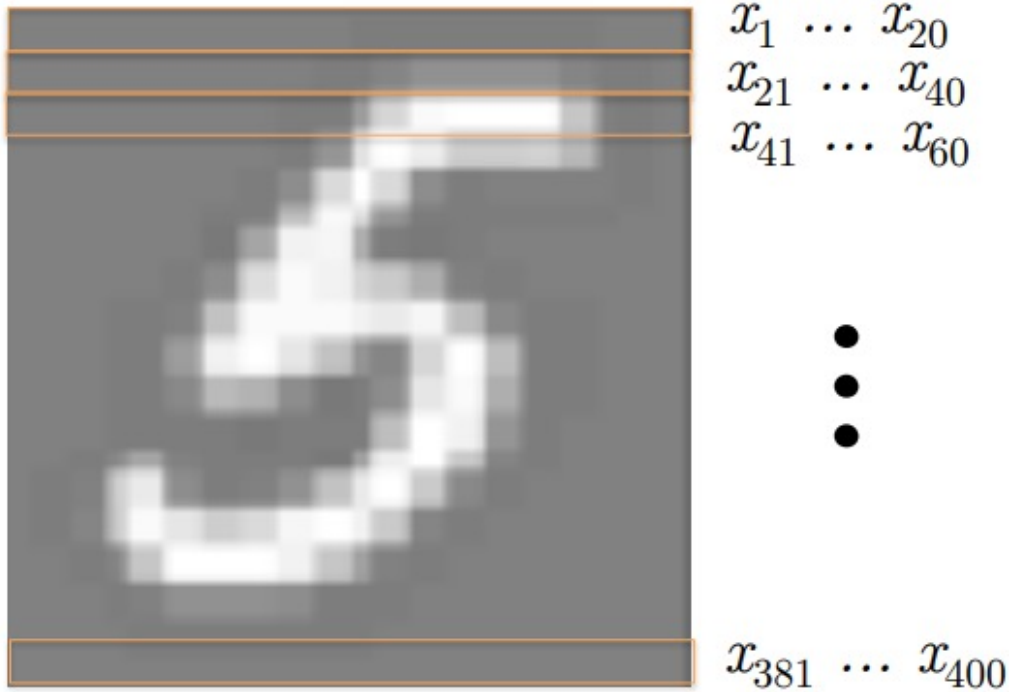


Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Layering Representations





20 × 20 pixel images

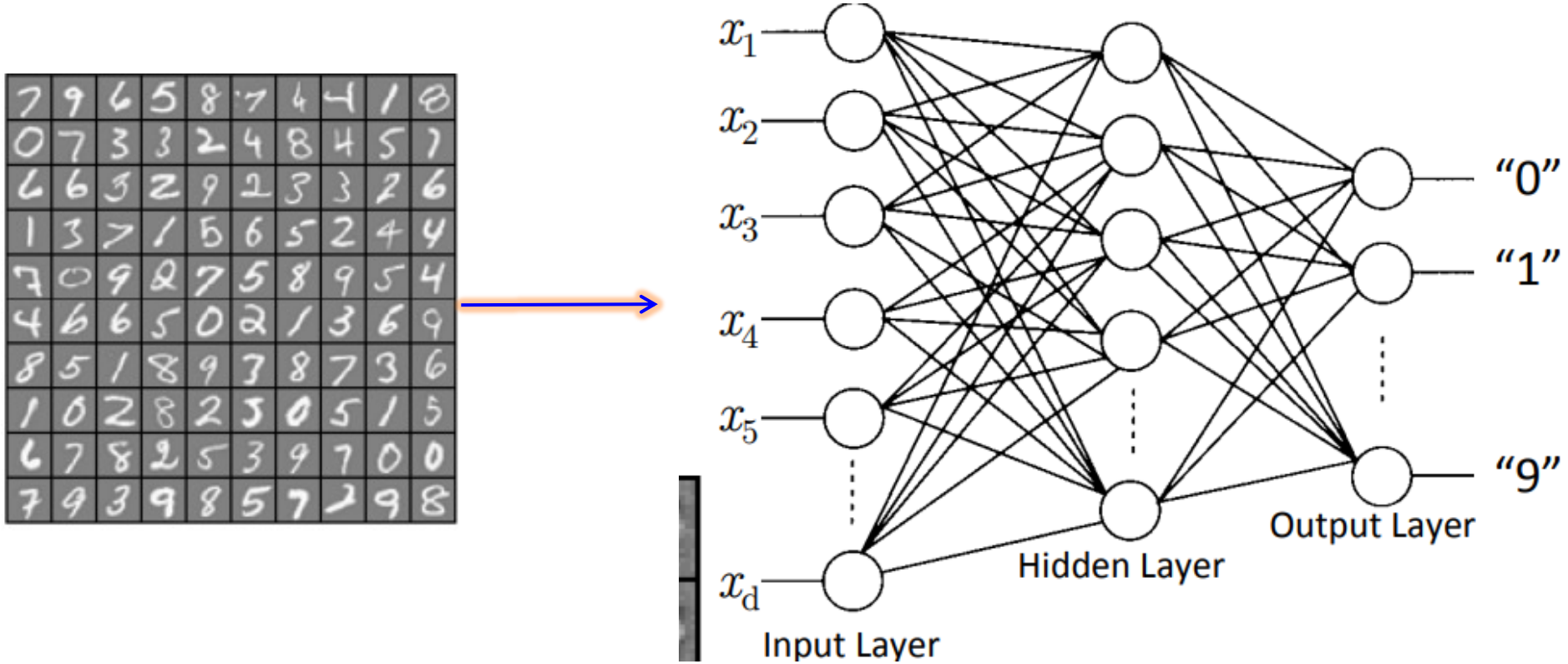
d = 400 10 classes

Each image is "unrolled" into a vector $\mathbf{x}$ of pixel intensities

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Layering Representations



https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Neural Network Learning

Perceptron Learning Rule

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(y - h(\mathbf{x}))\mathbf{x}$$

Equivalent to the intuitive rules:

- If output is correct, don't change the weights
- If output is low ($h(\mathbf{x}) = 0$, $y = 1$), increment weights for all the inputs which are 1
- If output is high ($h(\mathbf{x}) = 1$, $y = 0$), decrement weights for all inputs which are 1

Perceptron Convergence Theorem:

- If there is a set of weights that is consistent with the training data (i.e., the data is linearly separable), the perceptron learning algorithm will converge [Minicksy & Papert, 1969]

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

KS

What is Perceptron Learning Rule in Neural Network?

The Perceptron Learning Rule is a fundamental algorithm in neural network theory used to train a single-layer perceptron, which is the simplest form of a neural network. It was developed by Frank Rosenblatt in the late 1950s and is considered the basis for many later developments in neural network research.

The Perceptron Learning Rule is a supervised learning algorithm that aims to adjust the weights of the perceptron's connections to enable accurate classification of input patterns into different classes. The algorithm follows these steps:

1. Initialization: Initialize the weights and bias of the perceptron randomly or with small values close to zero.
2. Input and Activation: Present an input pattern (vector of features) to the perceptron and compute the weighted sum of the inputs, including the bias term. This sum is passed through an activation function to produce the perceptron's output. The activation function is typically a step function or a threshold function.

https://chat.openai.com

3. Comparison and Update: Compare the perceptron's output to the desired output (target value) for the given input pattern. If the outputs match, no weight update is necessary. However, if there is a misclassification, the weights and bias are adjusted.
4. Weight Update: For misclassified patterns, update the weights and bias using the Perceptron Learning Rule. The rule is based on the concept of gradient descent and aims to minimize the error by adjusting the weights in the direction that reduces the error. The weight update formula is:
   $\Delta w = \eta * (\text{target} - \text{output}) * \text{input}$
   where Δw is the change in weight, η (eta) is the learning rate (a small positive constant), target is the desired output, output is the perceptron's output, and input is the input pattern.
5. Repeat: Iterate through steps 2 to 4 for each input pattern in the training dataset until the perceptron achieves satisfactory performance or a predefined number of iterations.

The Perceptron Learning Rule is a binary classifier, meaning it can classify input patterns into two classes. It is primarily used for linearly separable problems, where the classes can be separated by a hyperplane. However, it may not converge or find a solution if the problem is not linearly separable.

It's important to note that the Perceptron Learning Rule is limited to single-layer perceptrons and cannot be directly applied to train more complex neural network architectures. However, it laid the foundation for more sophisticated learning algorithms, such as backpropagation, which can train multi-layer neural networks.

https://chat.openai.com

Learning in NN: Backpropagation

- Similar to the perceptron learning algorithm, we cycle through our examples
  - If the output of the network is correct, no changes are made
  - If there is an error, weights are adjusted to reduce the error

- The trick is to assess the blame for the error and divide it among the contribu1ng weights

https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Cost Function

Logistic Regression:

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log h_{\boldsymbol{\theta}}(\mathbf{x}_i) + (1-y_i)\log(1-h_{\boldsymbol{\theta}}(\mathbf{x}_i))] + \frac{\lambda}{2n}\sum_{j=1}^{d}\theta_j^2$$

Neural Network:

$$h_\Theta \in \mathbb{R}^K \qquad (h_\Theta(\mathbf{x}))_i = i^{th}\text{output}$$

$$J(\Theta) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik}\log(h_\Theta(\mathbf{x}_i))_k + (1-y_{ik})\log\left(1-(h_\Theta(\mathbf{x}_i))_k\right)\right]$$

$$+\frac{\lambda}{2n}\sum_{l=1}^{L-1}\sum_{i=1}^{s_{l-1}}\sum_{j=1}^{s_l}\left(\Theta_{ji}^{(l)}\right)^2$$

k^{th} class: true, predicted
not k^{th} class: true, predicted

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Optimizing the Neural Network

$$J(\Theta) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik}\log(h_\Theta(\mathbf{x}_i))_k + (1-y_{ik})\log\Big(1-(h_\Theta(\mathbf{x}_i))_k\Big)\right]$$
$$+\frac{\lambda}{2n}\sum_{l=1}^{L-1}\sum_{i=1}^{s_{l-1}}\sum_{j=1}^{s_l}\left(\Theta_{ji}^{(l)}\right)^2$$

Solve via: $\min_\Theta J(\Theta)$

$J(\Theta)$ is not convex, so GD on a neural net yields a local optimum

- But, tends to work well in practice

Need code to compute:

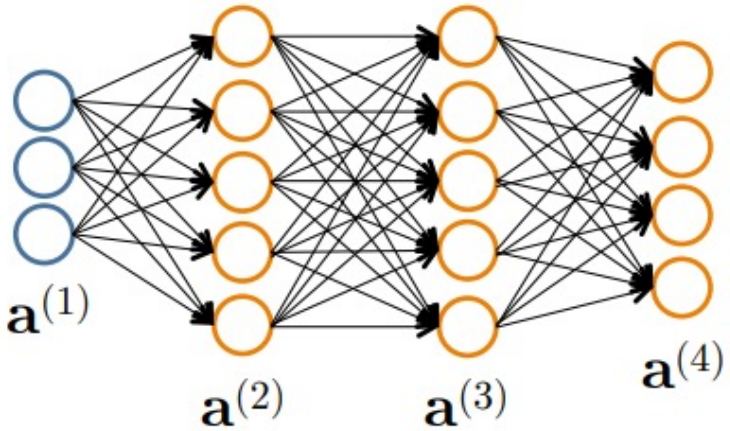
- $J(\Theta)$
- $\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)$

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Forward Propagation

- Given one labeled training instance $(\mathbf{x}, y)$:



Forward Propagation

- $\mathbf{a}^{(1)} = \mathbf{x}$
- $\mathbf{z}^{(2)} = \Theta^{(1)}\mathbf{a}^{(1)}$
- $\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$ [add $a_0^{(2)}$]
- $\mathbf{z}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}$
- $\mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$ [add $a_0^{(3)}$]
- $\mathbf{z}^{(4)} = \Theta^{(3)}\mathbf{a}^{(3)}$
- $\mathbf{a}^{(4)} = h_\Theta(\mathbf{x}) = g(\mathbf{z}^{(4)})$

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Backpropagation Intuition

- Each hidden node j is "responsible" for some fraction of the error $\delta_j^{(l)}$ in each of the output nodes to which it connects
- $\delta_j^{(l)}$ is divided according to the strength of the connection between hidden node and the output node
- Then, the "blame" is propagated back to provide the error values for the hidden layer

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

KS

what is Backpropagation in Neural network?

Backpropagation is a widely used algorithm for training multi-layer neural networks. It is an efficient and effective method for updating the weights of the network based on the observed errors during the training process. Backpropagation enables neural networks to learn complex patterns and make accurate predictions by iteratively adjusting the network's weights to minimize the difference between the predicted outputs and the desired outputs.

Here's an overview of how backpropagation works:

1. Forward Propagation: During the forward propagation step, an input pattern is fed into the neural network, and the outputs of each layer are calculated through the feed-forward process. The input propagates through the layers, and the final output is generated.
2. Error Calculation: The difference between the predicted output and the desired output for the given input pattern is calculated, resulting in an error value. This error value represents the discrepancy between the network's output and the target output.
3. Backward Pass: In the backward pass, the error is propagated back through the network, starting from the output layer and moving towards the input layer. The goal is to compute the contribution of each weight in the network to the overall error.

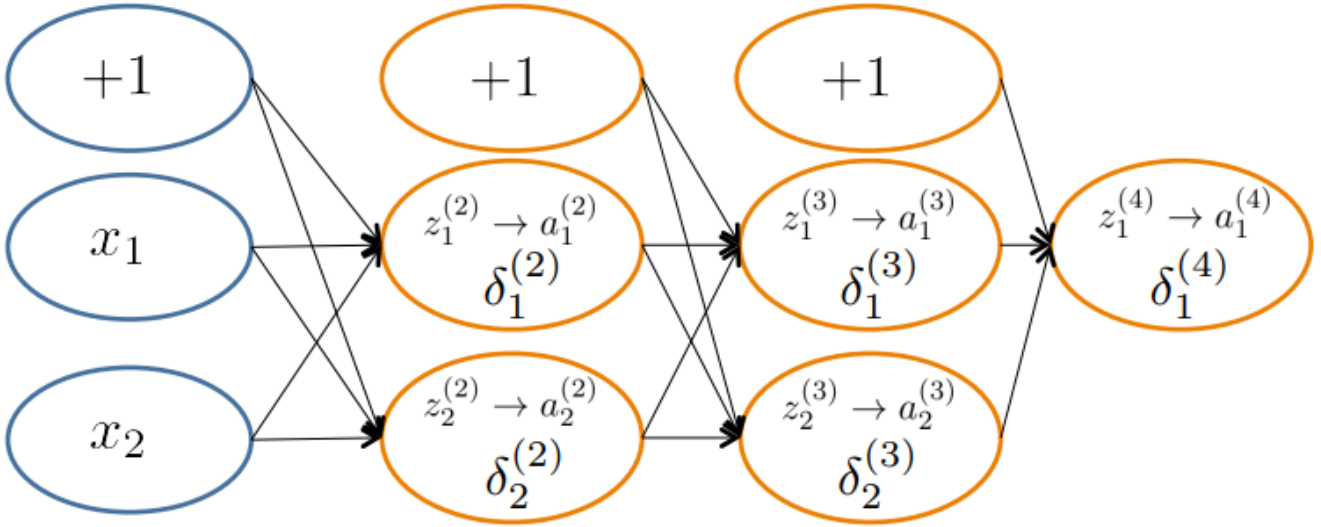
4. Gradient Calculation: The gradients of the weights with respect to the error are computed using the chain rule of calculus. The gradients indicate the direction and magnitude of the weight adjustments required to minimize the error. The derivative of the error with respect to each weight is calculated by considering the activation functions and the weights of the connections in each layer.
5. Weight Update: The calculated gradients are used to update the weights of the network. The weights are adjusted in the direction opposite to the gradient, scaled by a learning rate that determines the step size of the weight updates. The learning rate is a hyperparameter that controls the speed of convergence and must be carefully chosen to avoid overshooting or slow convergence.
6. Iterative Process: Steps 1 to 5 are repeated for each input pattern in the training dataset. The network's weights are updated iteratively, gradually reducing the error and improving the network's ability to make accurate predictions.

Backpropagation allows neural networks to learn from labeled training data by adjusting the weights to minimize the prediction error. Through repeated iterations of forward propagation, error calculation, and weight updates, the network learns to recognize patterns and make accurate predictions on new, unseen data. This process is known as supervised learning because the network learns from examples with known outputs.

Backpropagation has been a key factor in the success of neural networks, enabling them to solve complex problems such as image recognition, natural language processing, and speech recognition. It remains a fundamental algorithm in the field of deep learning.

Regenerate response

Backpropagation Intuition



$\delta_j^{(l)}$ = "error" of node j in layer l

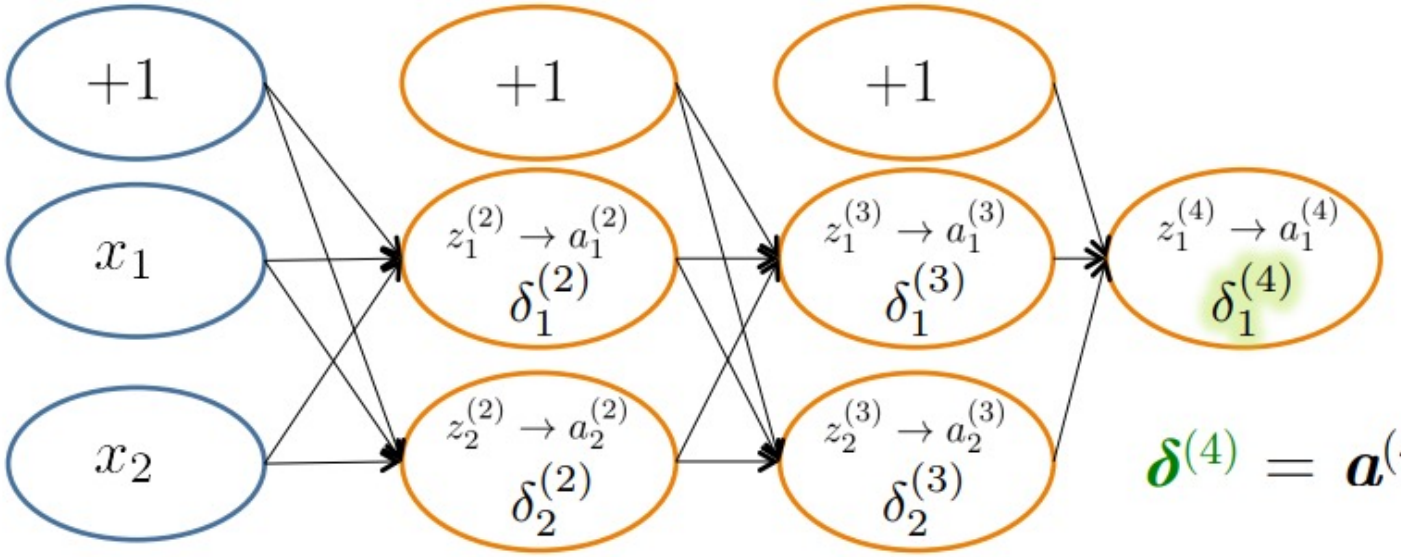
Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where $\text{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Backpropagation Intuition



$$\boldsymbol{\delta}^{(4)} = \boldsymbol{a}^{(4)} - \mathbf{y}$$

$\delta_j^{(l)}$ = "error" of node j in layer l

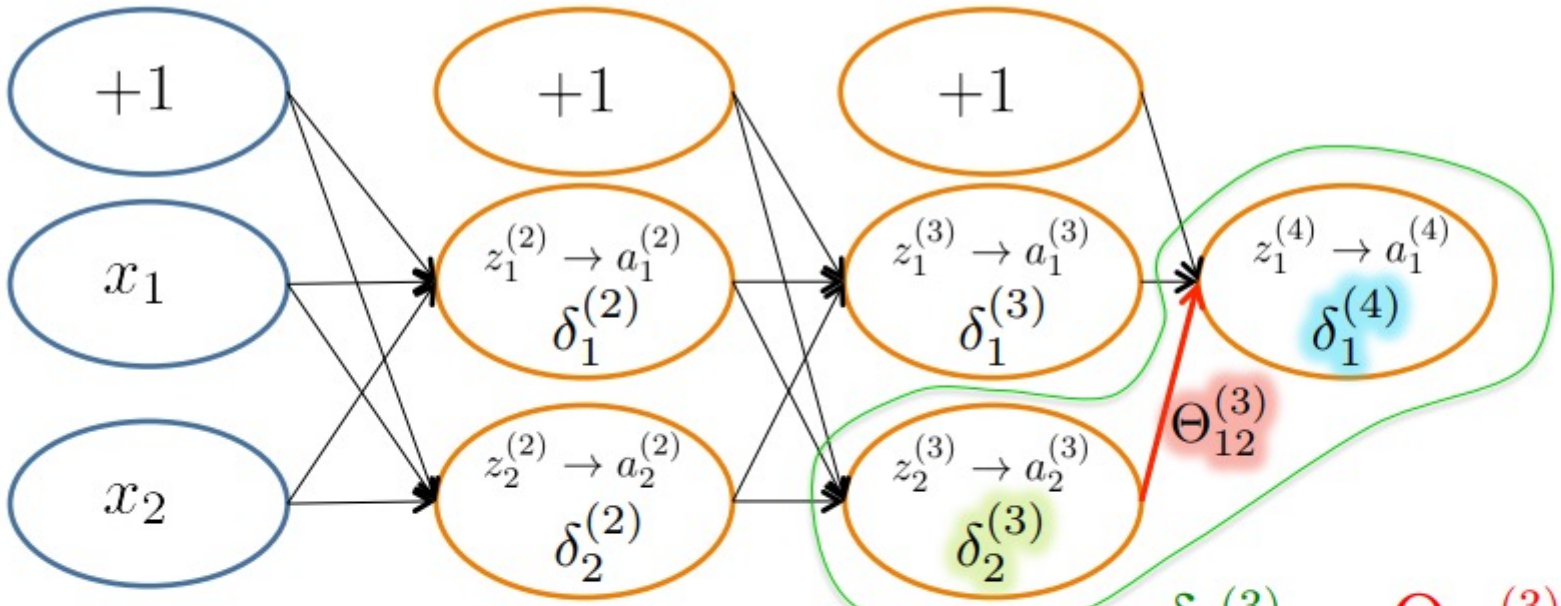
Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where $\text{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Backpropagation Intuition



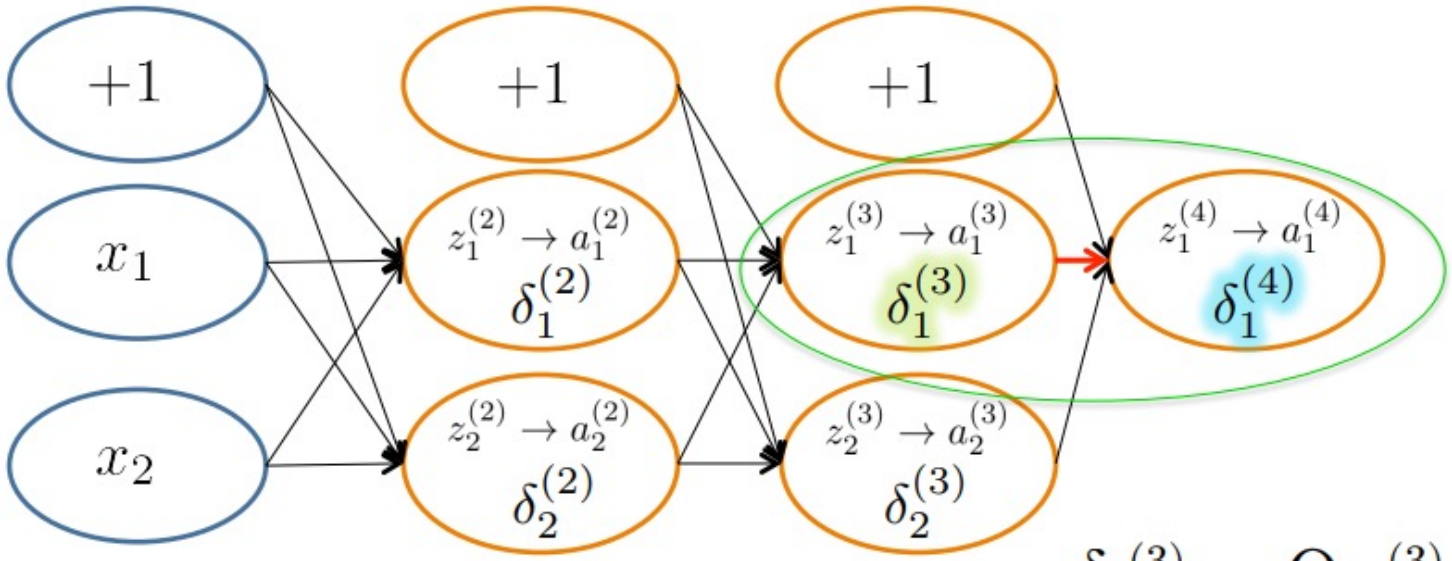
$\delta_j^{(l)}$ = "error" of node j in layer l

Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Backpropagation Intuition



$$\delta_2^{(3)} = \Theta_{12}^{(3)} \times \delta_1^{(4)}$$

$$\delta_1^{(3)} = \Theta_{11}^{(3)} \times \delta_1^{(4)}$$

$\delta_j^{(l)}$ = "error" of node j in layer l

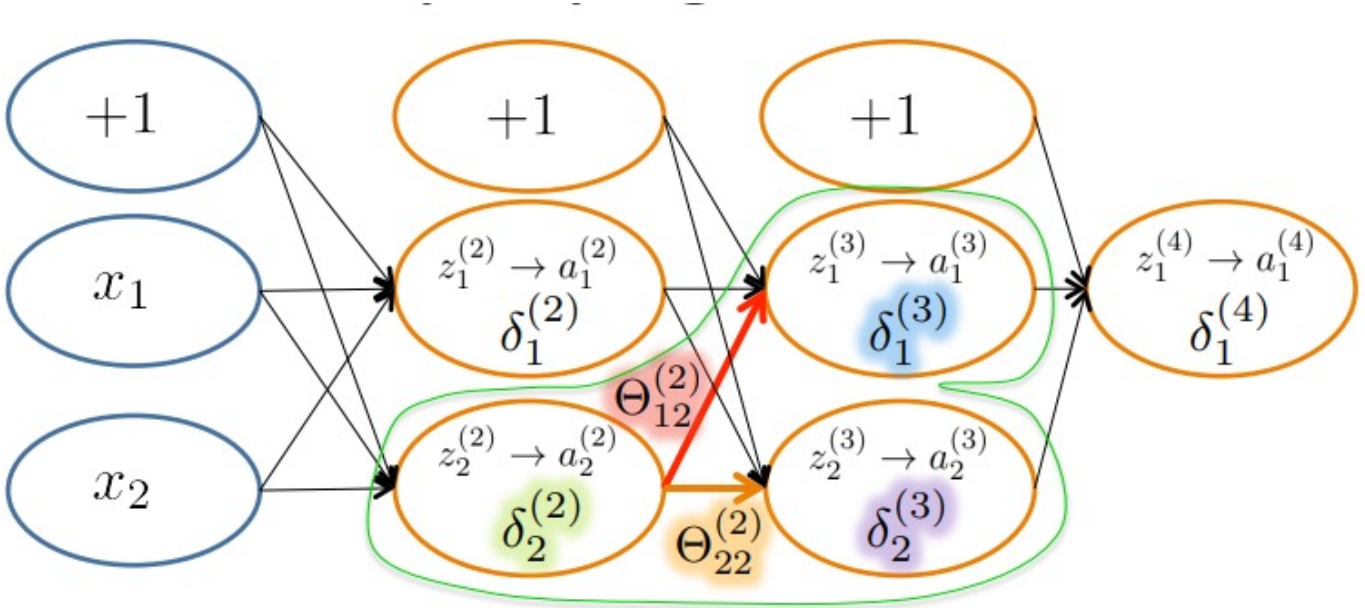
Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where $\text{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Backpropagation Intuition



$$\delta_2^{(2)} = \Theta_{12}^{(2)} \times \delta_1^{(3)} + \Theta_{22}^{(2)} \times \delta_2^{(3)}$$

$\delta_j^{(l)}$ = "error" of node j in layer l

Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

$$\text{where } \text{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$$

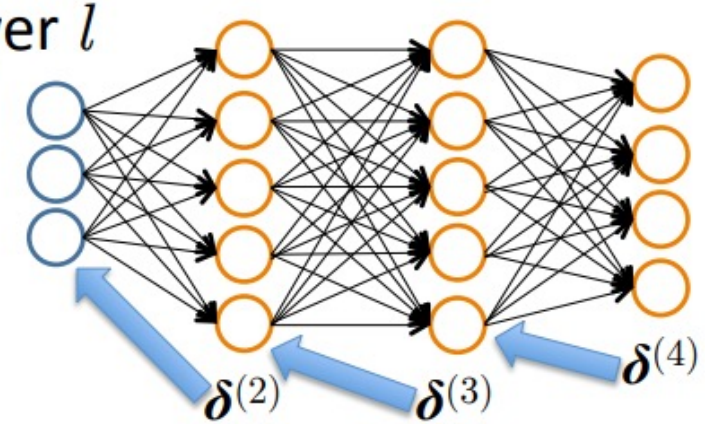
Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Backpropagation: Gradient Computation

Let $\delta_j^{(l)}$ = "error" of node j in layer l

(#layers $L = 4$)



Element-wise product .*

Backpropagation

- $\delta^{(4)} = a^{(4)} - y$
- $\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)}$.* $g'(z^{(3)})$
- $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)}$.* $g'(z^{(2)})$
- (No $\delta^{(1)}$)

$g'(z^{(3)}) = a^{(3)}$.* $(1-a^{(3)})$

$g'(z^{(2)}) = a^{(2)}$.* $(1-a^{(2)})$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$

(ignoring λ; if $\lambda = 0$)

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Backpropagation drawbacks

"Backprop is the cockroach of machine learning. It's ugly, and annoying, but you just can't get rid of it."

-Geoff Hinton

Problems:

- black box
- local minima

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

KS

what are the drawbacks of Backpropagation ?

While backpropagation is a powerful and widely used algorithm for training neural networks, it also has some drawbacks and limitations. Here are a few notable ones:

1. Vanishing and Exploding Gradients: Backpropagation involves propagating the error gradients backward through the layers of a neural network. In deep neural networks with many layers, the gradients can either become very small (vanishing gradients) or very large (exploding gradients) as they propagate backward. This phenomenon can make it difficult for the network to effectively update the weights in the earlier layers, leading to slow convergence or unstable training.
2. Local Minima and Plateaus: The optimization landscape of neural networks can be complex, with many local minima, plateaus, and saddle points. During training, backpropagation aims to find a set of weights that minimizes the error. However, it can sometimes get stuck in suboptimal local minima, preventing the network from achieving the best possible performance.
3. Overfitting: Backpropagation can make neural networks prone to overfitting, where the network becomes overly specialized to the training data and performs poorly on new, unseen data. Overfitting occurs when the network memorizes the training examples instead of learning the underlying patterns. Regularization techniques such as dropout, weight decay, or early stopping are often employed to mitigate overfitting.

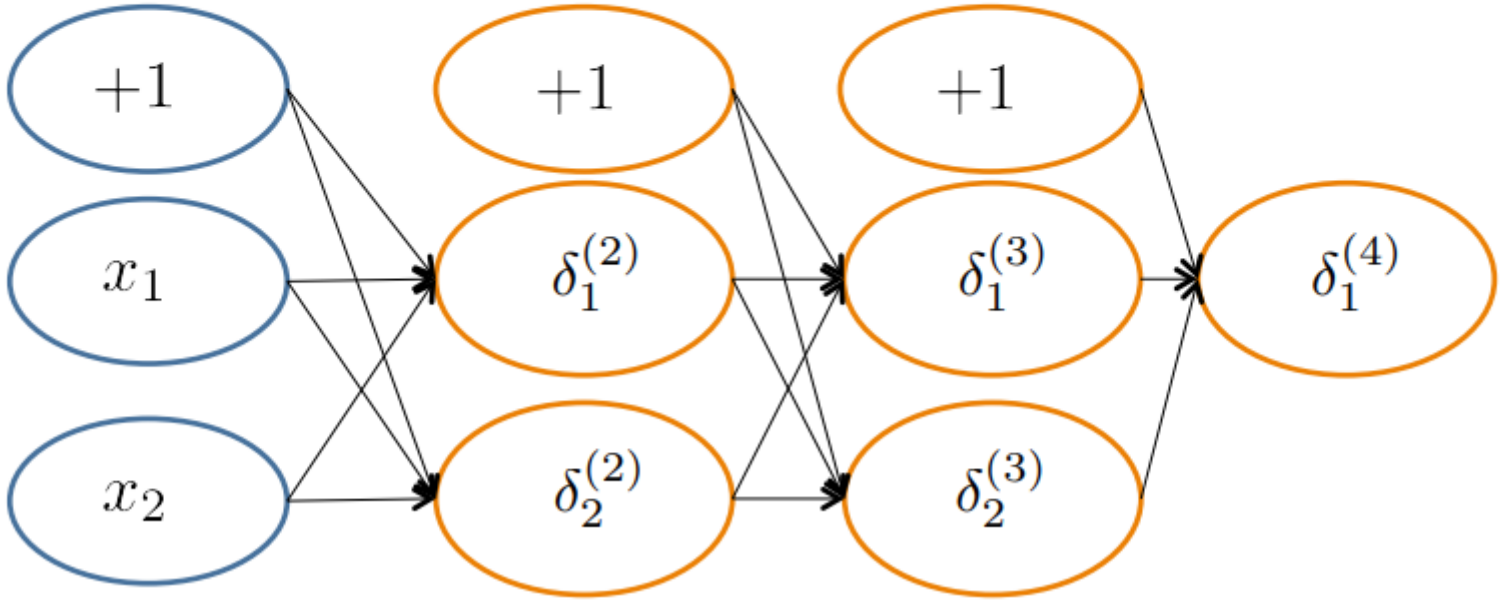
4. Training Time and Resource Intensity: Training deep neural networks with backpropagation can be computationally intensive and time-consuming, especially for large datasets and complex network architectures. The backpropagation algorithm requires multiple iterations over the entire training dataset for each update of the weights, which can be time-prohibitive for large-scale applications.
5. Lack of Robustness to Noisy Data: Backpropagation assumes that the training data is noise-free and that the desired outputs are accurately labeled. However, in real-world scenarios, training data may contain noise, errors, or inconsistencies, which can adversely affect the training process. Noisy or mislabeled data can lead to suboptimal weight updates and affect the network's generalization ability.

Researchers and practitioners have developed various techniques and advancements to address these drawbacks of backpropagation, such as different weight initialization strategies, adaptive learning rate algorithms, normalization techniques, and more advanced optimization algorithms. Additionally, alternative training algorithms like evolutionary algorithms and reinforcement learning have been explored as alternatives to backpropagation for certain types of neural networks or specific problem domains.

Implementation Details

Random Initialization

- Important to randomize initial weight matrices
- Can't have uniform initial weights, as in logistic regression
  - Otherwise, all updates will be identical & the net won't learn



https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Implementation Details

- For convenience, compress all parameters into $\boldsymbol{\theta}$
  - "unroll" $\Theta^{(1)}$, $\Theta^{(2)}$,... , $\Theta^{(L-1)}$ into one long vector $\boldsymbol{\theta}$
    - E.g., if $\Theta^{(1)}$ is 10 x 10, then the first 100 entries of $\boldsymbol{\theta}$ contain the value in $\Theta^{(1)}$
  - Use the `reshape` command to recover the original matrices
    - E.g., if $\Theta^{(1)}$ is 10 x 10, then

```
theta1 = reshape(theta[0:100], (10, 10))
```

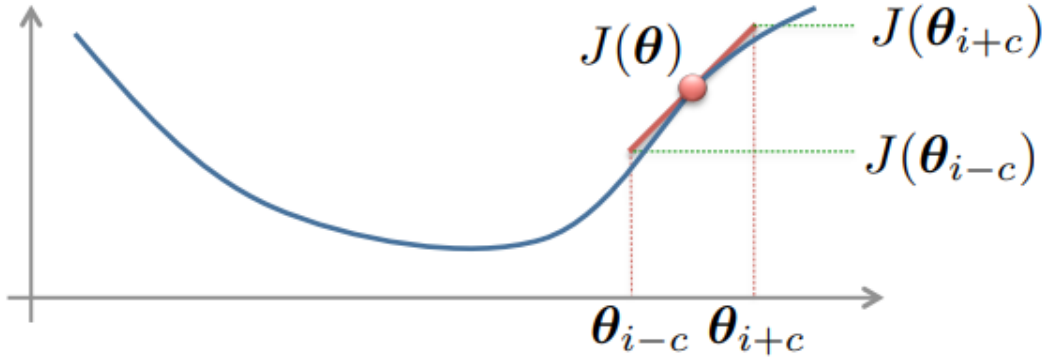
- Each step, check to make sure that $J(\boldsymbol{\theta})$ decreases

- Implement a gradient-checking procedure to ensure that the gradient is correct...

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Gradient Checking

Idea: estimate gradient numerically to verify implementation, then turn off gradient checking



$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{J(\boldsymbol{\theta}_{i+c}) - J(\boldsymbol{\theta}_{i-c})}{2c}$$

$$c \approx 1\text{E-}4$$

$$\boldsymbol{\theta}_{i+c} = [\theta_1, \theta_2, \ldots, \theta_{i-1}, \theta_i + c, \theta_{i+1}, \ldots]$$

Change ONLY the i^{th} entry in $\boldsymbol{\theta}$, increasing (or decreasing) it by c

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Gradient Checking

$\boldsymbol{\theta} \in \mathbb{R}^m$ $\boldsymbol{\theta}$ is an "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \ldots$

$\boldsymbol{\theta} = [\theta_1, \theta_2, \theta_3, \ldots, \theta_m]$

Put in vector called `gradApprox`

$$\frac{\partial}{\partial \theta_1} J(\boldsymbol{\theta}) \approx \frac{J([\theta_1 + c, \theta_2, \theta_3, \ldots, \theta_m]) - J([\theta_1 - c, \theta_2, \theta_3, \ldots, \theta_m])}{2c}$$

$$\frac{\partial}{\partial \theta_2} J(\boldsymbol{\theta}) \approx \frac{J([\theta_1, \theta_2 + c, \theta_3, \ldots, \theta_m]) - J([\theta_1, \theta_2 - c, \theta_3, \ldots, \theta_m])}{2c}$$

$$\vdots$$

$$\frac{\partial}{\partial \theta_m} J(\boldsymbol{\theta}) \approx \frac{J([\theta_1, \theta_2, \theta_3, \ldots, \theta_m + c]) - J([\theta_1, \theta_2, \theta_3, \ldots, \theta_m - c])}{2c}$$

Check that the approximate numerical gradient matches the entries in the D matrices

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Implementation Steps

- Implement backprop to compute `DVec`
  - `DVec` is the unrolled $\{D^{(1)}, D^{(2)}, \ldots\}$ matrices
- Implement numerical gradient checking to compute `gradApprox`
- Make sure `DVec` has similar values to `gradApprox`
- Turn off gradient checking. Using backprop code for learning.

Important: Be sure to disable your gradient checking code before training your classifier.

- If you run the numerical gradient computation on every iteration of gradient descent, your code will be very slow

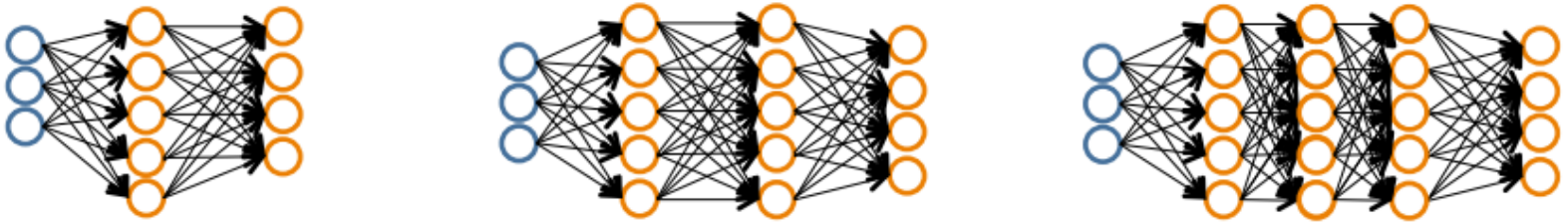
Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Putting It All Together

Training a Neural Network

Pick a network architecture (connectivity pattern between nodes)



- # input units = # of features in dataset
- # output units = # classes

Reasonable default: 1 hidden layer

- or if >1 hidden layer, have same # hidden units in every layer (usually the more the better)

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Training a Neural Network

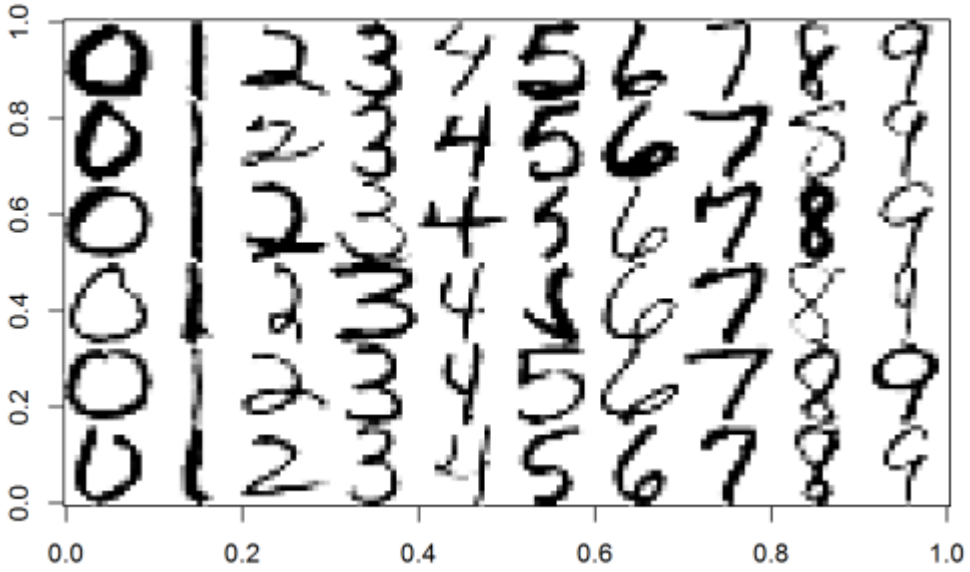
1. Randomly initialize weights
2. Implement forward propagation to get $h_\Theta(\mathbf{x}_i)$ for any instance $\mathbf{x}_i$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$
5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation vs. the numerical gradient estimate.
   - Then, disable gradient checking code
6. Use gradient descent with backprop to fit the network

Based on slide by Andrew Ng

https://www.cis.upenn.edu/~cis5190/fall2014/lectures/11_NeuralNets.pdf

Example

- Classify 16×16 grayscale images into $0 - 9$ numeral digits
- These $16 \times 16 = 256$ pixel values are used as inputs to the neural network classifier. There are 10 target measurements $Y_k, k = 0, 1, \cdots, 9$, each being coded as a $0/1$ variable for digit k.



demo

Applied Multivariate Statistical Analysis (6th Edition) by Richard A. Johnson , Dean W.Wichern

Homework 10 (submitted to e3.nycu.edu.tw before Dec 20, 2023)

Use R, Python, and suitable computer packages to perform Neural Networks.

Possible sources of open datasets:

- UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets.php)
- Kaggle Datasets (https://www.kaggle.com/datasets)
- World Health Organization Datasets (https://www.who.int/)