Applied Machine Learning

Deep Learning Advance

Computer Science, Fall 2022

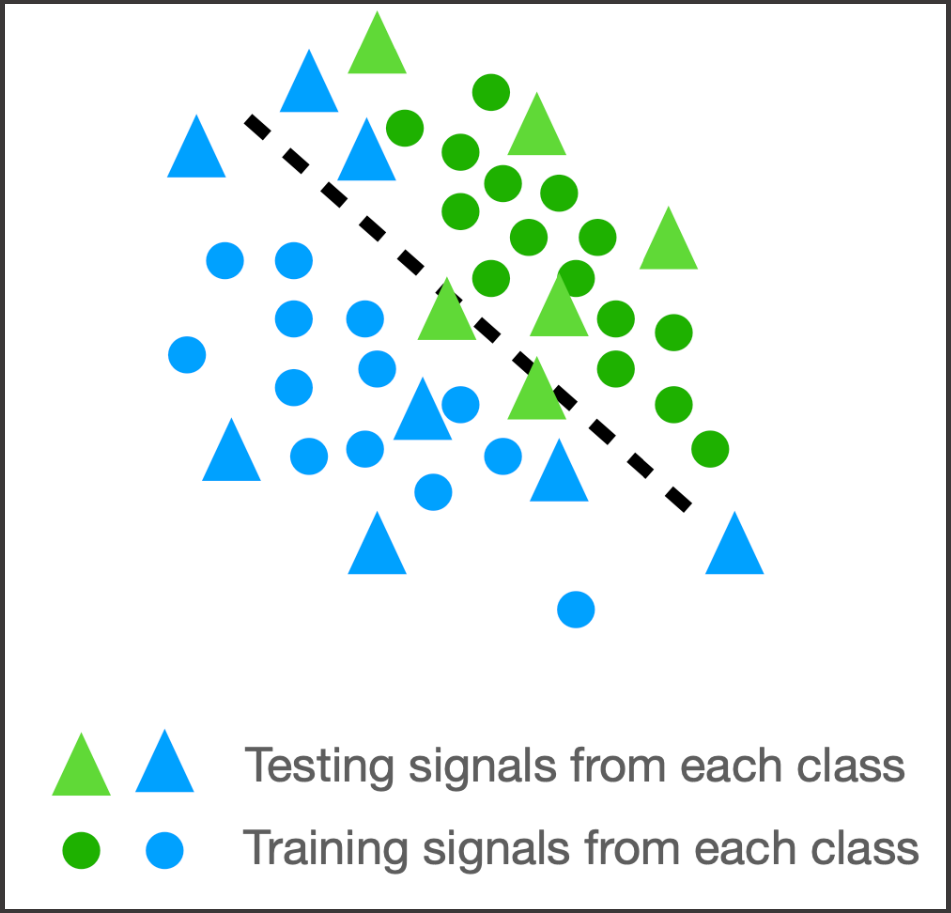
Instructor: Xuhong Zhang

Learning Objectives

- Understand Multi-layer Perceptrons (MLP)
- Understand Neural Networks

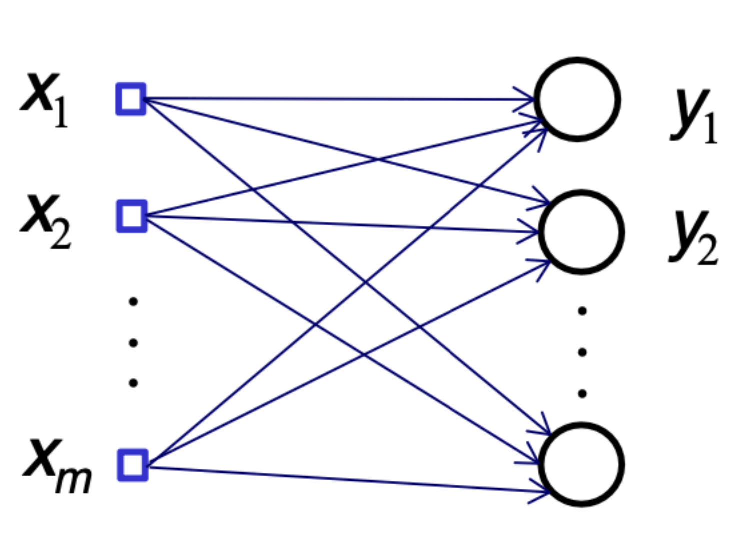
Generalization Once Trained

- Perceptrons during testing
  - Perceptrons generalize by deriving a decision boundary in the input space
  - The selection of training patterns is thus important for generalization
  - The solution weight vector is not unique. There are infinite possible solutions and decision boundaries.



Perceptrons for Multi-Dimensional Outputs

- Multiple perceptrons can be used when performing multi-class classification (one for each class) or multi-dimensional estimation
  - ***Classification***: Perceptron output is 1 when input is from the corresponding class. Its output is 0 otherwise. Each perceptron forms its own decision boundary.
  - ***Regression***: Each perception corresponds to one of the output dimensions

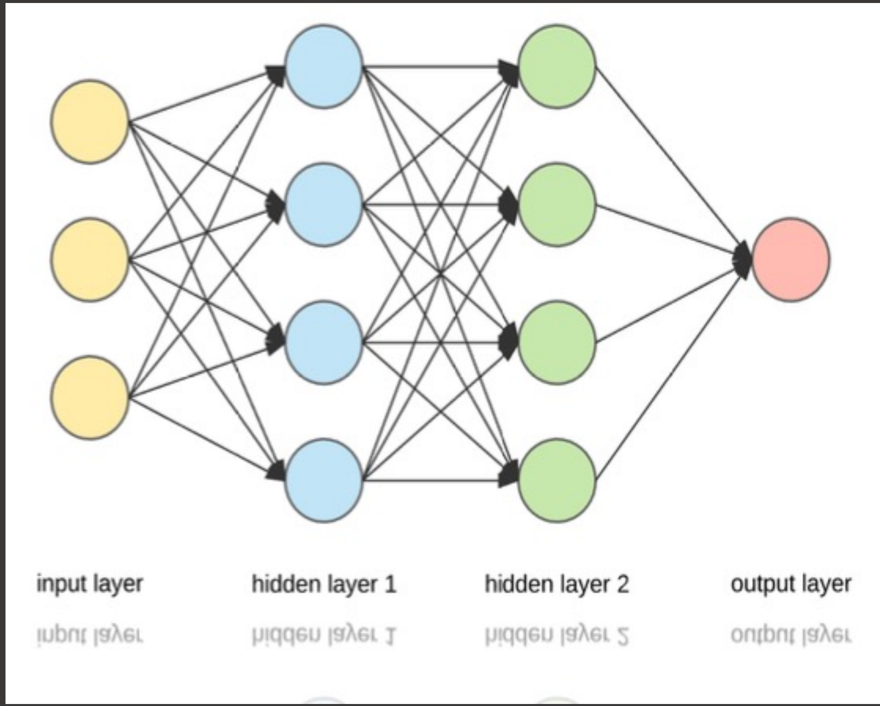


Perceptrons for Multi-Dimensional Outputs

- When these perceptrons have the same inputs (but with different weights), this stacking of perceptrons is called ***a layer***

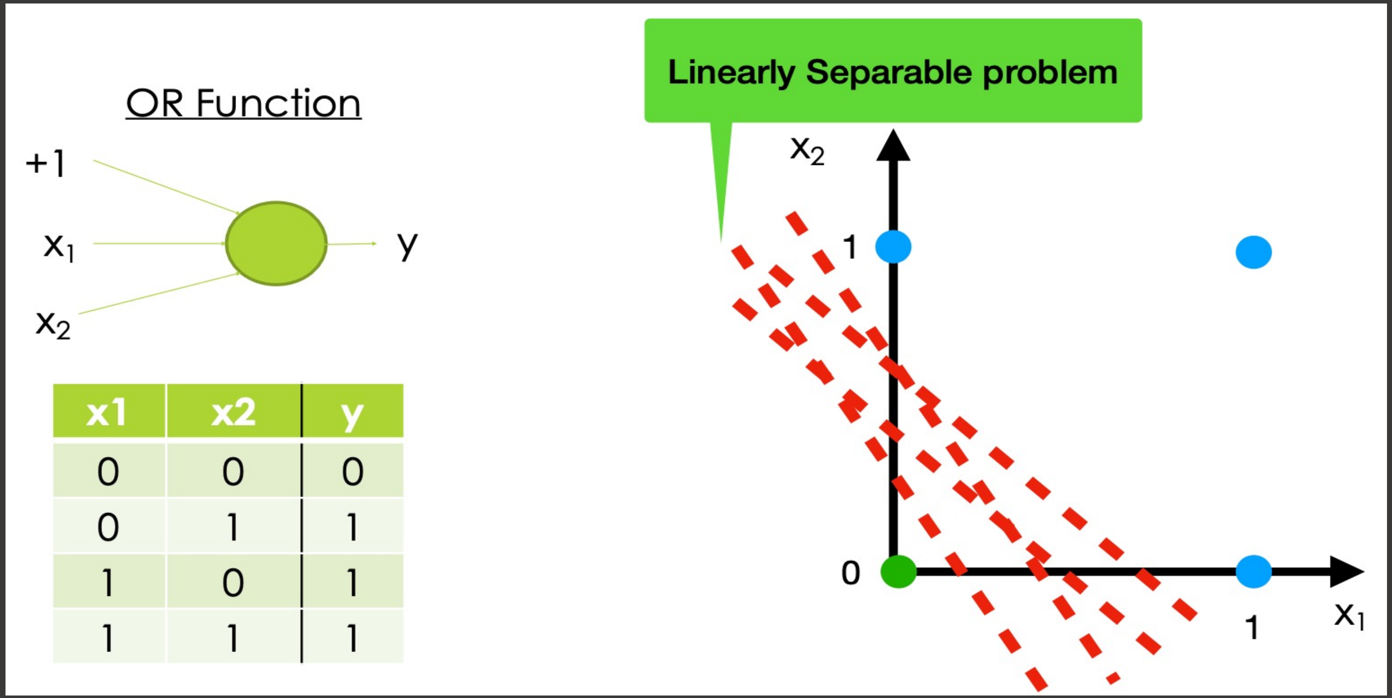
- When the outputs of one layer become the input to another, we call this a ***multi-layer perceptron (MLP)*** or ***neural network***.

- ***Deep neural networks*** (typically) have three or more layers.



A Limitation of Single-layer Perceptron

- Logic gate example: OR Gate
  - Single-layer perceptron networks can be used to solve linearly-separable problems.

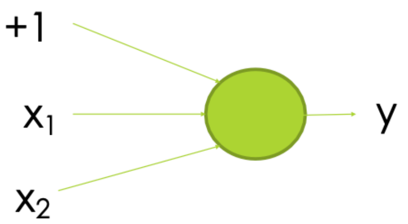


x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

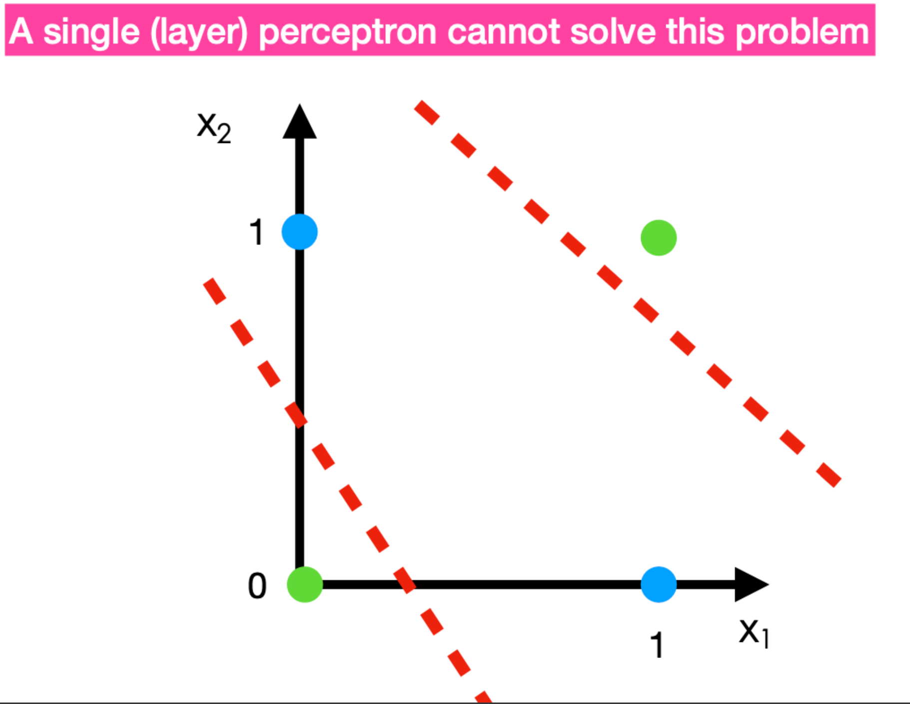
A Limitation of Single-layer Perceptron

- Logic gate example: XOR Gate
  - Single-layer perceptron networks, however, are not as useful for problems that are linearly inseparable or linearly-separable problems that require multiple boundaries per class

XOR Function



x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

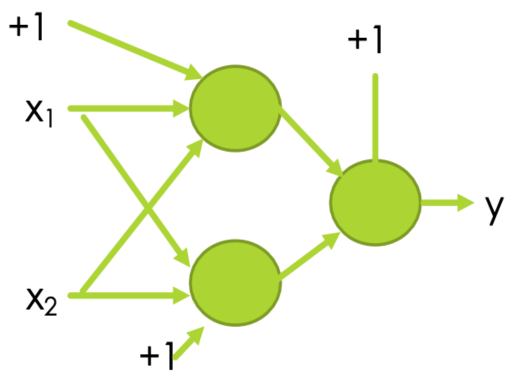


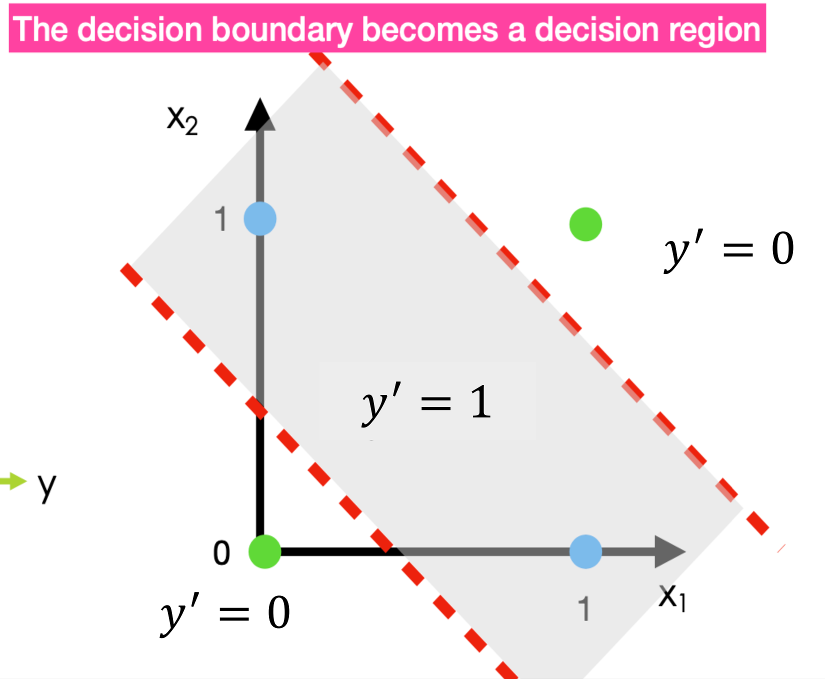
The Case for Multi-Layered Perceptrons (MLP)

- Logic gate example
  - Adding layers to a network can help solve more complicated problems. The resulting neural network is called a ***Multi-Layer Perceptron (MLP)***

XOR Function

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0





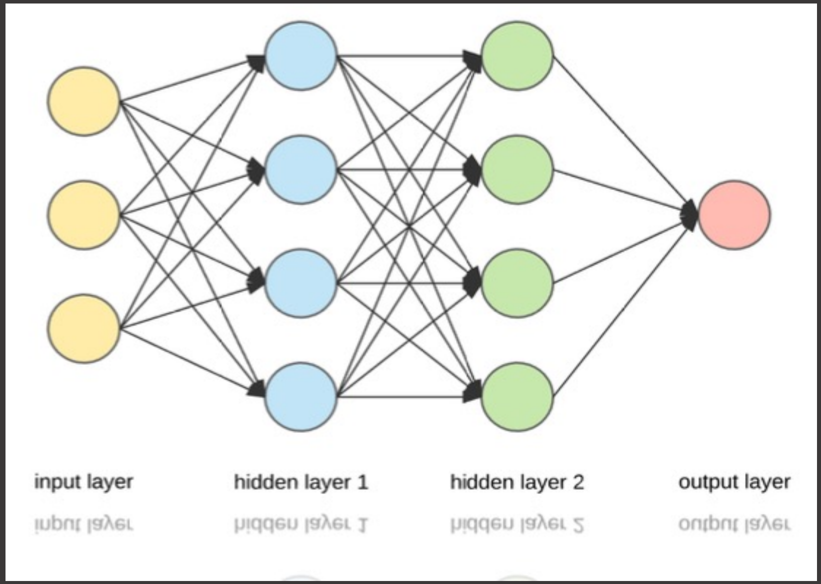
Multi-Layer Perceptron (MLP)

- ***Input layer***: not really a layer, but serves to represent the inputs to the network

- ***Hidden layers***: layers where the output goes to another layer of neurons

- ***Output layers***: final layer of network, where final output(s) are computed

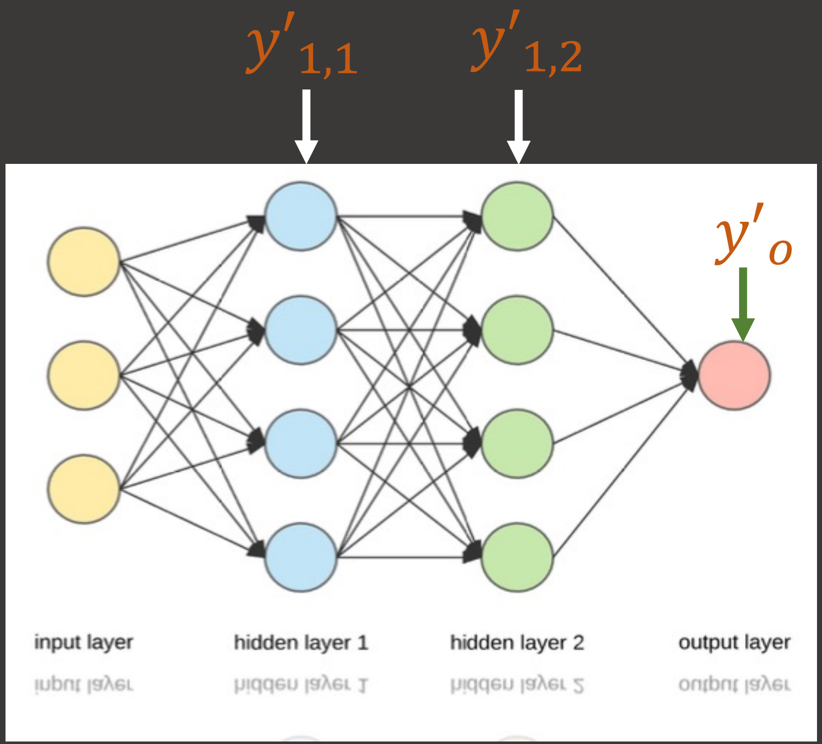
- **Deep neural networks** (DNN) have two or More hidden layers



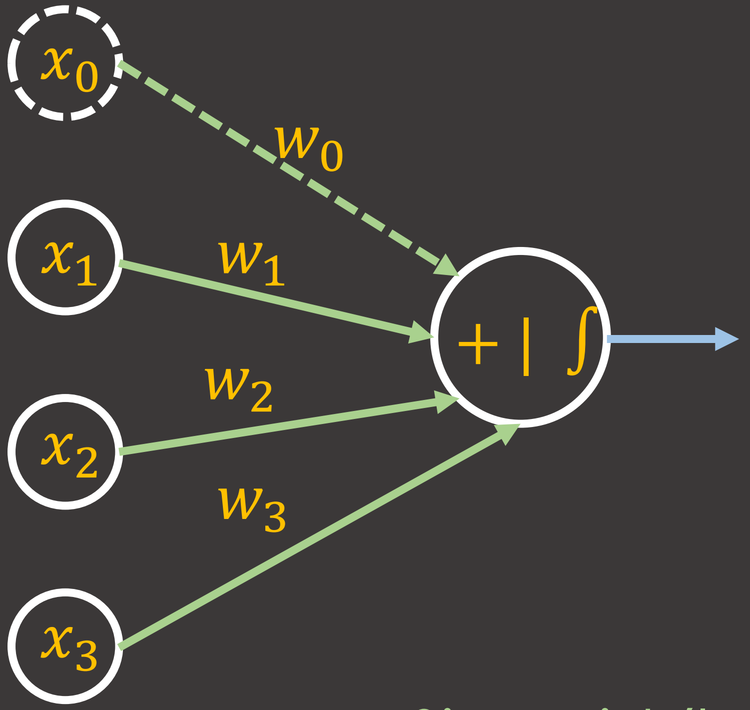
Computing Outputs: Forward Pass

- The output(s) of the network is(are) computed in a layer-wise fashion
- The output from layer one is first computed and this becomes the input to the next layer
- This continues until the output(s) is(are) computed:

$$y'_{1,1} = \phi\left(w_{1,1}^{T} x\right)$$
$$y'_{1,2} = \phi\left(w_{1,2}^{T} y'_{1}\right)$$
$$y'_{o} = \phi\left(w_{o}^{T} y'_{2}\right)$$



Neuron Model: Logistic Unit



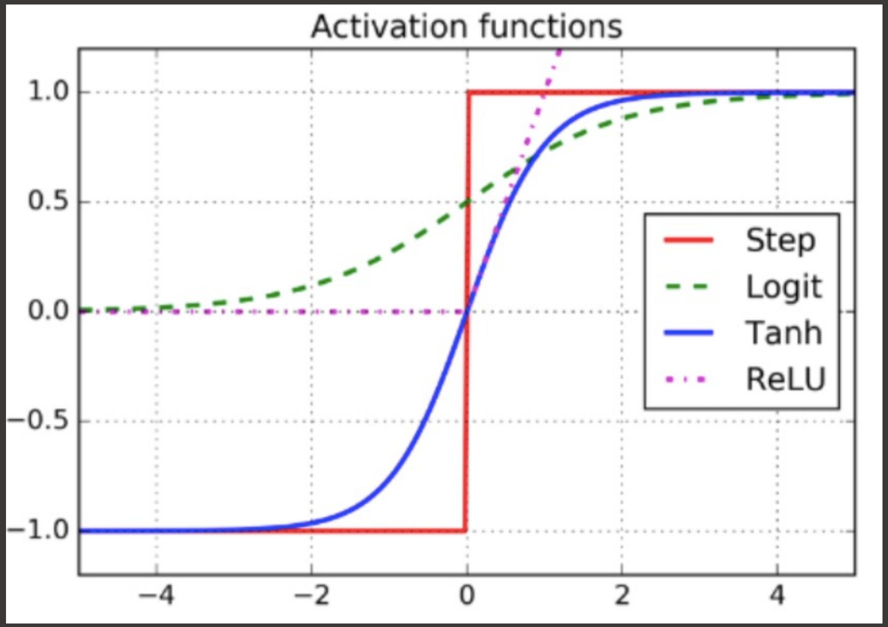
$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$h_w(x) = g(w^T x) = \frac{1}{1 + e^{-w^T x}}$$

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

Activation Functions

- The sign (or step) function of MLPs is mostly useful for classification. It also is not as useful for more complicated problems.
- To address these problems, other activation functions are often used in DNNs. These activation Functions all use the activation potential as originally defined, as their input
  - ***Sigmoid (or logit):*** $\phi(x) = 1/(1 + e^{x})$
  - ***Hyperbolic Tangent (tanh):*** $\phi(x) = \frac{2}{1+e^{-2x}} - 1$
  - ***Rectified Linear (ReLU):*** $\phi(x) = \max(0, x)$
  - ***Linear:*** $\phi(x) = x$
- Different activation functions may be used in different layers (not required)

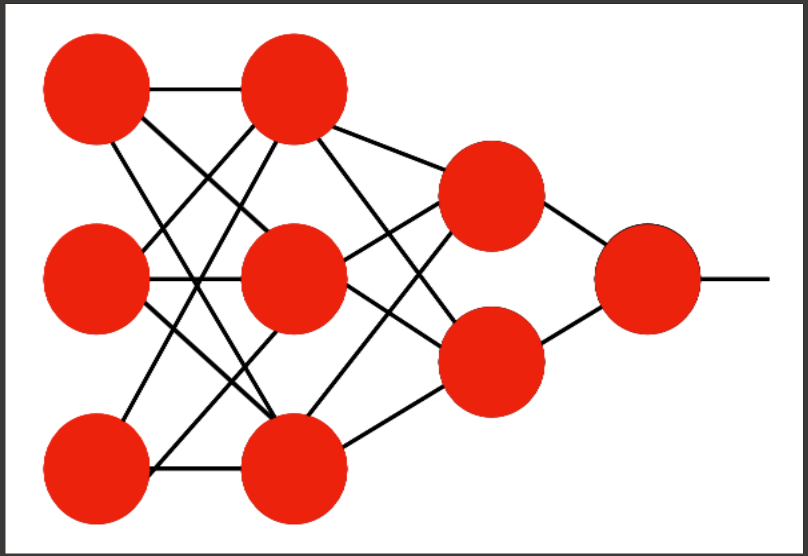


Network Weight Learning

- DNNs are ***supervised*** learning algorithms
- The goal is to find the weights that ***minimize the error*** (often MSE) between the true and predicted values for the training set
- During the training phase, the ***back propagation*** algorithm is used to find these weights
- General idea of back propagation
  - For each training sample (e.g. input), compute the output using the forward pass
  - Compute the estimation/prediction error
  - Sequentially go through each layer (back order) to measure the error contribution from each connection
  - Update the weights based on the error contribution to reduce the error (e.g. ***gradient descent***)

Back Propagation

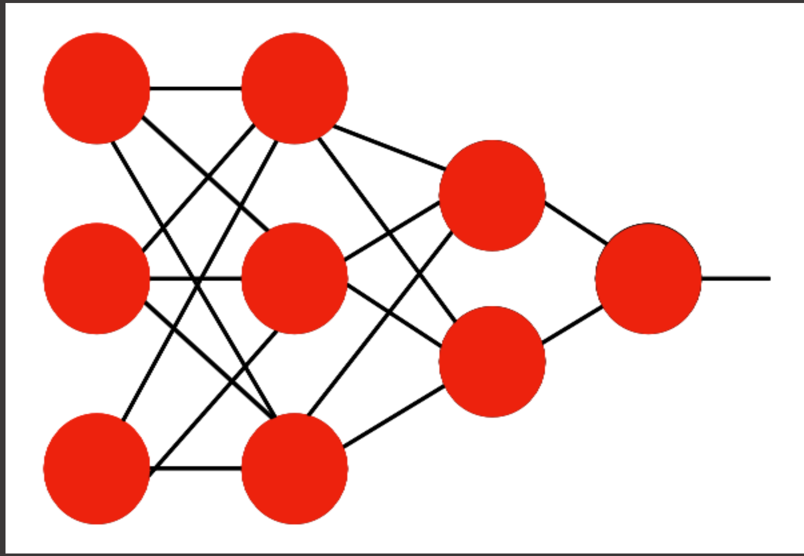
- General idea of back propagation
  - For each training sample (e.g. input), compute the output using the forward pass
  - Compute the estimation/prediction error
  - Sequentially go through each layer (back order) to measure the error contribution from each connection
  - Update the weights based on the error contribution to reduce the error (e.g. gradient descent)



Compute forward pass of DNN to get output y' from input

Back Propagation

- General idea of back propagation
  - For each training sample (e.g. input), compute the output using the forward pass
  - Compute the estimation/prediction error
  - Sequentially go through each layer (back order) to measure the error contribution from each connection
  - Update the weights based on the error contribution to reduce the error (e.g. gradient descent)

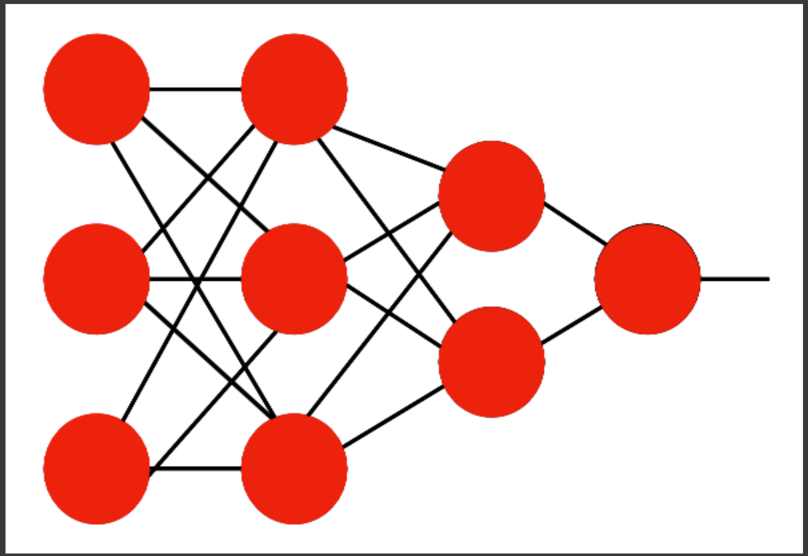


Compute output error

$y - y'$

Back Propagation

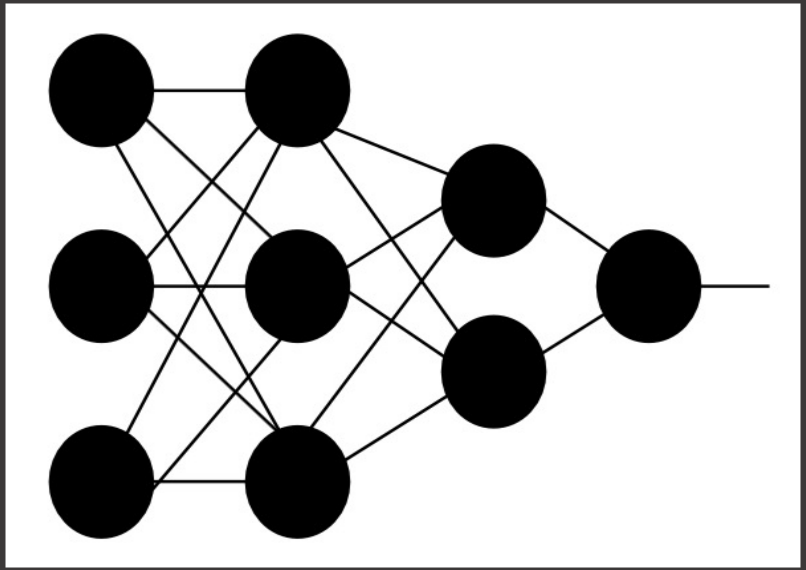
- General idea of back propagation
  - For each training sample (e.g. input), compute the output using the forward pass
  - Compute the estimation/prediction error
  - Sequentially go through each layer (back order) to measure the error contribution from each connection
  - Update the weights based on the error contribution to reduce the error (e.g. gradient descent)



Compute contribution from each neuron in each layer δ_k

Back Propagation

- General idea of back propagation
  - For each training sample (e.g. input), compute the output using the forward pass
  - Compute the estimation/prediction error
  - Sequentially go through each layer (back order) to measure the error contribution from each connection
  - Update the weights based on the error contribution to reduce the error (e.g. gradient descent)



Update weights based
On error and (weighted)
Contributions for each
Neuron and layer

$$w_k = w_k - \eta \nabla E(w)$$

Perceptron Learning Rule vs. Gradient Descent

- ***Perceptron learning*** is for McCulloch-Pitts neurons (with real inputs), which are nonlinear
- ***Perceptron learning*** is for classification
- ***Gradient decent*** is for linear updates (e.g. linear regression)
- Linear regression learning, via ***gradient descent***, is for estimation (or regression)

Perceptron Learning

- $v = \sum_{i=1}^{m} w_i x_i + b$
- $\phi(v) = \begin{cases} 1, & if\ v \geq 0 \\ -1, & if\ v < 0 \end{cases}$
- $y = \phi(v)$

Linear Regression

- $y = \sum_{i=1}^{m} w_i x_i + b$

Nonlinear Regression

- $y = \sum_{i=1}^{m} w_i \phi(x_i) + b$

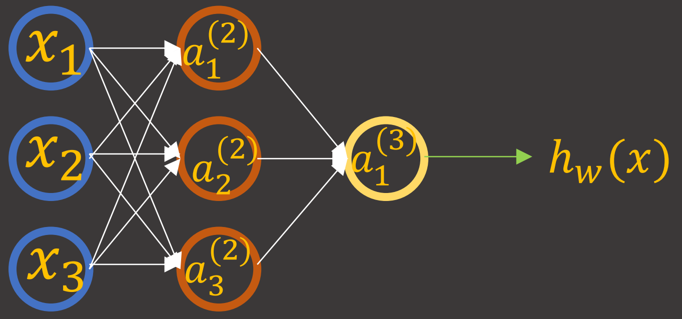
Neural Networks

- 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

Neural Networks

- Key components for the prosperity of NN:
  - ReLu
  - SGD
  - GPUs

Neural Networks



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

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

$$a_1^{(2)} = g(w_{10}^{(1)}x_0 + w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(w_{20}^{(1)}x_0 + w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3)$$

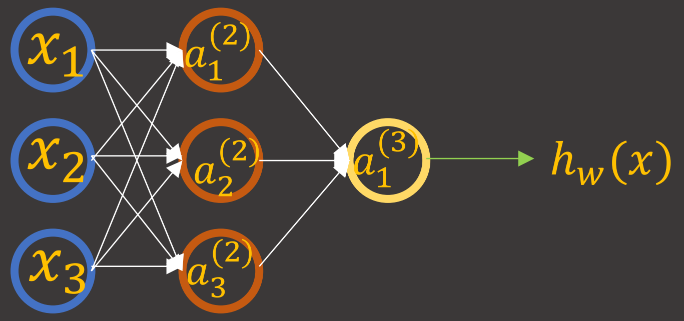
$$a_3^{(2)} = g(w_{30}^{(1)}x_0 + w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3)$$

$$h_w(x) = a_1^{(3)} = g(w_{10}^{(2)}a_0^{(2)} + w_{11}^{(2)}a_1^{(2)} + w_{12}^{(2)}a_2^{(2)} + w_{13}^{(2)}a_3^{(2)})$$

If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $w^{(j)}$ has dimension $s_{j+1} \times (s_j + 1)$: $w^{(1)} \in \mathbb{R}^{3\times 4}, w^{(2)} \in \mathbb{R}^{1\times 4}$

Vectorization

- $a_1^{(2)} = g\left(w_{10}^{(1)} x_0 + w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3\right) = g(z_1^{(2)})$
- $a_2^{(2)} = g\left(w_{20}^{(1)} x_0 + w_{21}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{23}^{(1)} x_3\right) = g(z_2^{(2)})$
- $a_3^{(2)} = g\left(w_{30}^{(1)} x_0 + w_{31}^{(1)} x_1 + w_{32}^{(1)} x_2 + w_{33}^{(1)} x_3\right) = g(z_3^{(2)})$
- $h_w(x) = g\left(w_{10}^{(2)} a_0^{(2)} + w_{11}^{(2)} a_1^{(2)} + w_{12}^{(2)} a_2^{(2)} + w_{13}^{(2)} a_3^{(2)}\right) = g(z_1^{(3)})$



Feed-Forward Steps:

$z^{(2)} = w^{(1)} x$

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

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

$z^{(3)} = w^{(2)} a^{(2)}$

$h_w(x) = a^{(3)} = g(z^{(3)})$

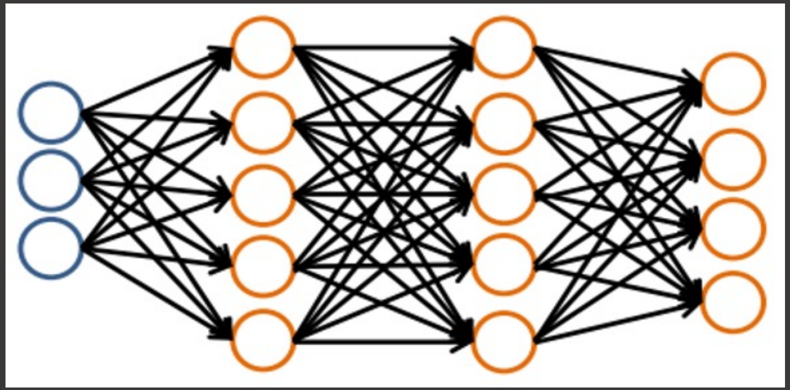
Multiple Output Units: One vs. Rest

Pedestrian

Car

Motorcycle

Truck



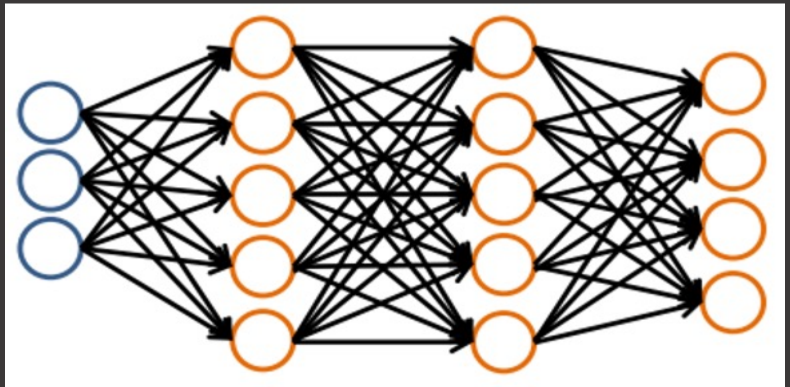
$h_w(x) \in \mathbb{R}^K$

What we want:

$h_w(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ when pedestrian; $h_w(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$ when car

$h_w(x) \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$ when motorcycle; $h_w(x) \approx \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$ when truck

Multiple Output Units: One vs. Rest



$h_w(x) \in \mathbb{R}^K$

What we want:

$h_w(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ when pedestrian; $h_w(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ when car

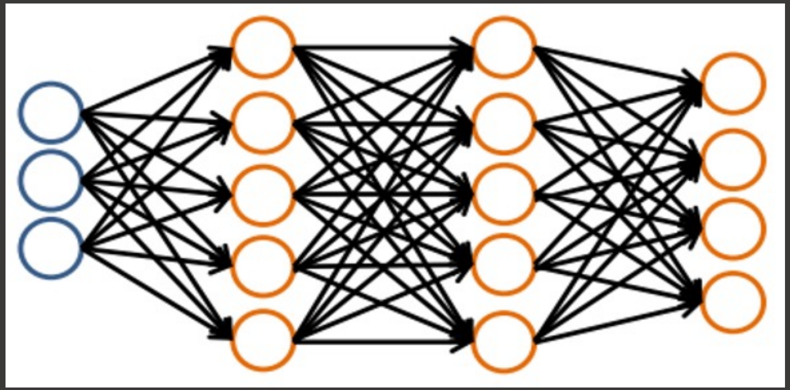
$h_w(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when motorcycle; $h_w(x) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ when truck

---

Given the training data $\{(x_1, y_1), (x_2, y_2), \ \ldots\ , \ (x_n, y_n)\}$, must convert labels to 1-of-K representation:

e.g. $y_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when motorcycle, $y_i = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ when car, etc.

Multiple Output Units: One vs. Rest



Given the training data $\{(x_1, y_1), (x_2, y_2),$ … , $(x_n, y_n)\}$

<u>Binary Classification</u>

$y = 0$ or 1

1 output unit

<u>Multi-class Classification (K classes)</u>

$y \in \mathbb{R}^K$

e.g., $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

K output units

Understanding Presentations

- ***Data representation*** plays a crucial role on the performance of NN, especially for the applications of NNs in a real world.

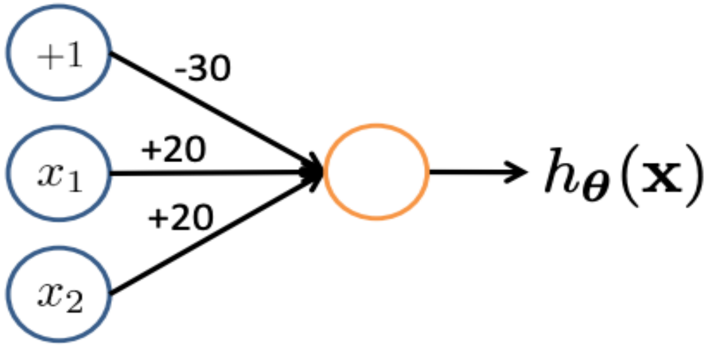
- *The structure, final prediction task, the input data* together decide what representation the network is looking for.

Representing Boolean Functions

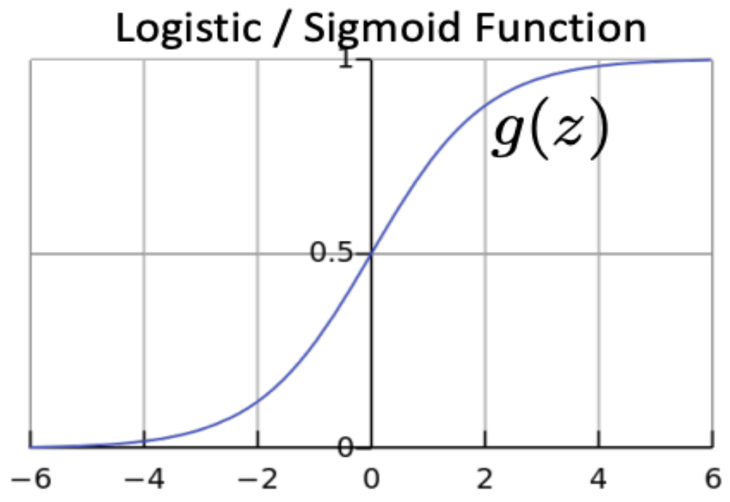
Simple example: AND

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

$y = x_1 \text{ AND } x_2$



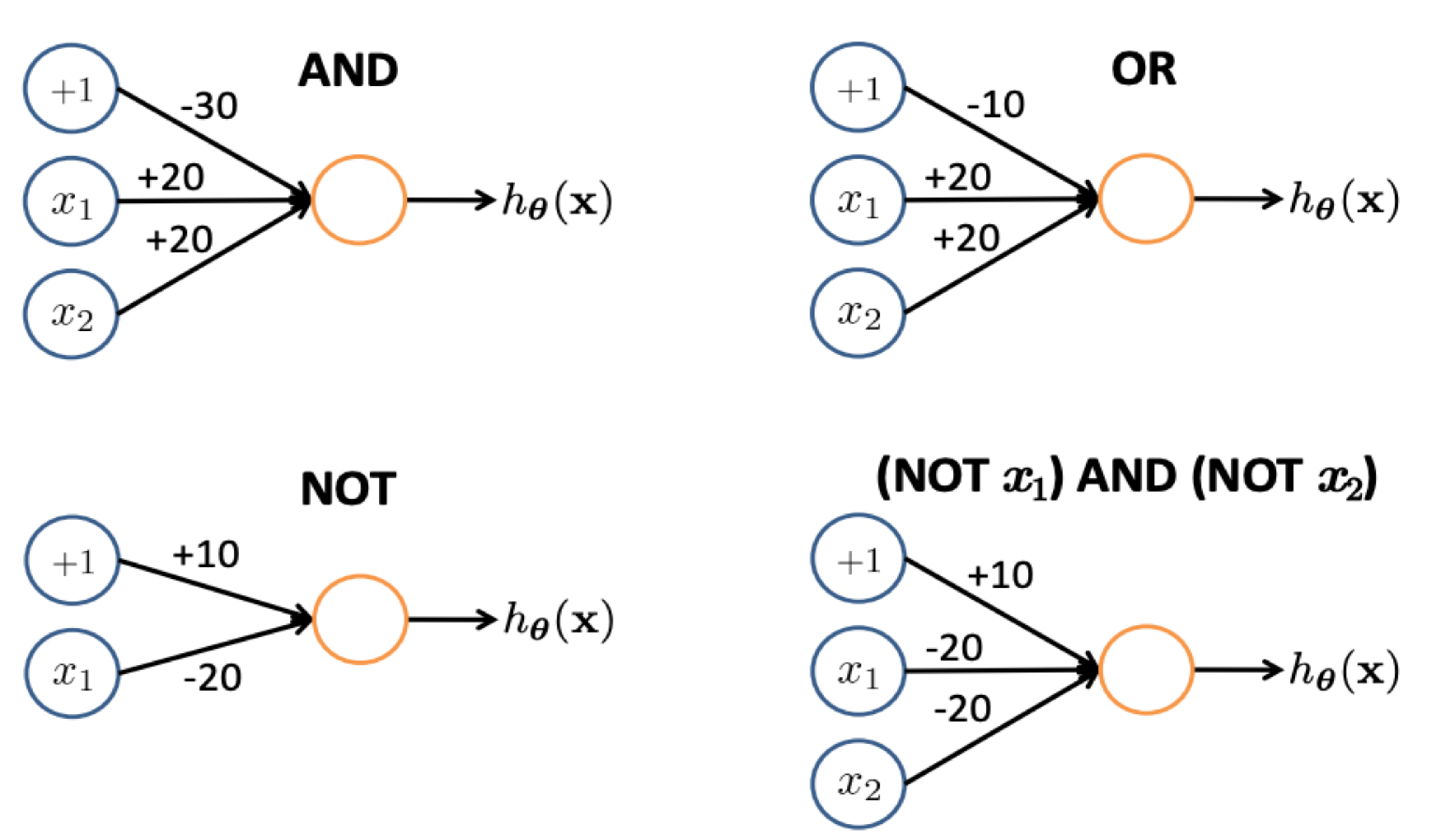
$$h_\Theta(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$



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

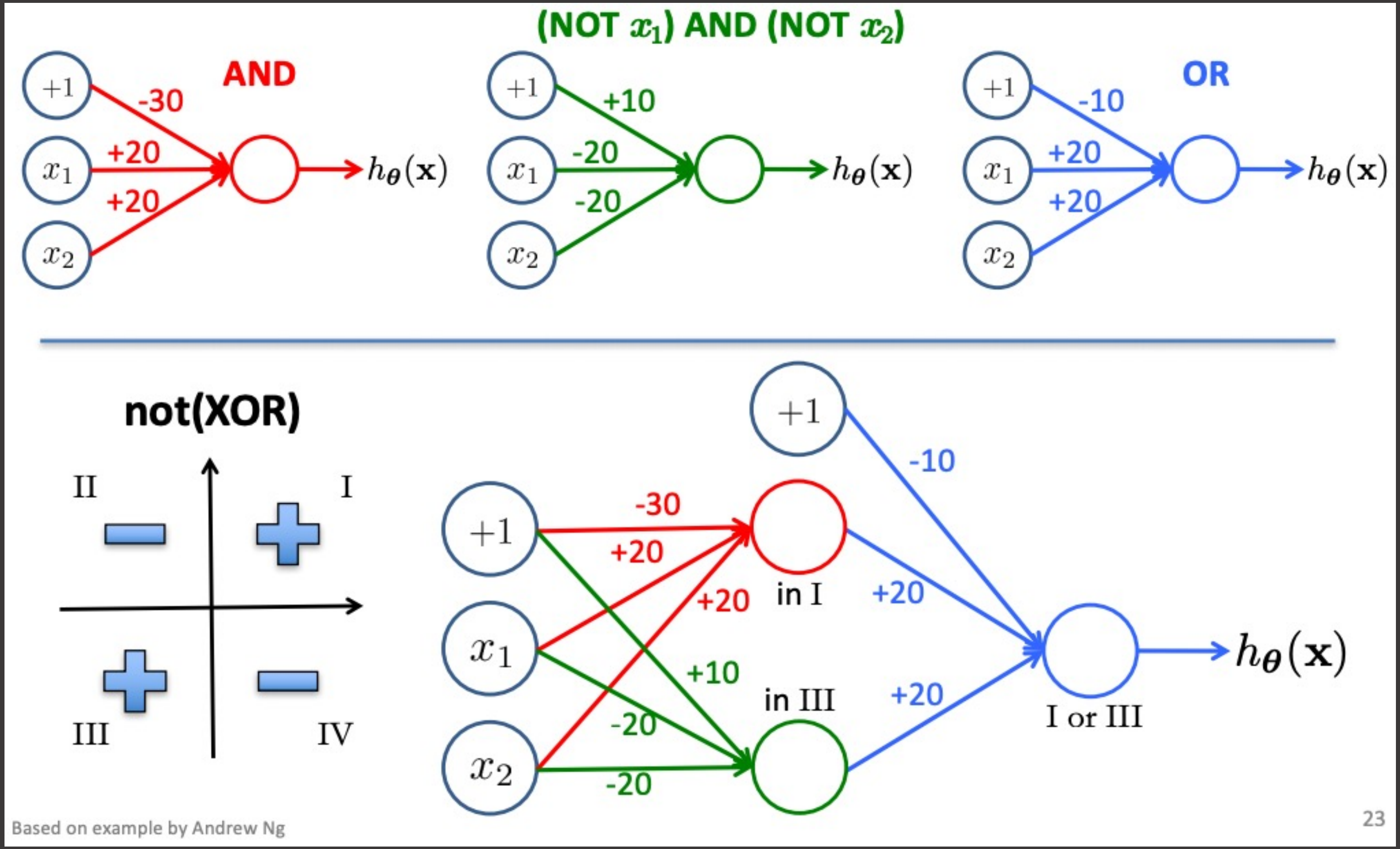
Slide Credit: Andrew Ng

Representing Boolean Functions



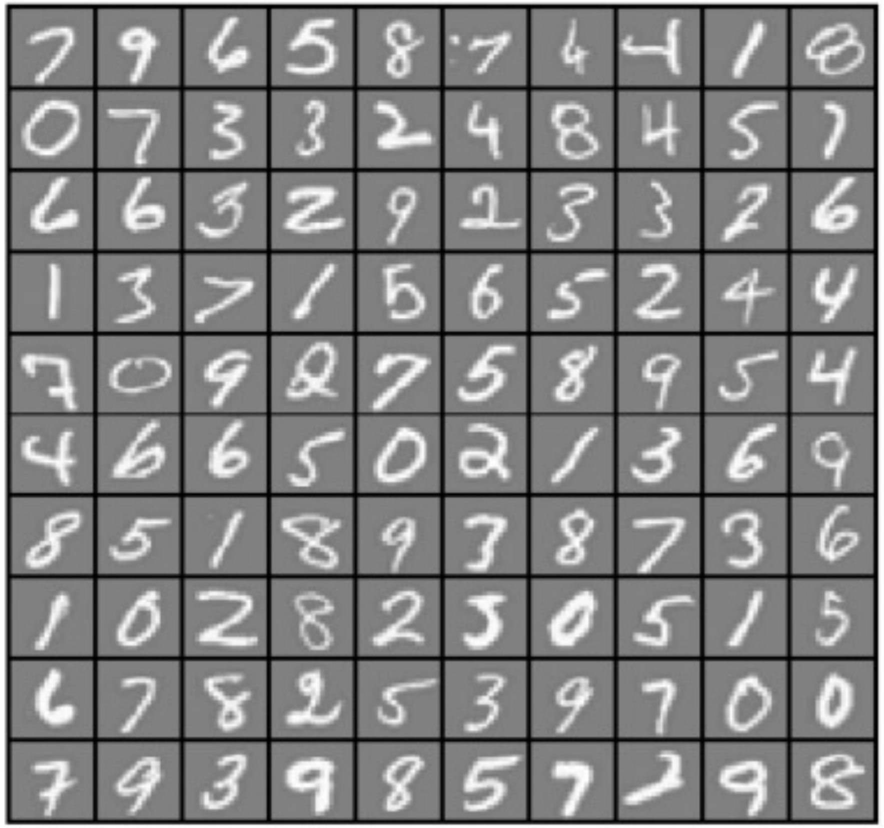
Slide Credit: Andrew Ng

Combining Representations to Create Non-Linear Functions

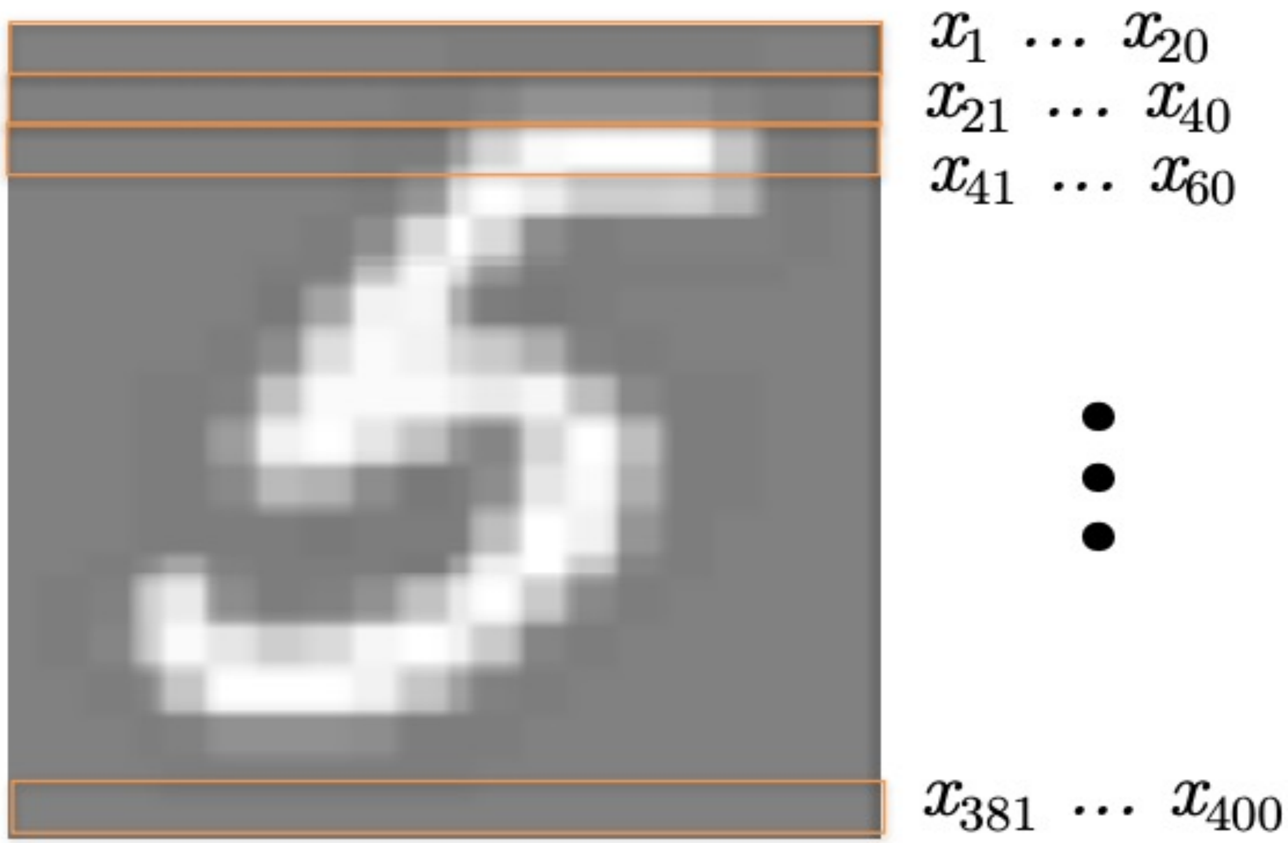


Slide Credit: Andrew Ng

Layering Representations of Images

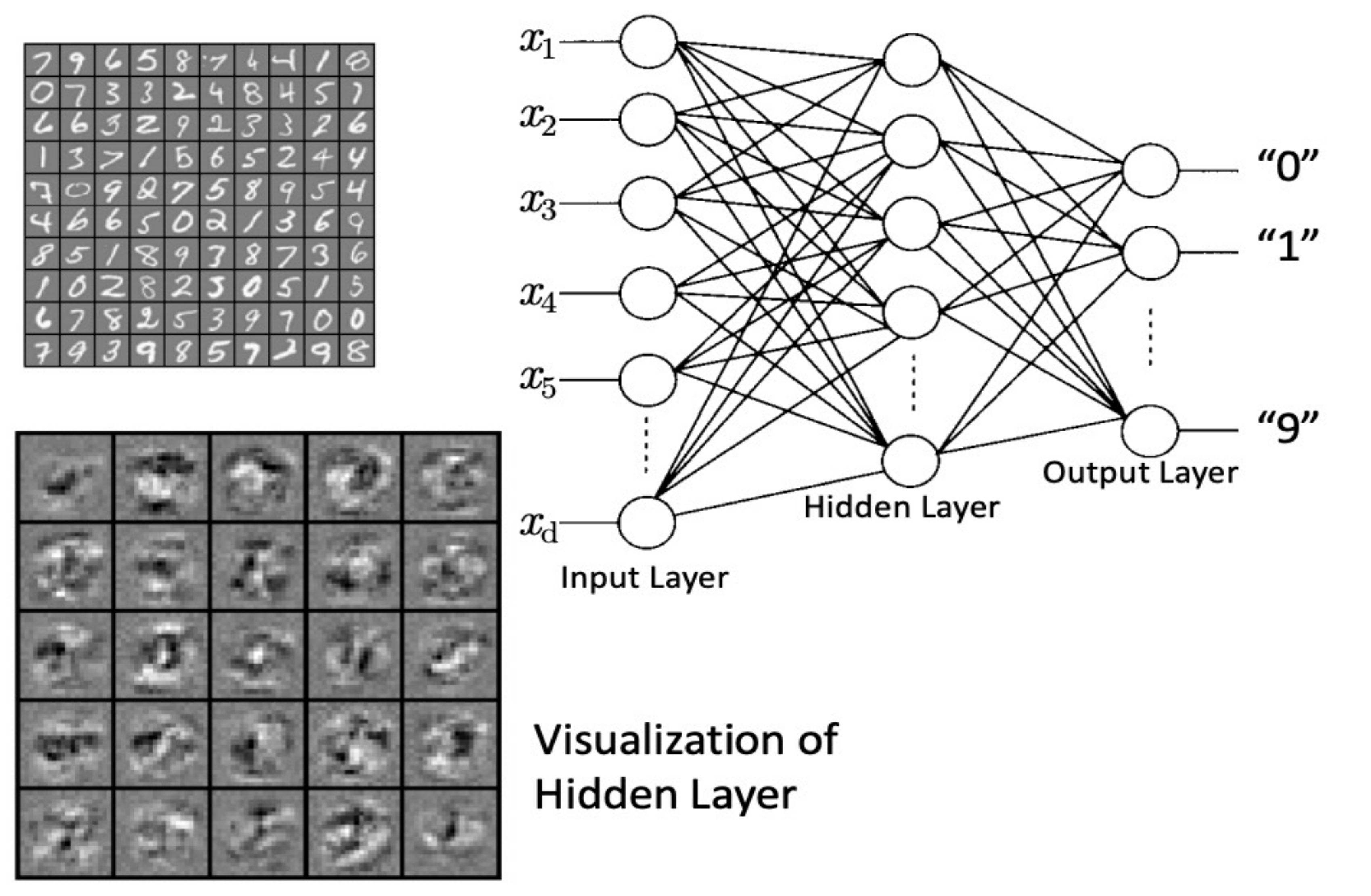


20×20 pixel image
$d = 400$, 10 classes



Each image is "unrolled" into a vector of intensities

Layering Representations of Images

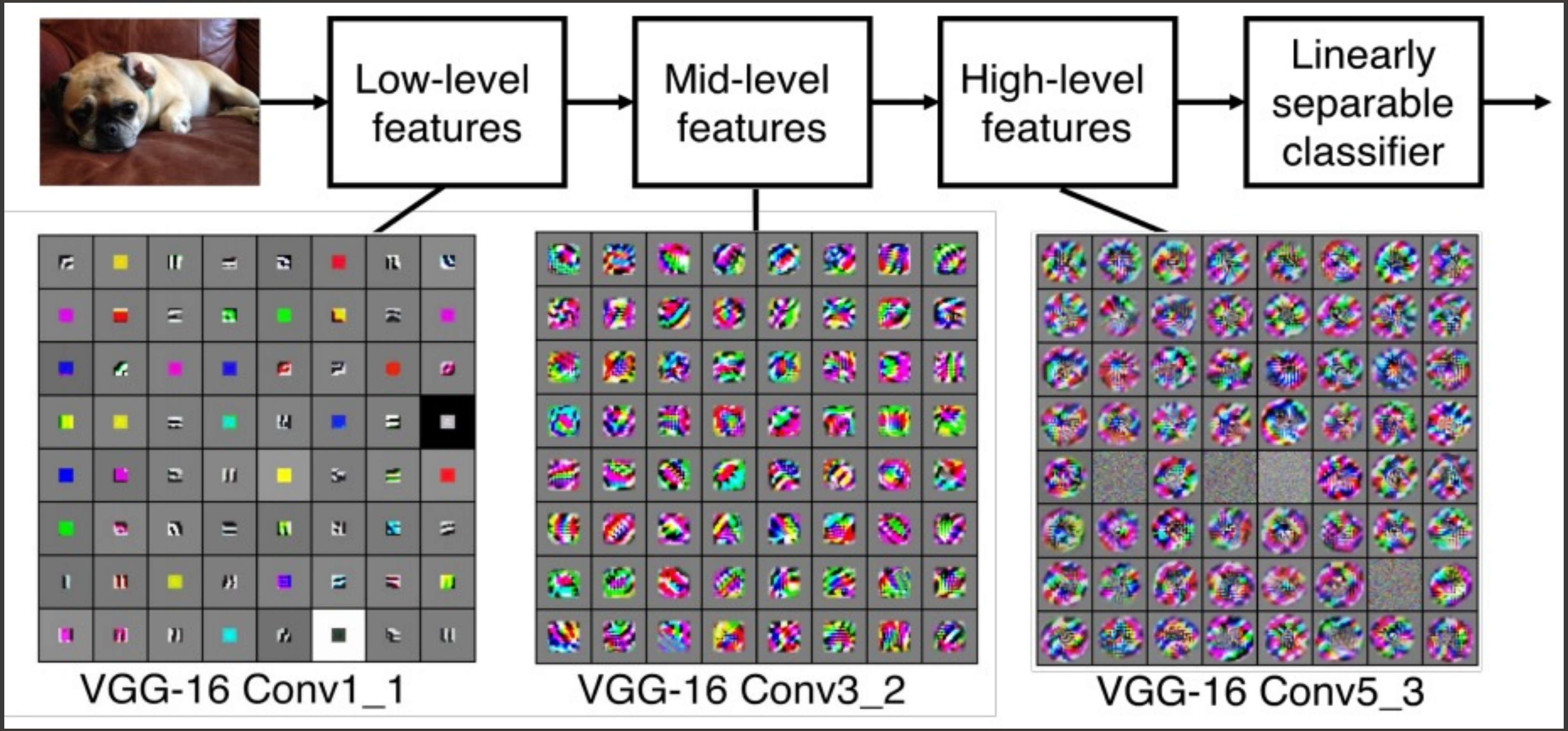


$$\Phi(x) = \begin{bmatrix} h_1(x) \\ \vdots \\ h_m(x) \end{bmatrix}$$

- Neural network learns Φ.
- Each $h_i(x)$ is a linear classifier.
- In digit classification, these classifiers detect vertical edges, round shapes, horiztonals.
- Their output then becomes the input to the main linear classifier.

Layering Representations of Images

Examples of visualized weights for the first layer of a neural network.



"Visualization of VGG-16" by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014]

Demo

- http://yann.lecun.com/exdb/lenet/

Softmax Classifier

- Softmax Classifier (is also the multinomial logistic regression)
- Remember that we can get scores for each classes
- ***Key***: we want to interpret the raw scores as probabilities
  - Probabilities must >= 0
  - Must sum up as 1

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

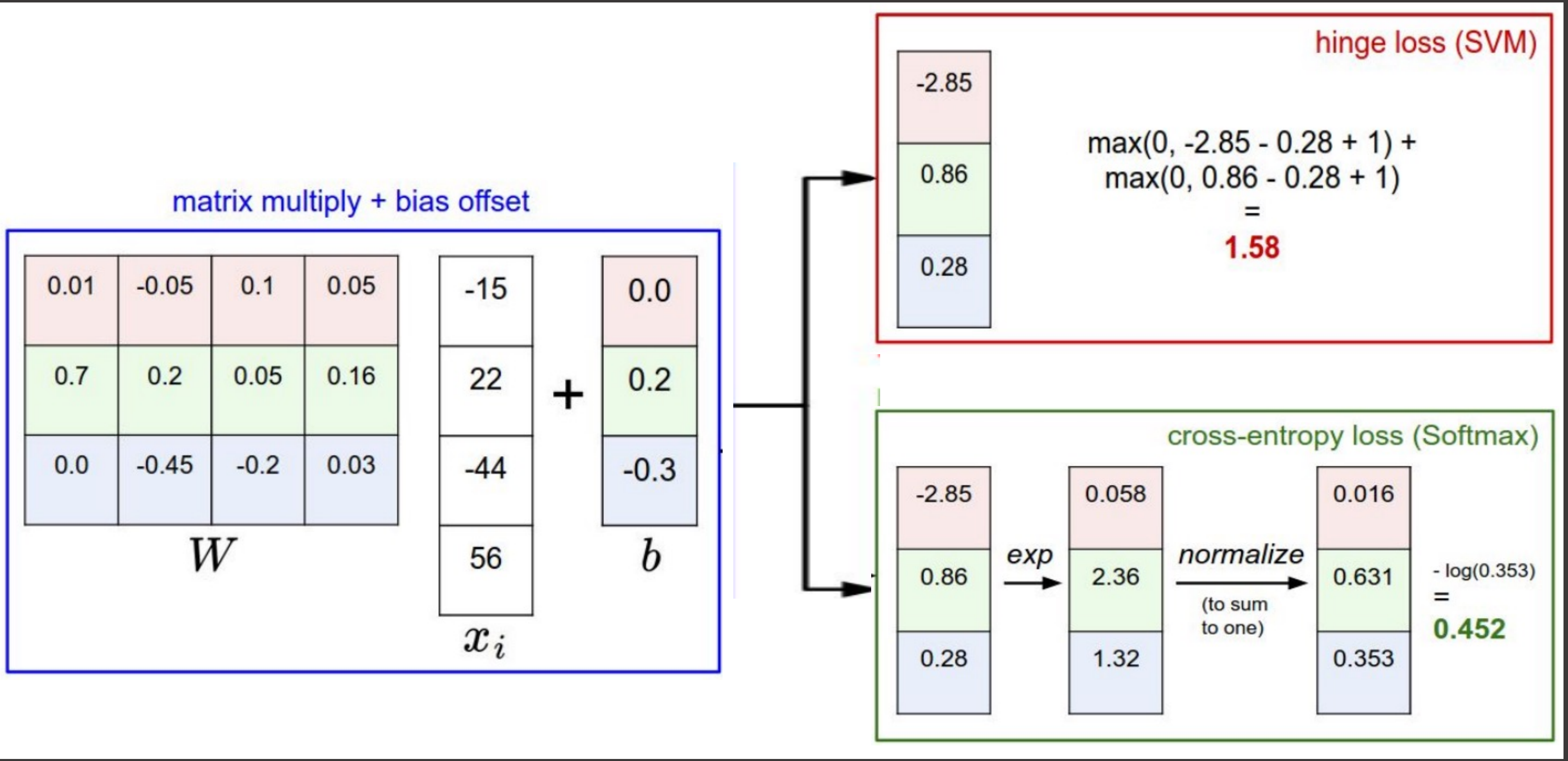
Softmax Classifier

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax vs. SVM (Hinge Loss)

Softmax: $P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$

SVM: $L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$



Slide Credit: Fei-fei Li

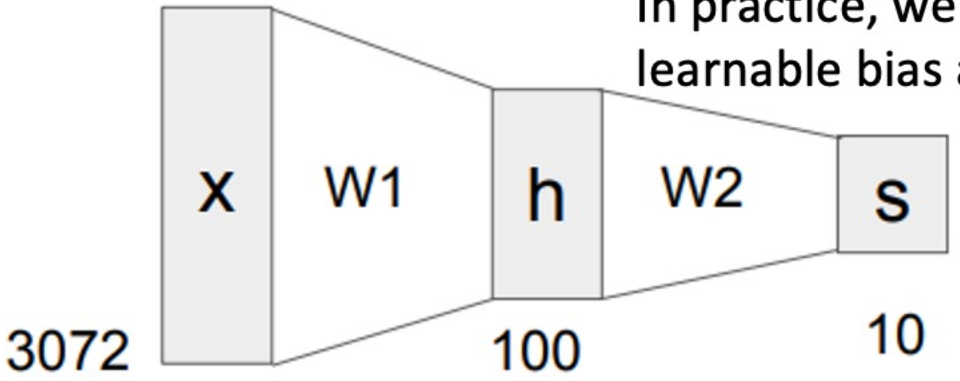
Linear Classifier vs. NN

- (Before) Linear Score Function: $f = W^T X$ $\quad x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$
- (Now) 2-Layer Neural Network: $f = W_2 \max(0, W_1 x)$
  $x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$
- 3-Layer Neural Network: $f = W_3 \max(0, W_2 \max(0, W_1 x))$

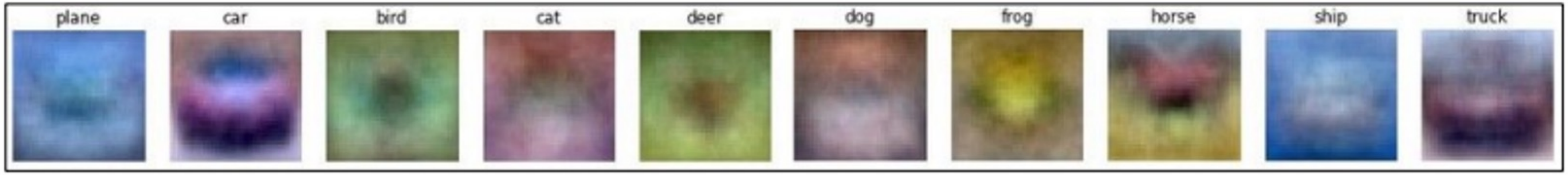
Linear Classifier vs. NN

2-layer Neural Network:

In practice, we will usually add a learnable bias at each layer as well.



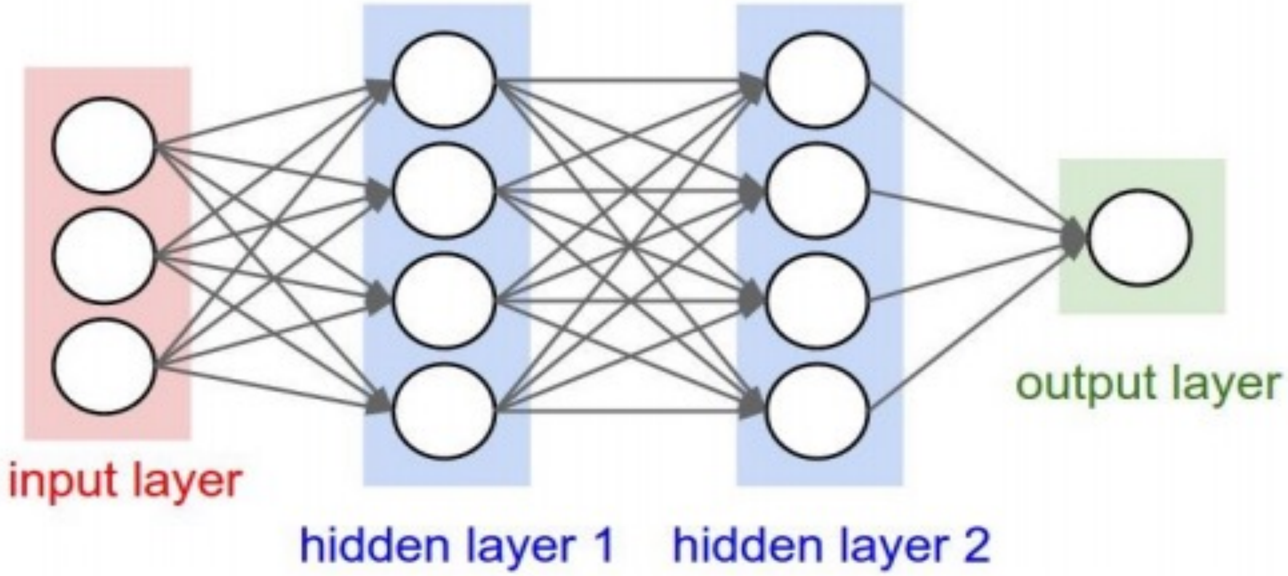
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$



We learned 10 different template images (W), and use these template images for future prediction.

Slide Credit: Fei-fei Li

Example




```python
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```