Artificial Neural Networks

Alymzhan Toleu
alymzhan.toleu@gmail.com

Motivation

Synthesizing Obama: Learning Lip Sync from Audio

SIGGRAPH 2017

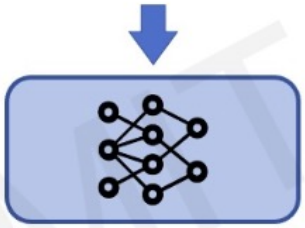
Supasorn Suwajanakorn, Steven M. Seitz, Ira Kemelmacher-Shlizerman

Output Obama Video

demo: https://grail.cs.washington.edu/projects/AudioToObama/

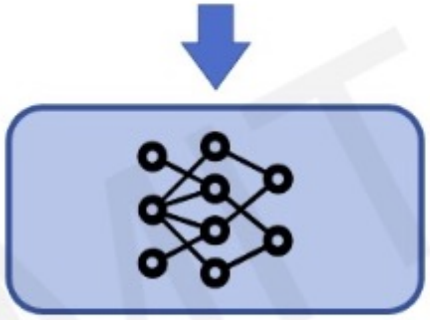
Generating Images from Natural Language

"A photo of an astronaut riding a horse."



Generating Language from Natural Language

"Write code in TensorFlow to train a neural network."



OpenAI.

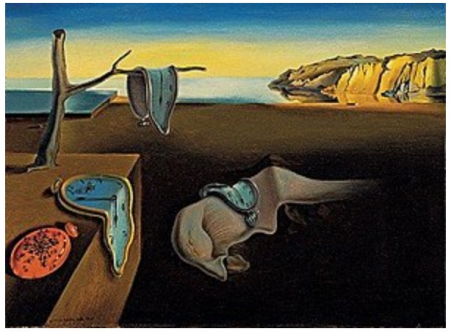
Certainly! Here is an example of how you can use TensorFlow to train a neural network for binary classification:

Copy code

```
import tensorflow as tf

# Load the data
(x_train, y_train), (x_test, y_test) = load_data()

# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=x_train.shape[1:]),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=
['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=
(x_test, y_test))
```

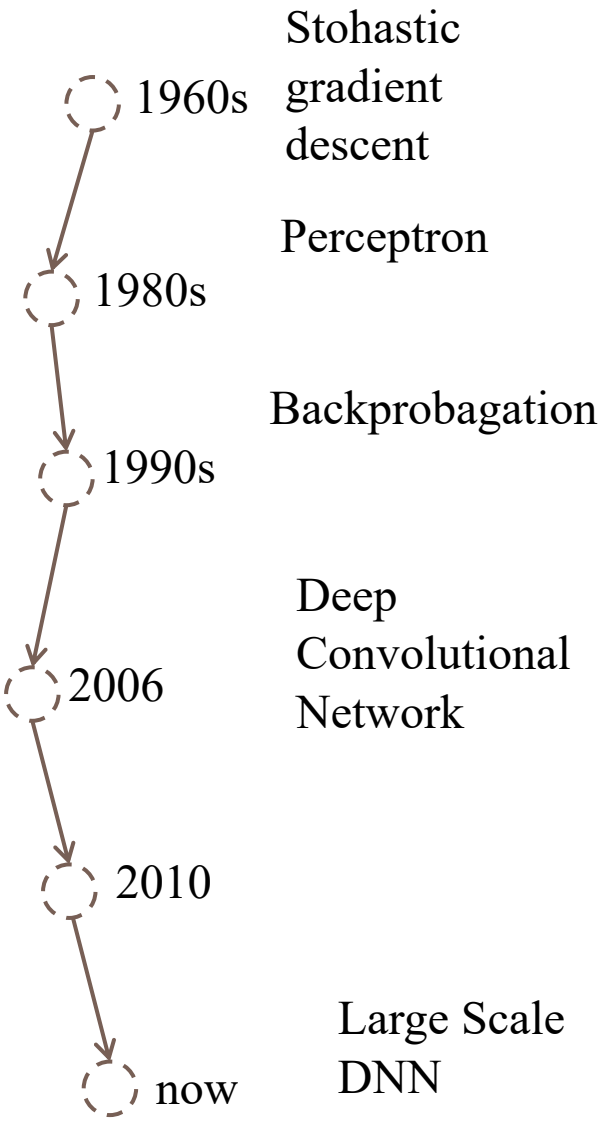
This code assumes that you have a function `load_data()` that returns the training and test datasets as tuples of NumPy arrays. The input shape of the model is determined by the shape of the training data, and the model has two dense layers with 16 and 1 units, respectively, with ReLU and sigmoid activations. The model is then compiled with an Adam optimizer and a binary cross-entropy loss function, and is trained using the `fit()` method.

Generating Music From Text

Painting title and author	Painting image (from Wikipedia)	Painting description	Generated audio
The Persistence of Memory- Salvador Dalí		*"His melting-clock imagery mocks the rigidity of chronometric time. The watches themselves look like soft cheese—indeed, by Dali s own account they were inspired by hallucinations after eating Camembert cheese. In the center of the picture, under one of the watches, is a distorted human face in profile. The ants on the plate represent decay."* By Gromley, Jessica. "The Persistence of Memory". Encyclopedia Britannica, 14 Apr. 2022.	0:00 / 0:30
Napoleon Crossing the Alps - Jacques-Louis David		*"The composition shows a strongly idealized view of the real crossing that Napoleon and his army made across the Alps through the Great St Bernard Pass in May 1800."* By wikipedia	0:00 / 0:30

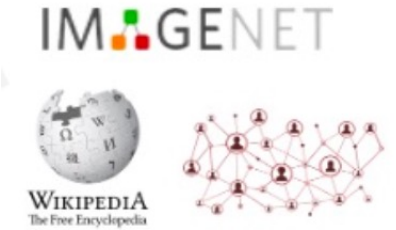
Demo: https://google-research.github.io/seanet/musiclm/examples/

Deep Learning



Why neural network resurgence?

1. **Big Data**
   - Larger Datasets
   - Easier collections



2. **Hardware**
   - Graphics Processing Units (GPU)
   - Massively Parallelizable.

3. **Software**
   - New models
   - Toolboxes

Outline

- Logistic Regression (Recap)
- Neural Networks
- Backpropagations (Training)

Logistic Regerssion (Recap)

Logistic regression

Data: Inputs are continuous vectors of length K. Outputs are discrete.

$$\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N} \text{ where } \mathbf{x} \in \mathbb{R}^K \text{ and } y \in \{0, 1\}$$

Model: Logistic function applied to dot product of parameters with input vector.

$$p_{\boldsymbol{\theta}}(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$

Learning: finds the parameters that minimize some objective function.

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$$

Prediction: Output is the most probable class.

$$\hat{y} = \underset{y \in \{0,1\}}{\operatorname{argmax}}\, p_{\boldsymbol{\theta}}(y|\mathbf{x})$$

Logistic regression: Gradient Descent

Hypothesis: $f_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$ where, $\theta^T x = w_0 x_0 + w_1 x_1 + \cdots + w_n x_n$

Cost function: $J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log f_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - f_\theta(x^{(i)})]$

Parameters: $\theta = \{w_0, \ldots, w_n\}$

Goal: minimize $J(\theta)$

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} [f_\theta(x^{(i)}) - y^{(i)}] x^{(i)}$$

$f_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$

}

(simultaneously update all θ)

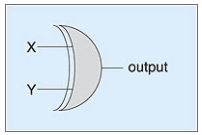
Neural Networks

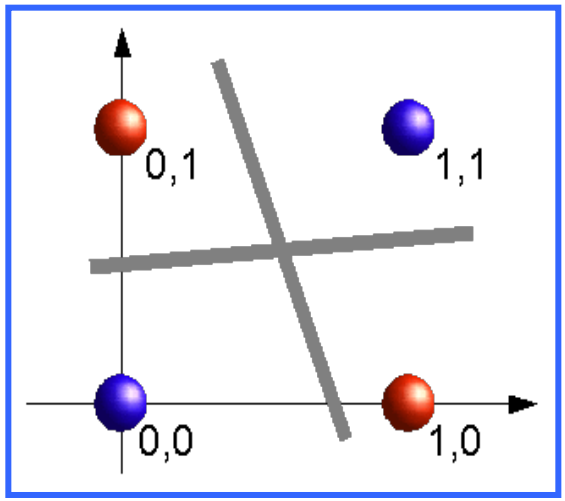
Learning highly non-linear functions

f: X → Y

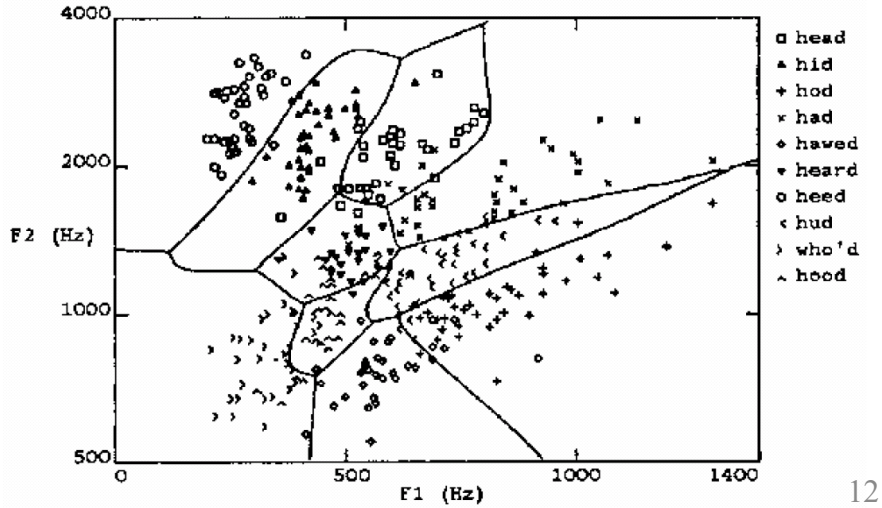
- f might be non-linear function
- X (vector of) continuous and/or discrete vars
- Y (vector of) continuous and/or discrete vars

The XOR gate



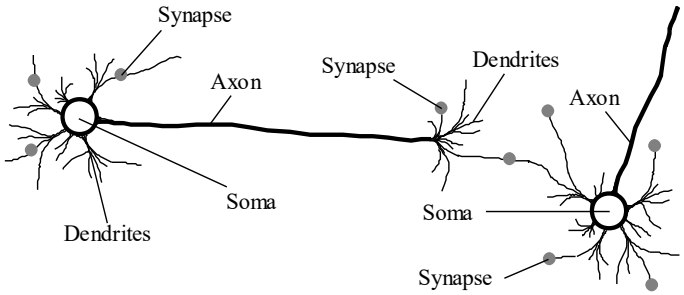


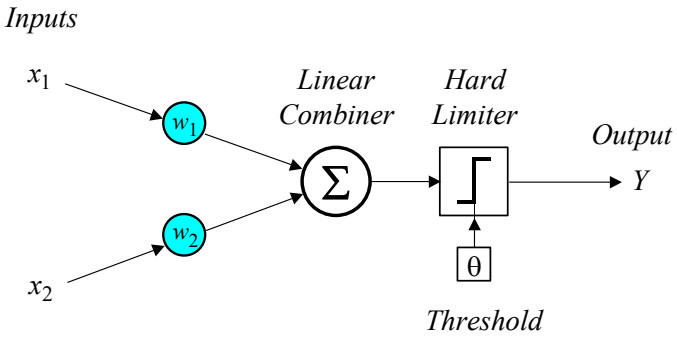
Speech recognition



Perceptron and Neural Nets

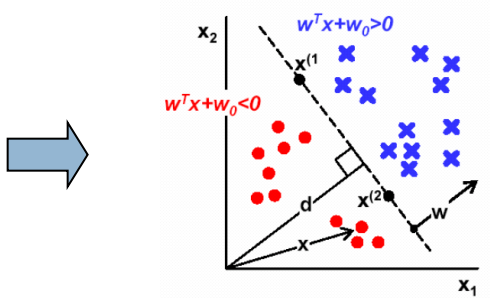
- From biological neuron to artificial neuron (perceptron)



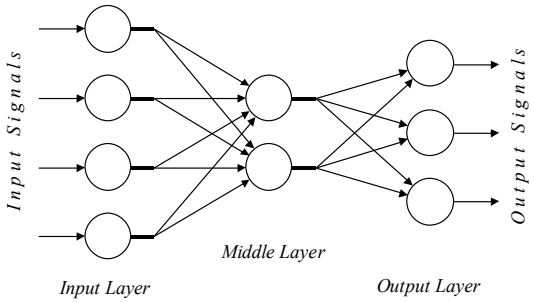


- Activation function

$$X = \sum_{i=1}^{n} x_i w_i \qquad y = \begin{cases} +1, & \text{if } X \geq \omega_0 \\ -1, & \text{if } X < \omega_0 \end{cases}$$

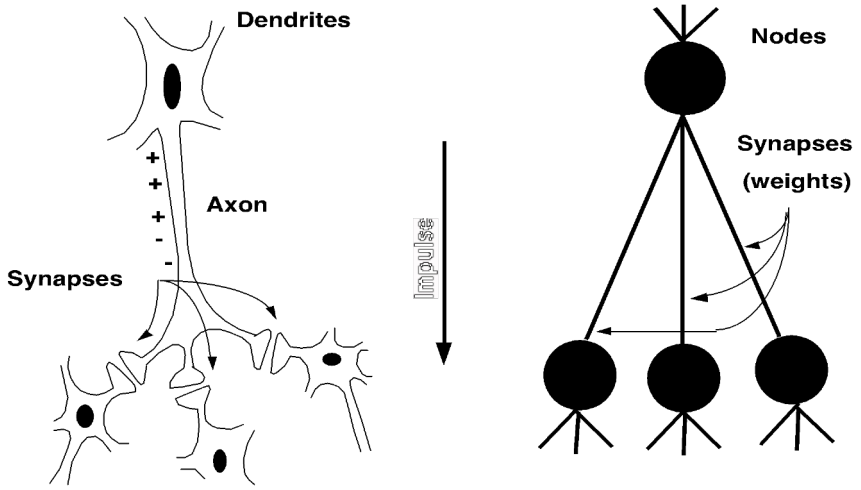


- Artificial neuron networks
  - supervised learning
  - gradient descent

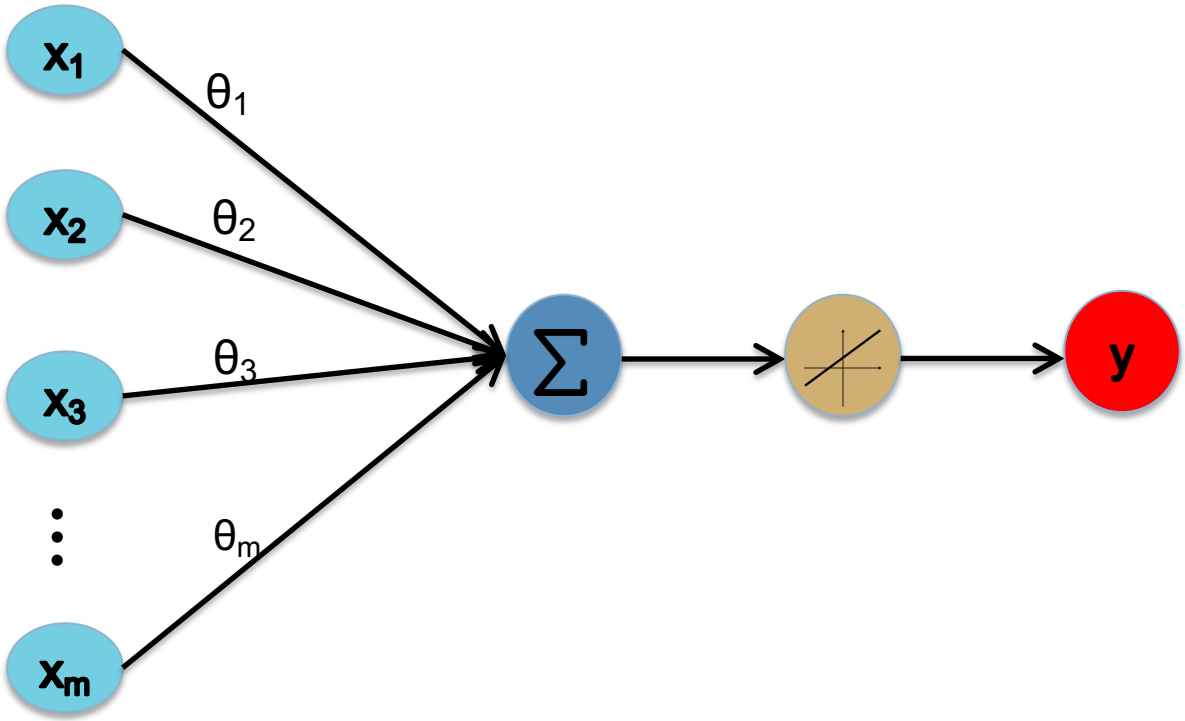


Connectionist Models

- Consider humans:
  - Neuron switching time
    ~ 0.001 second
  - Number of neurons
    ~ 10^{10}
  - Connections per neuron
    ~ 10^{4-5}
  - Scene recognition time
    ~ 0.1 second
  - 100 inference steps doesn't seem like enough
    → much parallel computation
- Properties of artificial neural nets (ANN)
  - Many neuron-like threshold switching units
  - Many weighted interconnections among units
  - Highly parallel, distributed processes



Linear Regression



Linear combination of inputs

Output

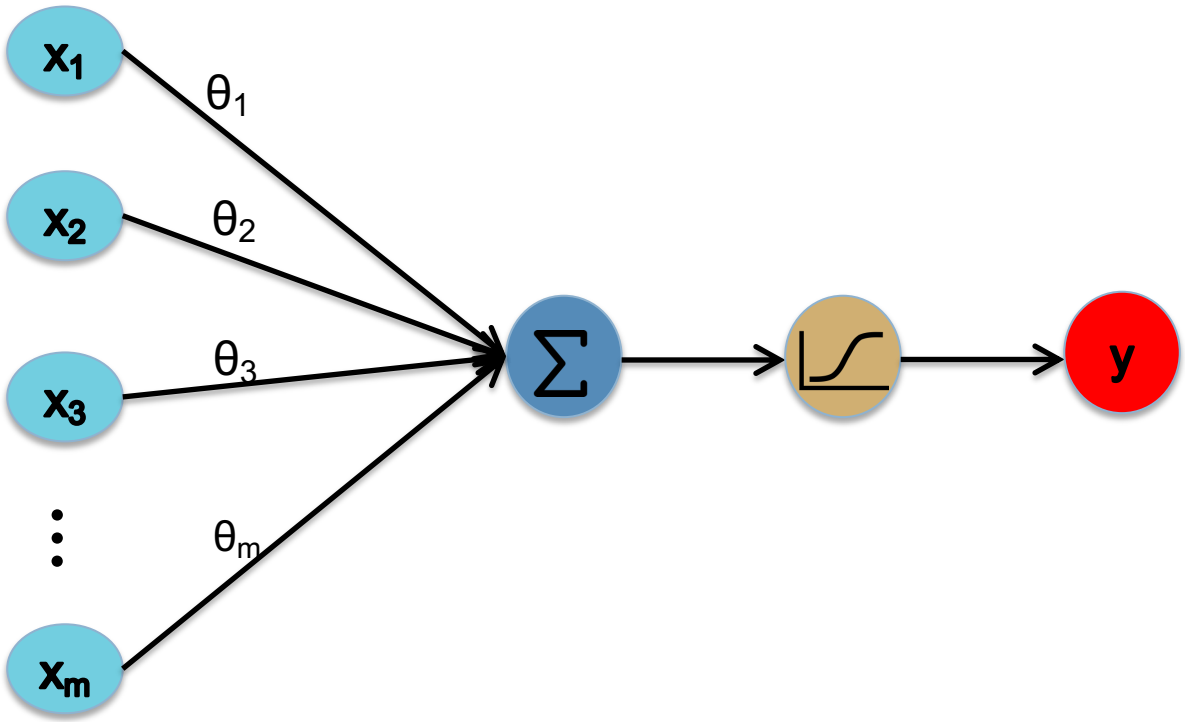
$$y = g\left(\sum_i x_i \theta_i\right)$$

Where, $g(z) = z$

Activation function

Input Weights Sum Linear Output

Logistic Regression



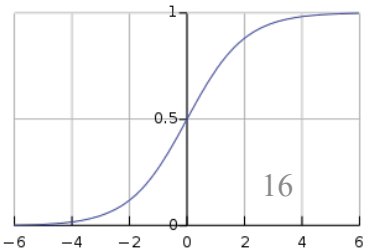
Linear combination of inputs

Output

$$y = g\left(\sum_i x_i \theta_i\right)$$

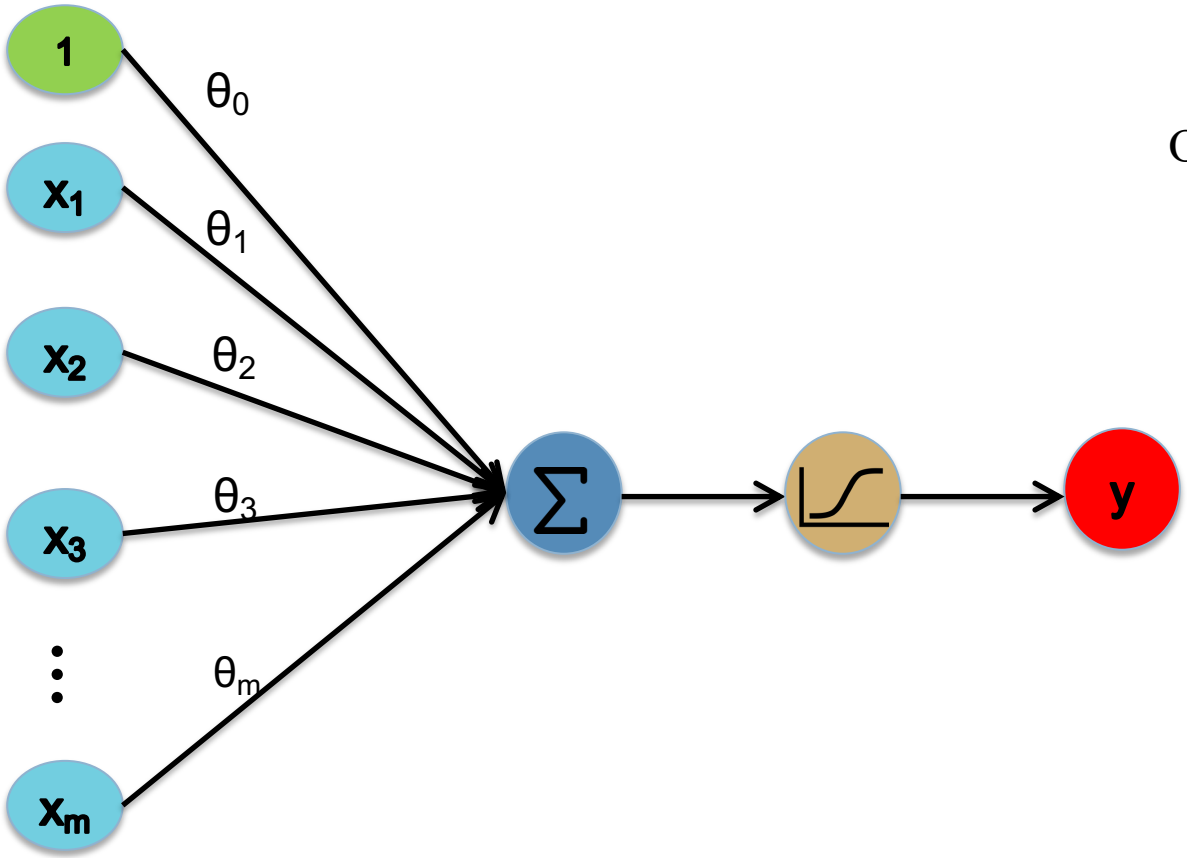
Sigmoid activation function

$$where, g(z) = \frac{1}{1+e^{-z}}$$



Input Weights Sum Non-Linearity Output

The Perceptron: Forward Propagation



Linear combination of inputs

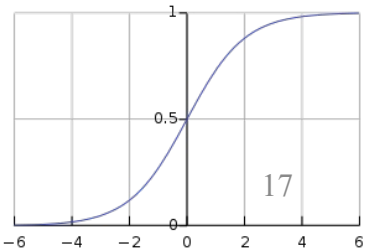
Output

Bias

$$y = g\left(\theta_0 + \sum_{i=1}^{m} x_i\theta_i\right)$$

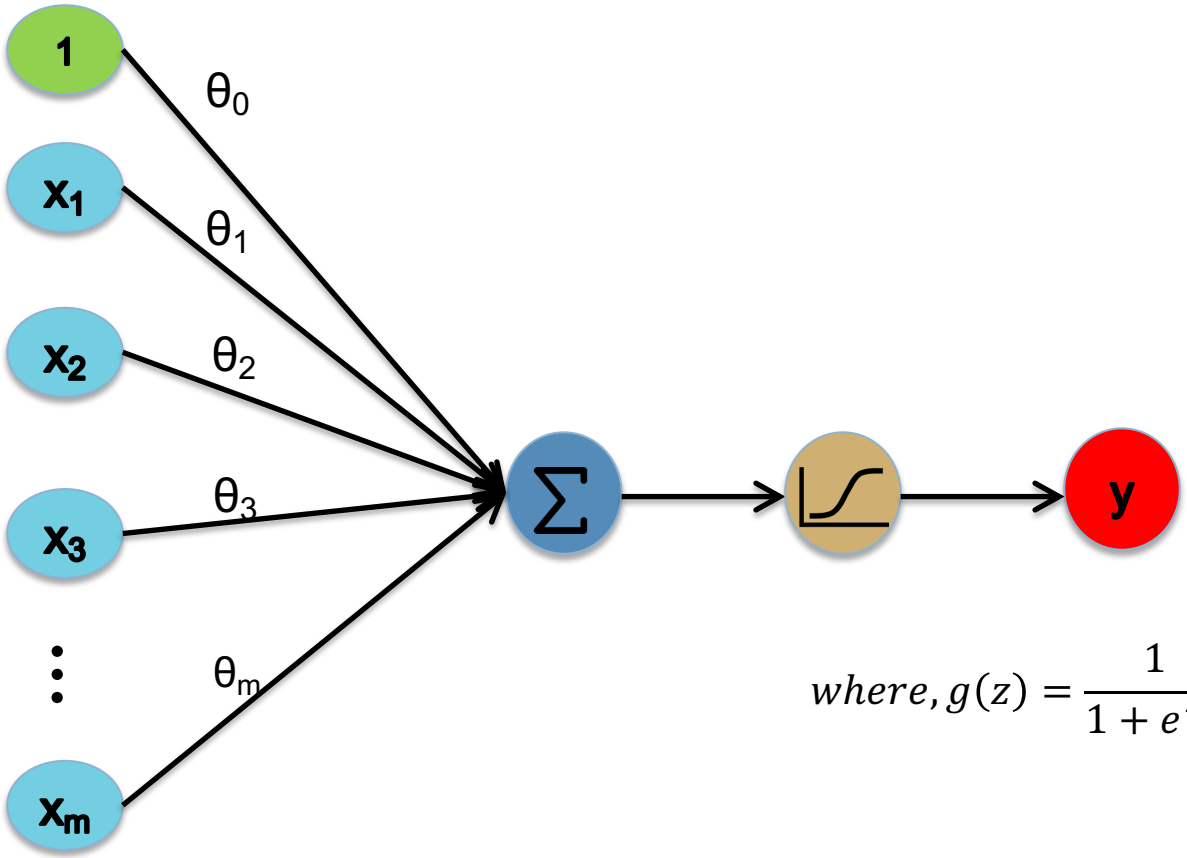
Activation function

$$where, g(z) = \frac{1}{1+e^{-z}}$$



Input Weights Sum Non-Linearity Output

The Perceptron: Forward Propagation

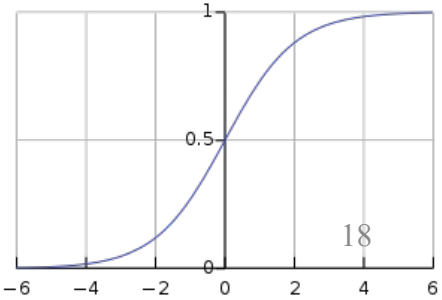


$$y = g\left(\theta_0 + \sum_{i=1}^{m} x_i\theta_i\right)$$

$$y = g(\theta_0 + X^T\Theta)$$

$$where, g(z) = \frac{1}{1+e^{-z}}$$

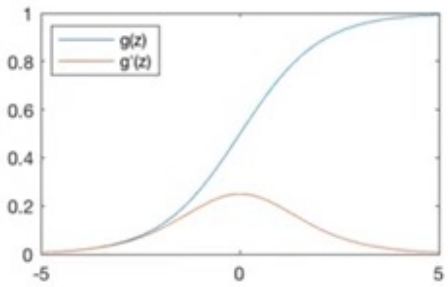
$$X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{bmatrix} \quad \Theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \dots \\ \theta_m \end{bmatrix}$$



Input Weights Sum Non-Linearity Output

Common Activation Function

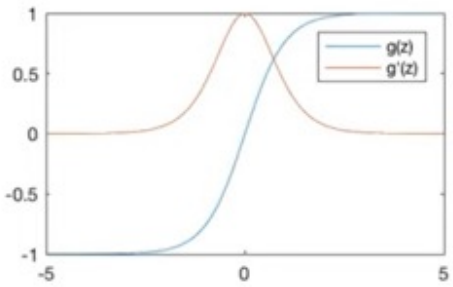
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

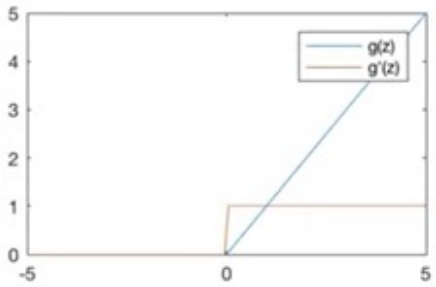
Hyperbolic Tangent



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

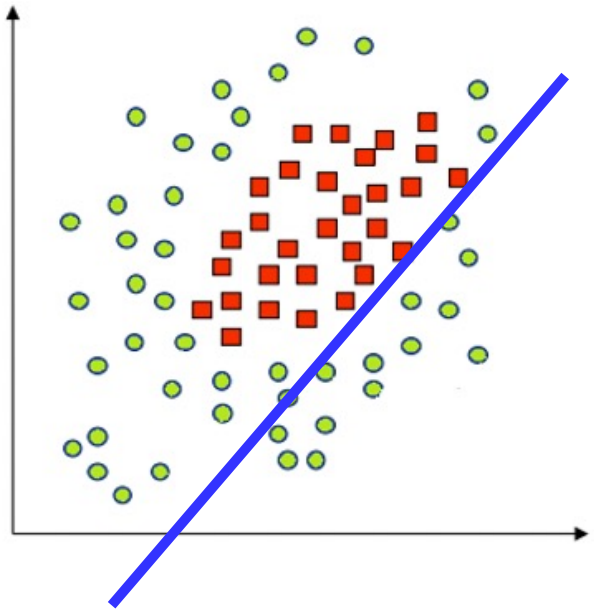
Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Importance of Activation Function



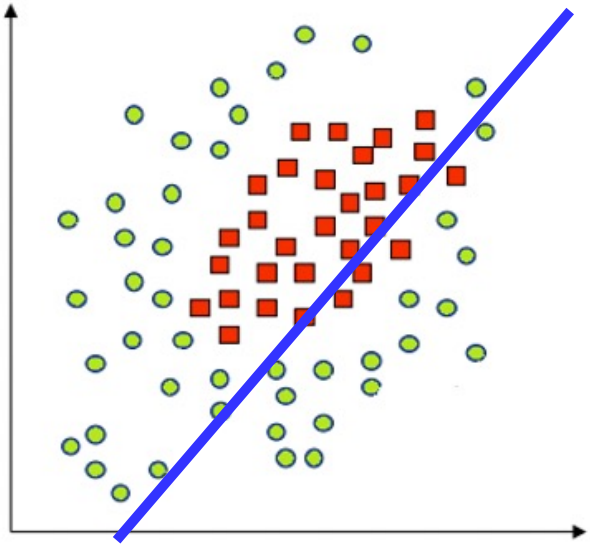
What if we wanted to build a network to distiguish the green vs reds points?

Linear activation function produces linear decision no matter the network size.

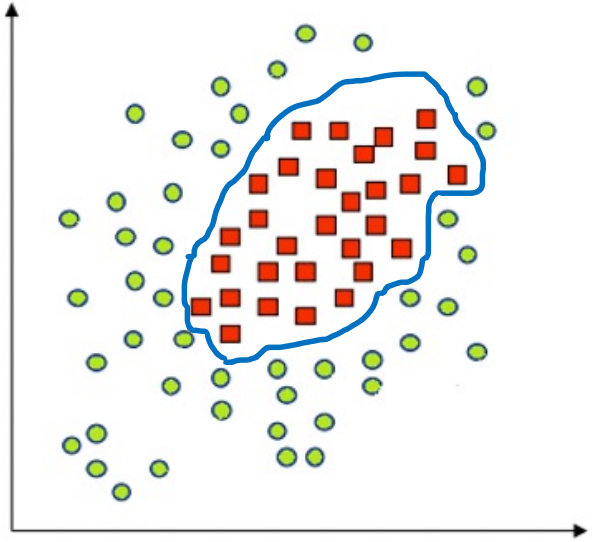
The purpose of a activation function is to introduce non-linearities into the network.

Importance of Activation Function

The purpose of a activation function is to introduce non-linearities into the network.

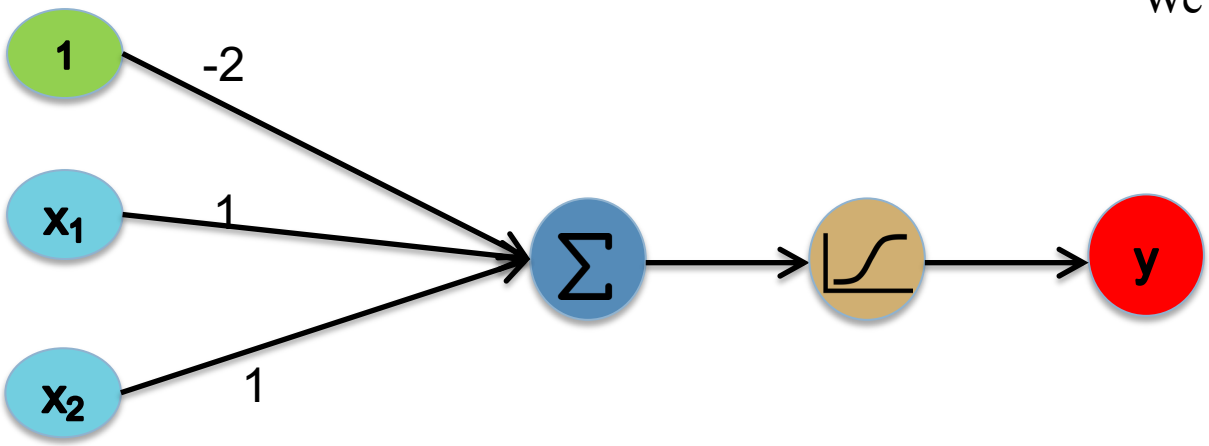


Linear activation function produces linear decision no matter the network size.



Non-Linearity allow us approximate arbitrarily complex function.

The Perceptron: Example



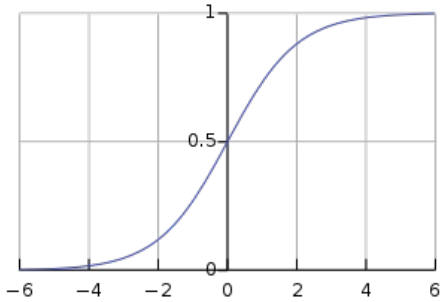
We have : $\theta_0 = -2$, $\Theta = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

$$y = g(\theta_0 + X^T \Theta)$$
$$= g(-2 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 1 \\ 1 \end{bmatrix})$$

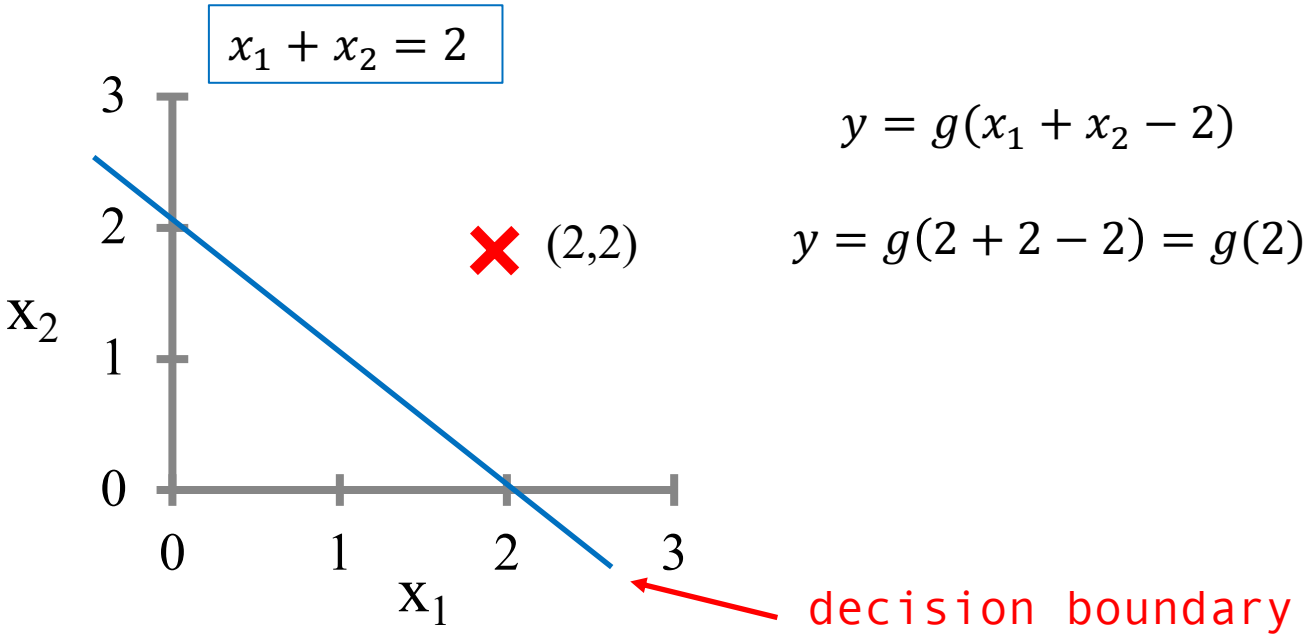
$$y = g(\underbrace{x_1 + x_2 - 2}_{\text{This is a 2D line.}})$$

Input Weights Sum Non-Linearity Output

$$where, g(z) = \frac{1}{1 + e^{-z}}$$



The Perceptron: Decision Boundary



$$y = g(x_1 + x_2 - 2)$$

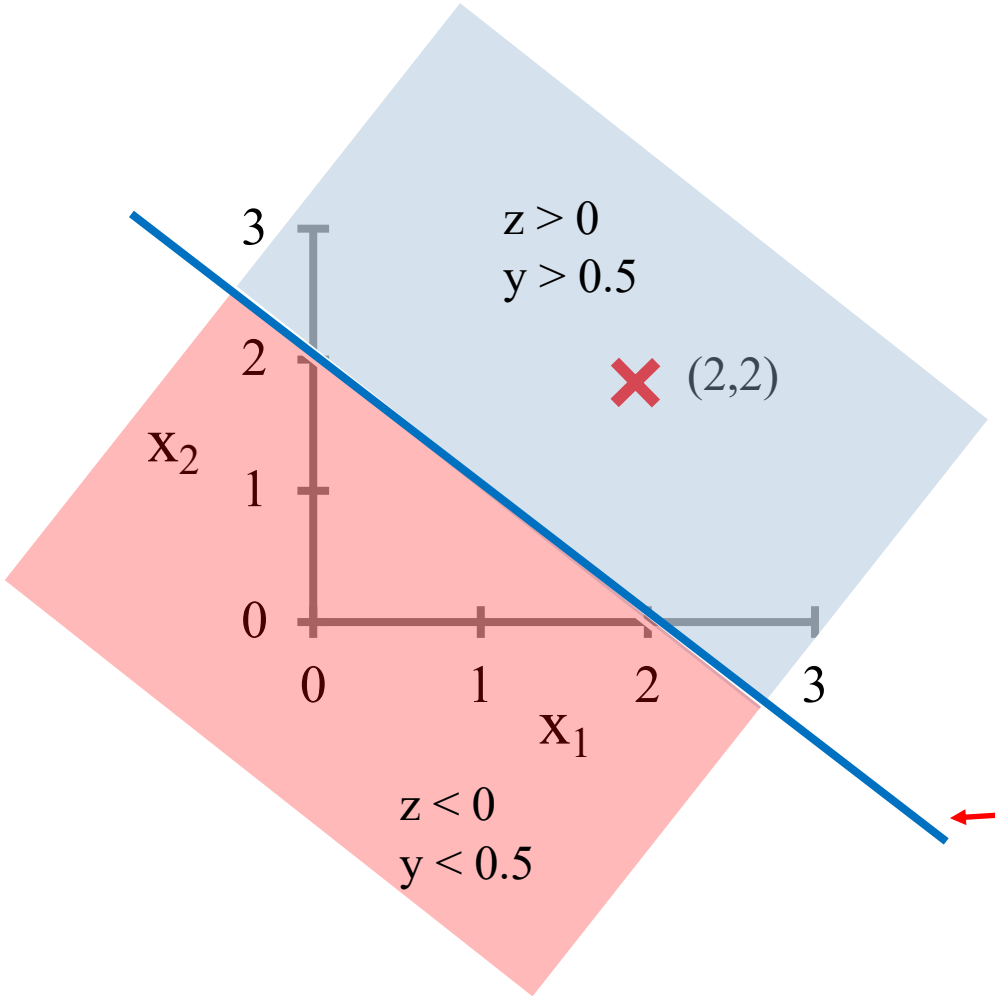
$$y = g(2 + 2 - 2) = g(2) \approx 0.88$$

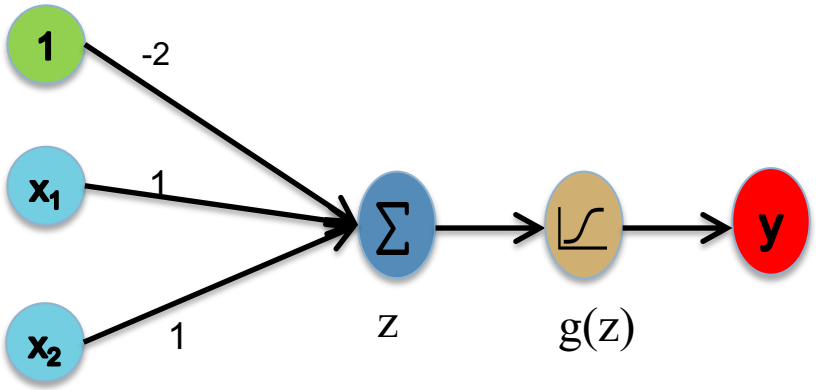
Predict $y = 1$, $g(\theta^T x) \geq 0.5$, $\theta^T x \geq 0$ $\Rightarrow$ $-2 + x_1 + x_2 \geq 0$

Predict $y = 0$, $g(\theta^T x) < 0.5$, $\theta^T x < 0$ $\Rightarrow$ $-2 + x_1 + x_2 < 0$

The Perceptron: Decision Boundary

$$y = g(x_1 + x_2 - 2)$$





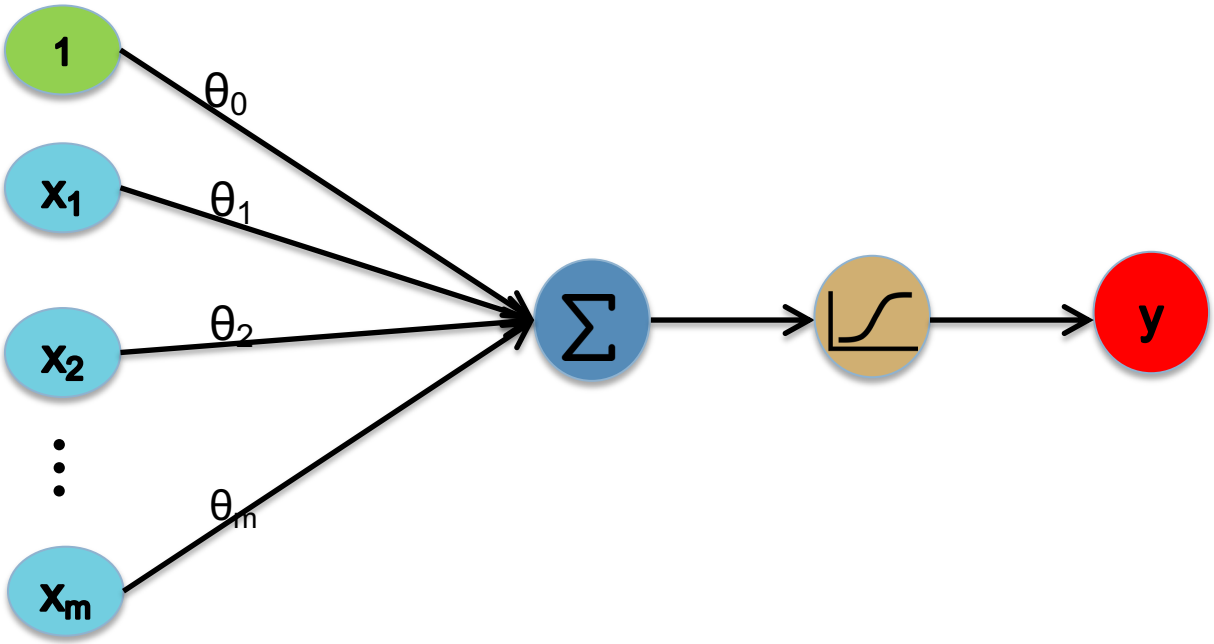
decision boundary

$$x_1 + x_2 = 2$$

Build a Neural Network with Perceptron

Peceptron: Single Neuron

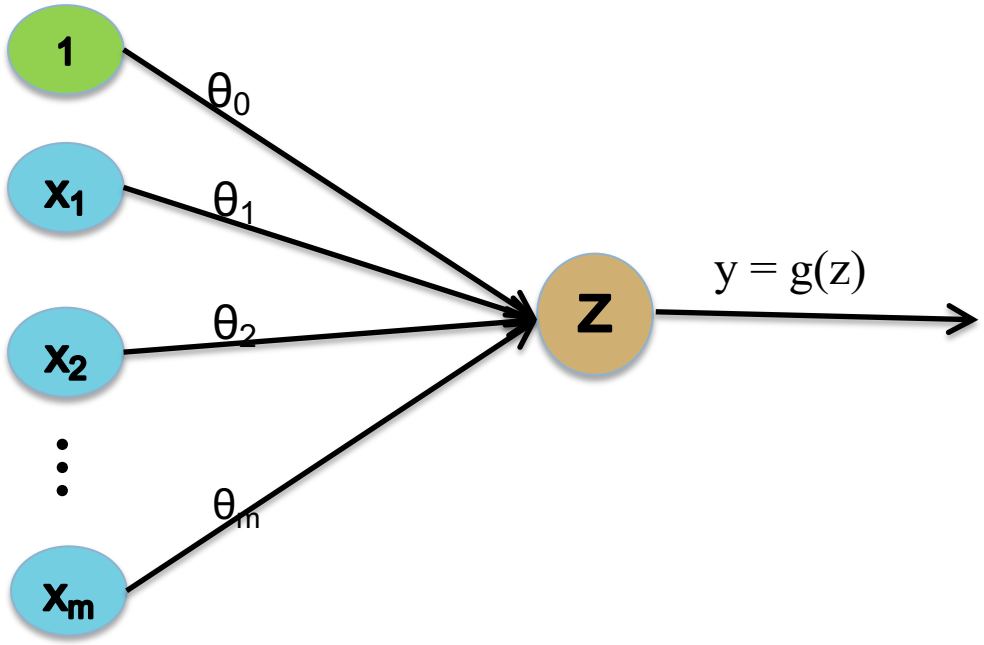
$$y = g(\theta_0 + X^T \Theta)$$



Input Weights Sum Non-Linearity Output

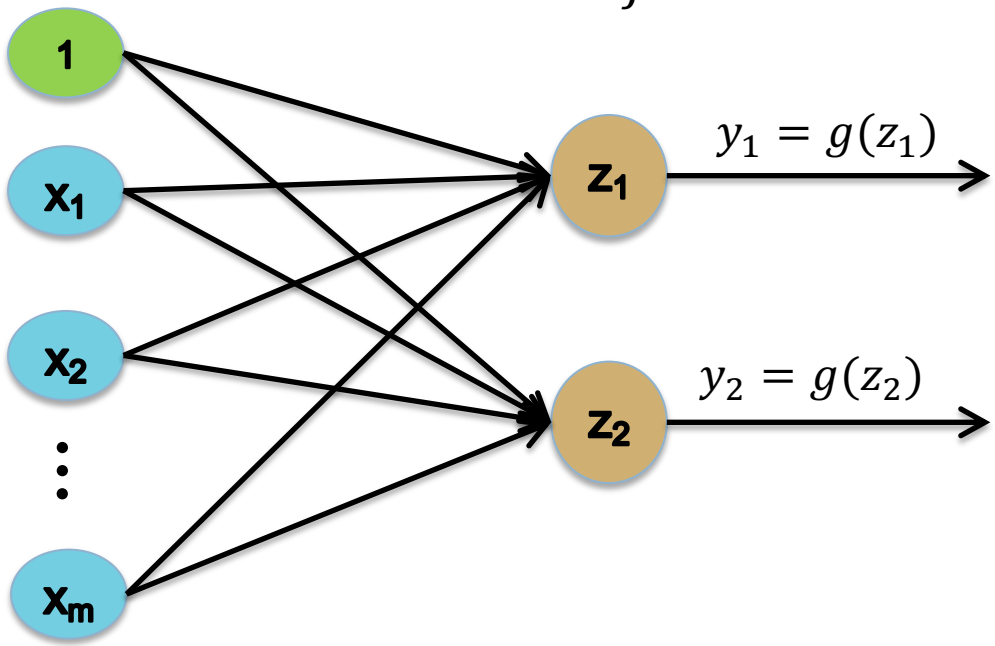
Peceptron: Single Neuron

$$z = \theta_0 + \sum_{j=1}^{m} x_j \theta_j$$



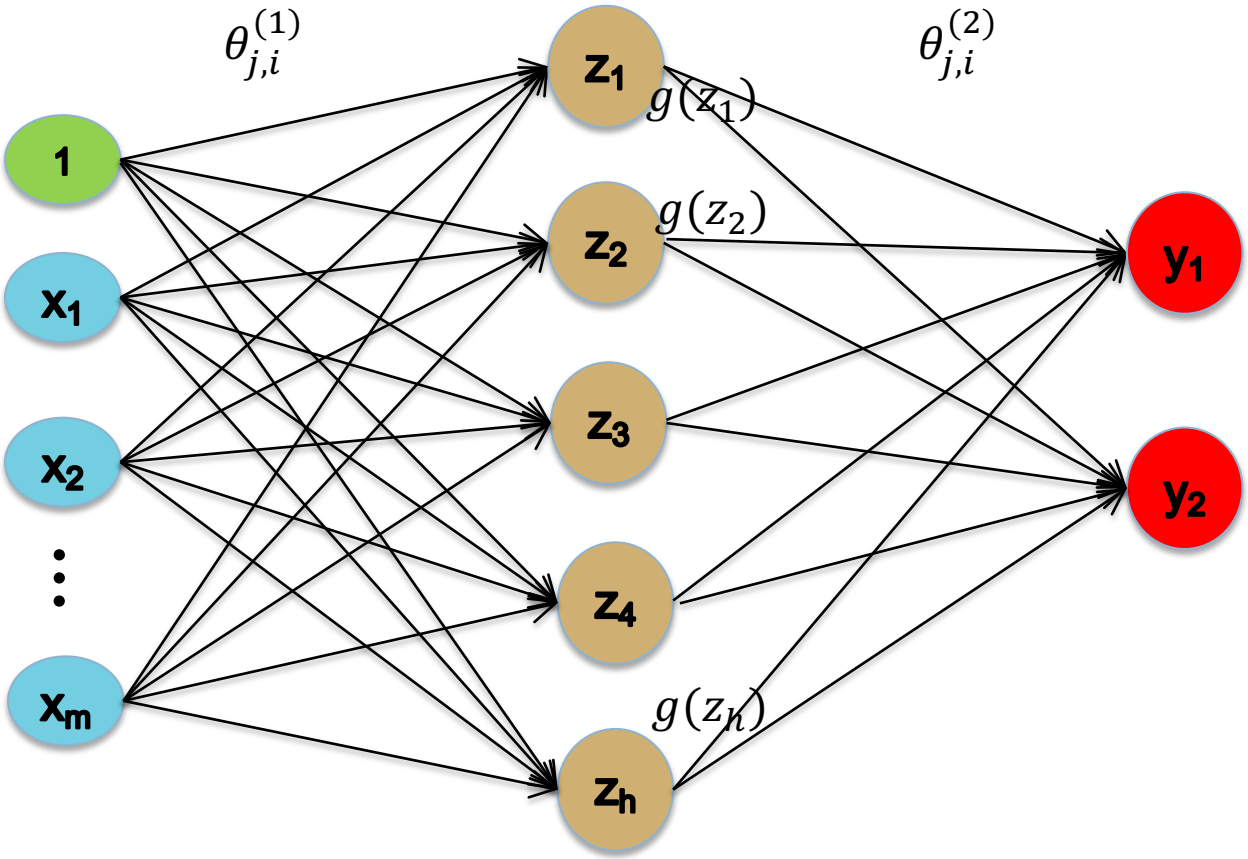
Peceptron: Multiple Neurons

$$z_i = \theta_{0,i} + \sum_{j=1}^{m} x_j \theta_{j,i}$$



All inputs are fully connected to ouputs, it is called Dense layer.

Single Layer Neural Network

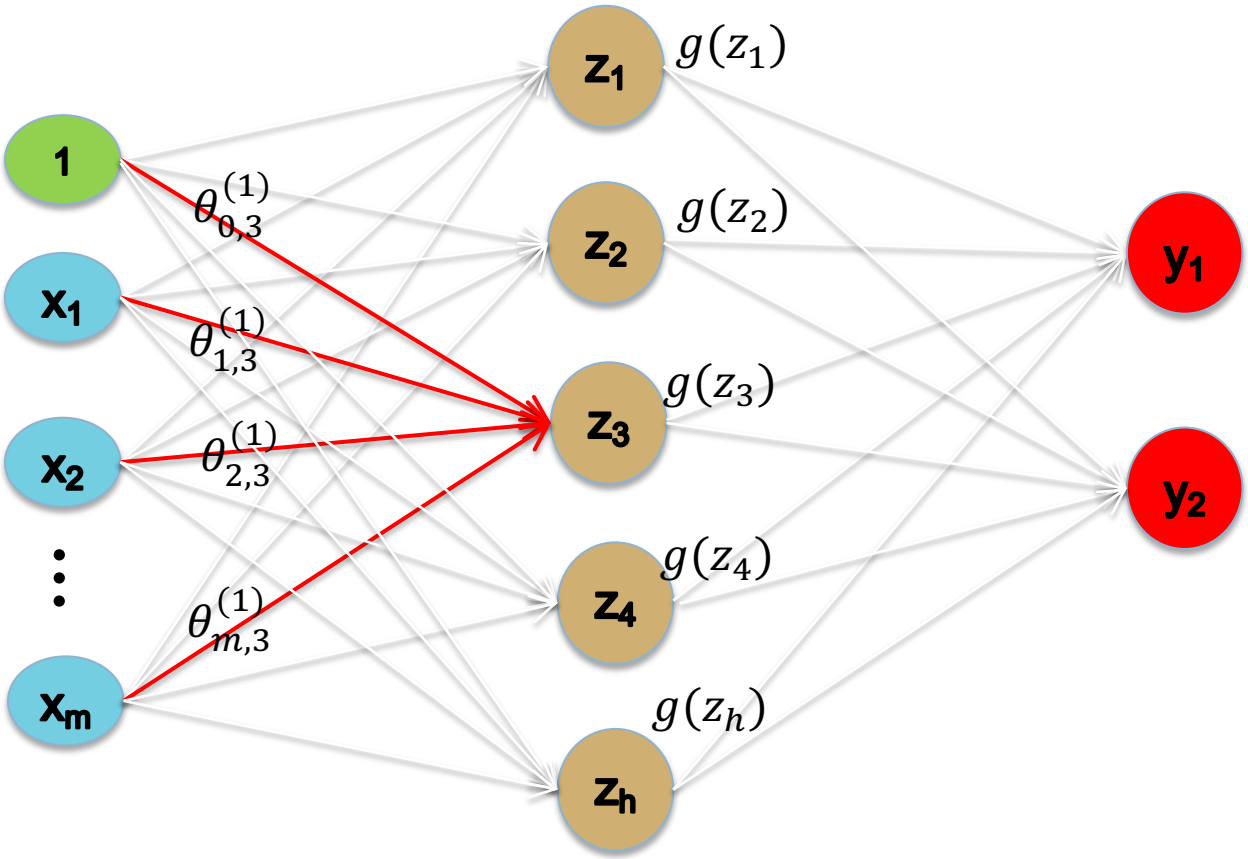


Inputs Hidden Layer Outputs

$$z_i = \theta_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \theta_{j,i}^{(1)}$$

$$y_i = g\left(\theta_{0,i}^{(2)} + \sum_{j=1}^{h} g(z_j)\,\theta_{j,i}^{(2)}\right)$$

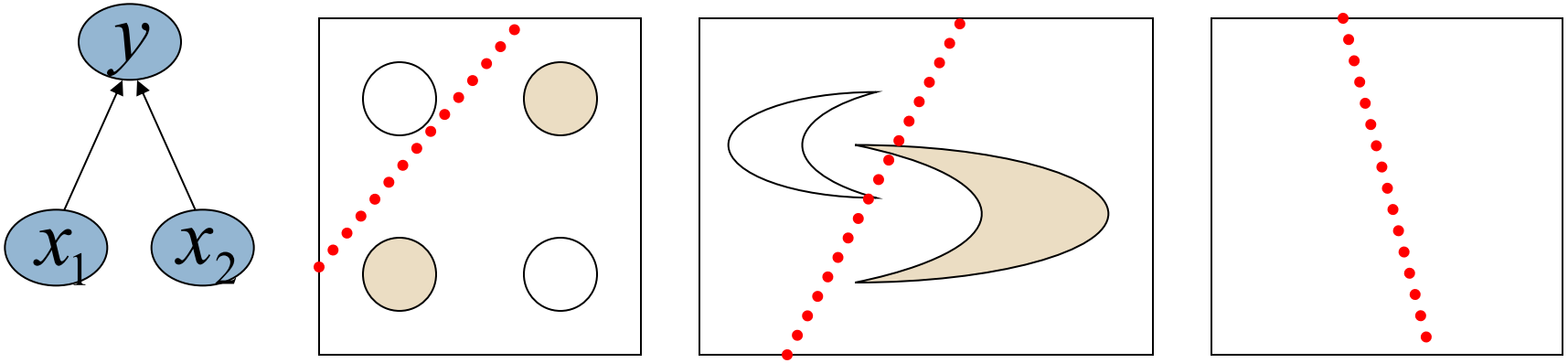
Single Layer Neural Network



$$z_3 = \theta_{0,3}^{(1)} + \sum_{j=1}^{m} x_j \theta_{j,3}^{(1)} = \theta_{0,3}^{(1)} + x_1\theta_{1,3}^{(1)} + x_2\theta_{2,3}^{(1)} + \cdots + x_m\theta_{m,3}^{(1)}$$

Decision Boundary

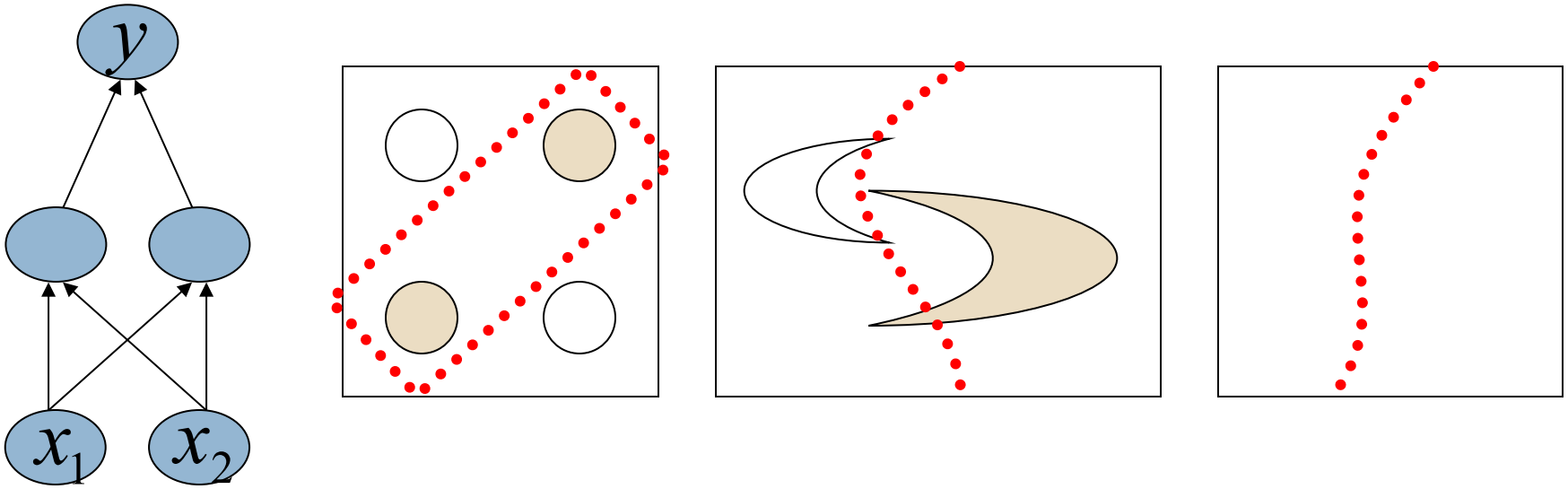
- 0 hidden layers: linear classifier
  - Hyperplanes



Example from to Eric Postma via Jason Eisner

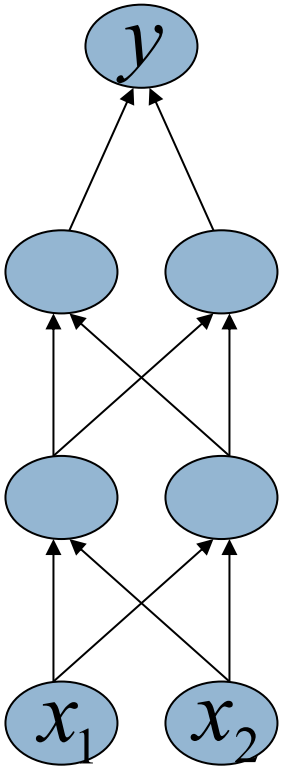
Decision Boundary

- 1 hidden layer
  - Boundary of convex region (open or closed)

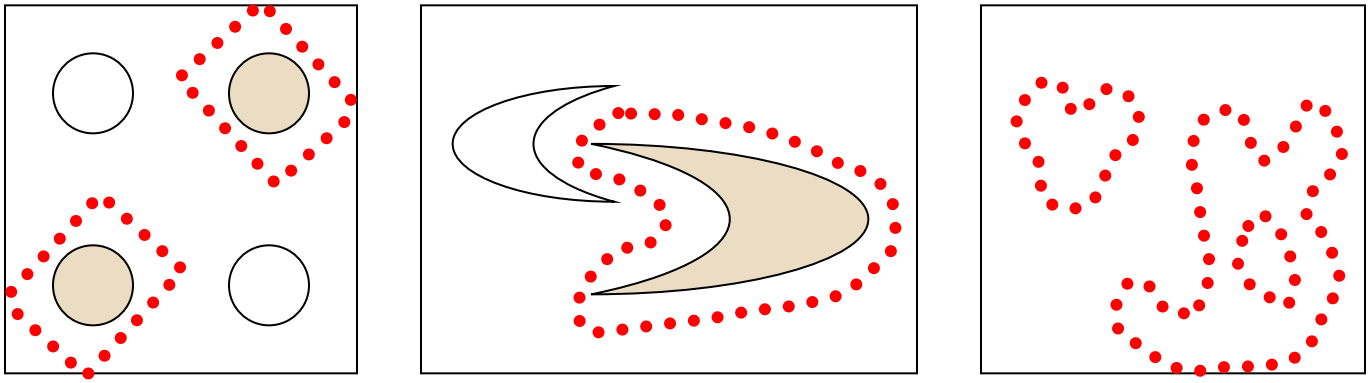


Example from to Eric Postma via Jason Eisner

Decision Boundary

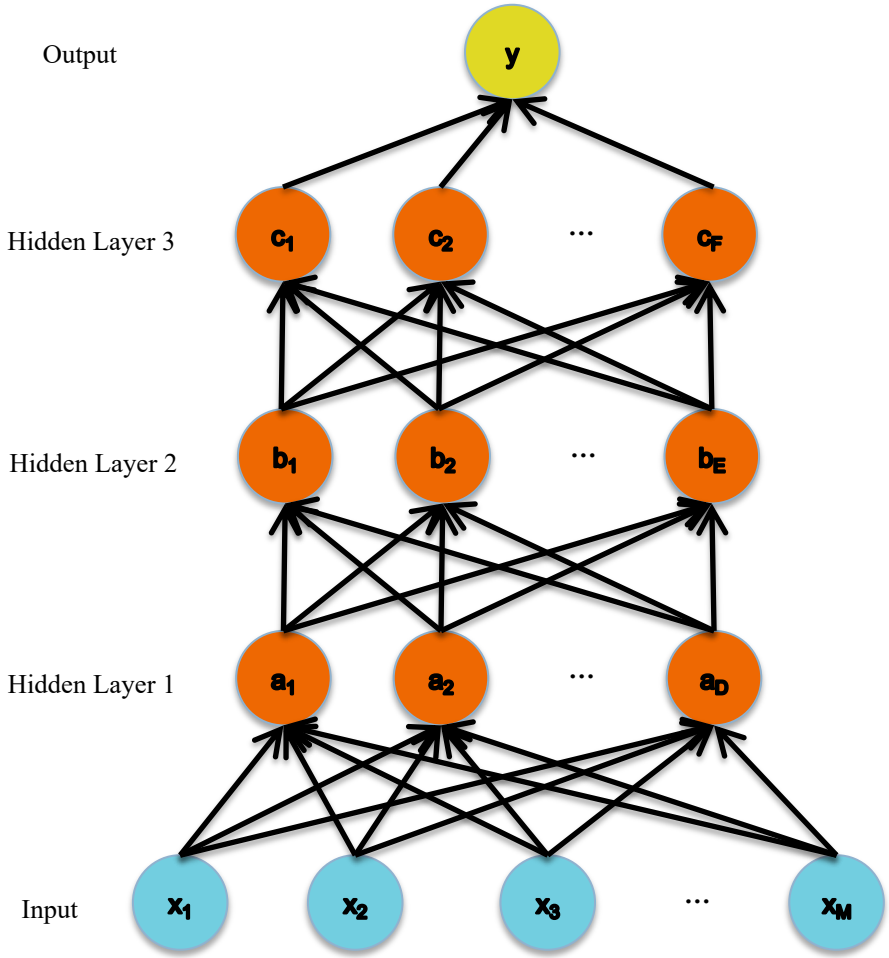


- 2 hidden layers
  - Combinations of convex regions



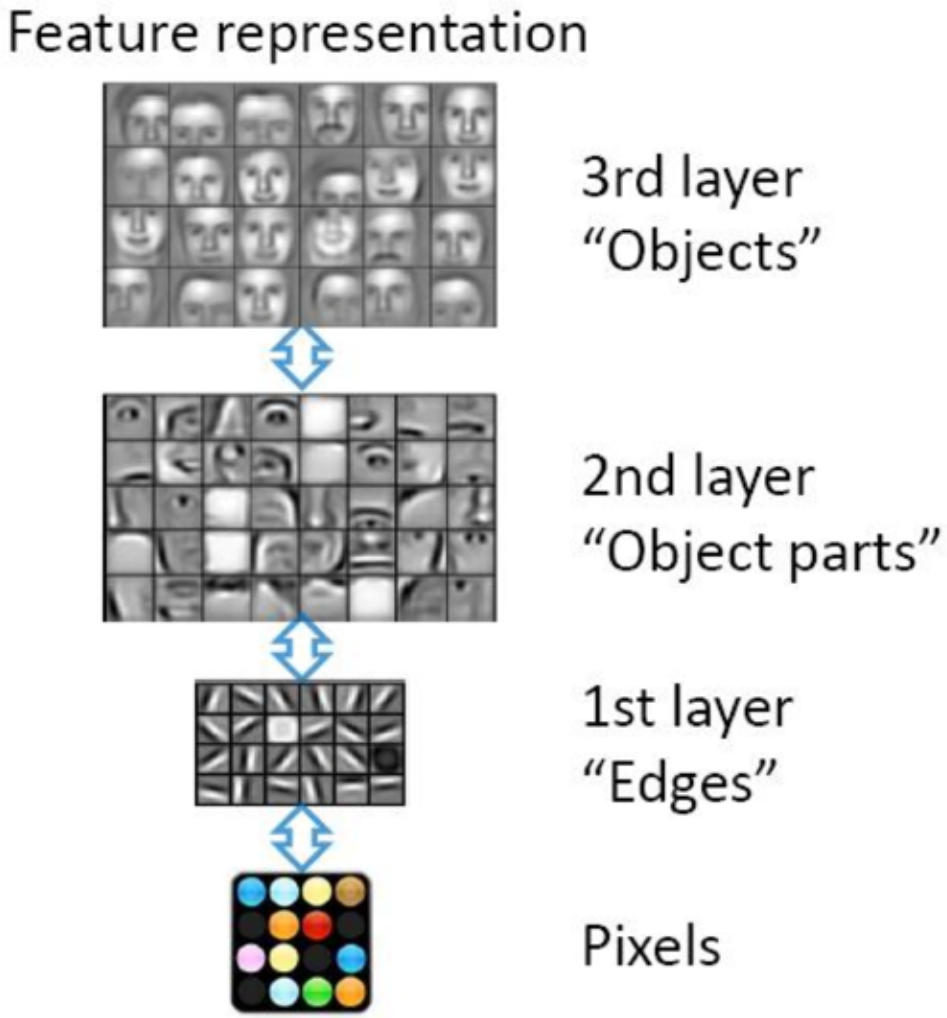
Example from to Eric Postma via Jason Eisner

Deeper Neural Network



Different Levels of Abstraction

- We don't know the "right" levels of abstraction
- So let the model figure it out!

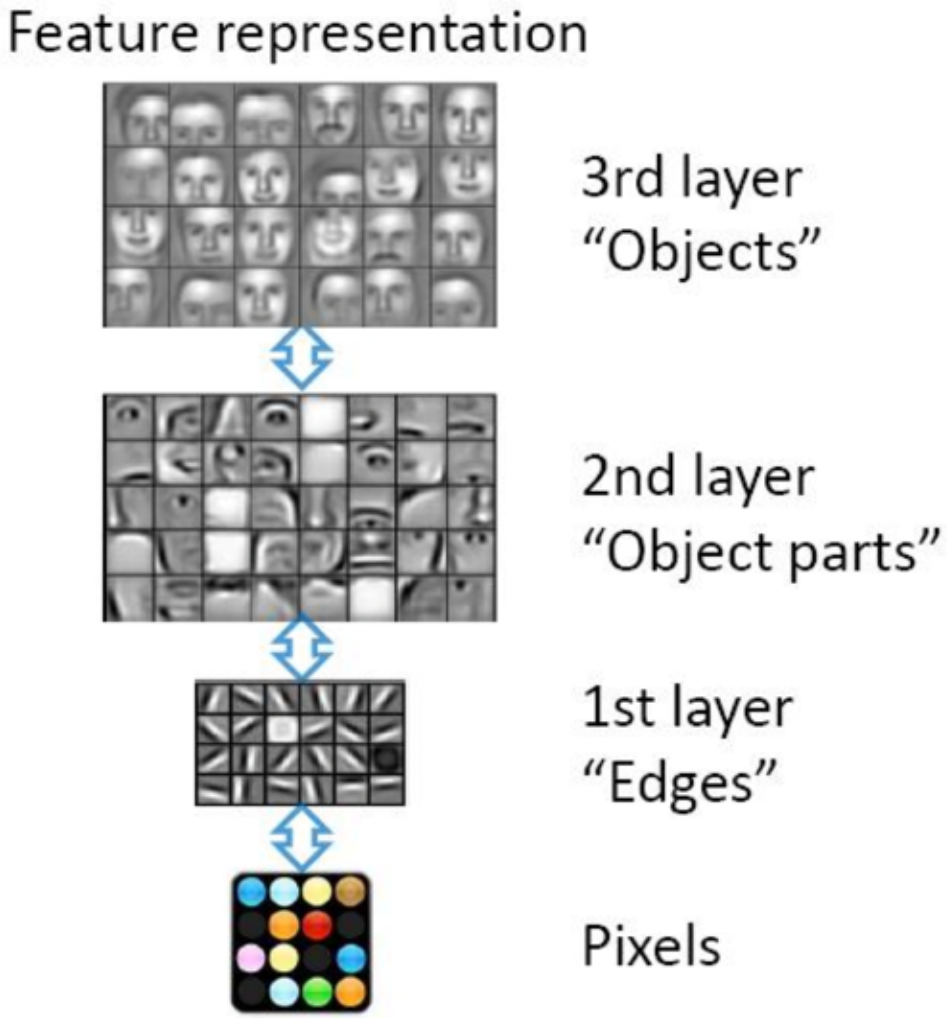


Example from Honglak Lee (NIPS 2010)

Different Levels of Abstraction

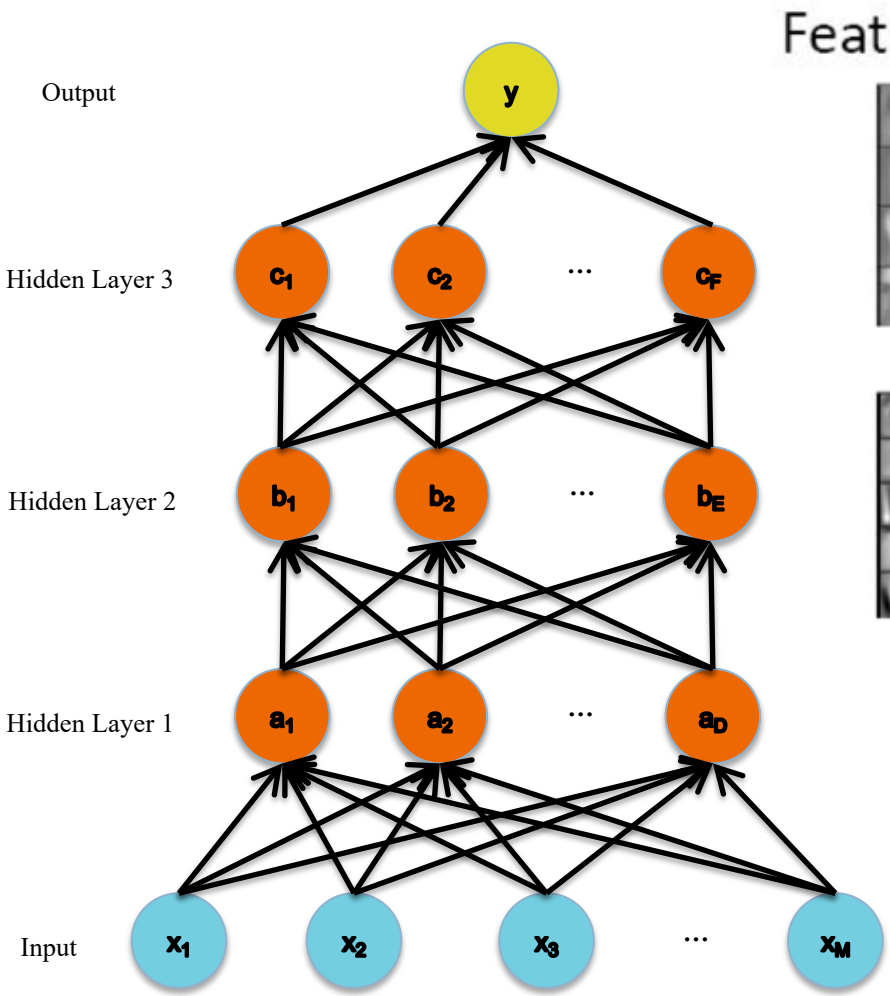
Face Recognition:

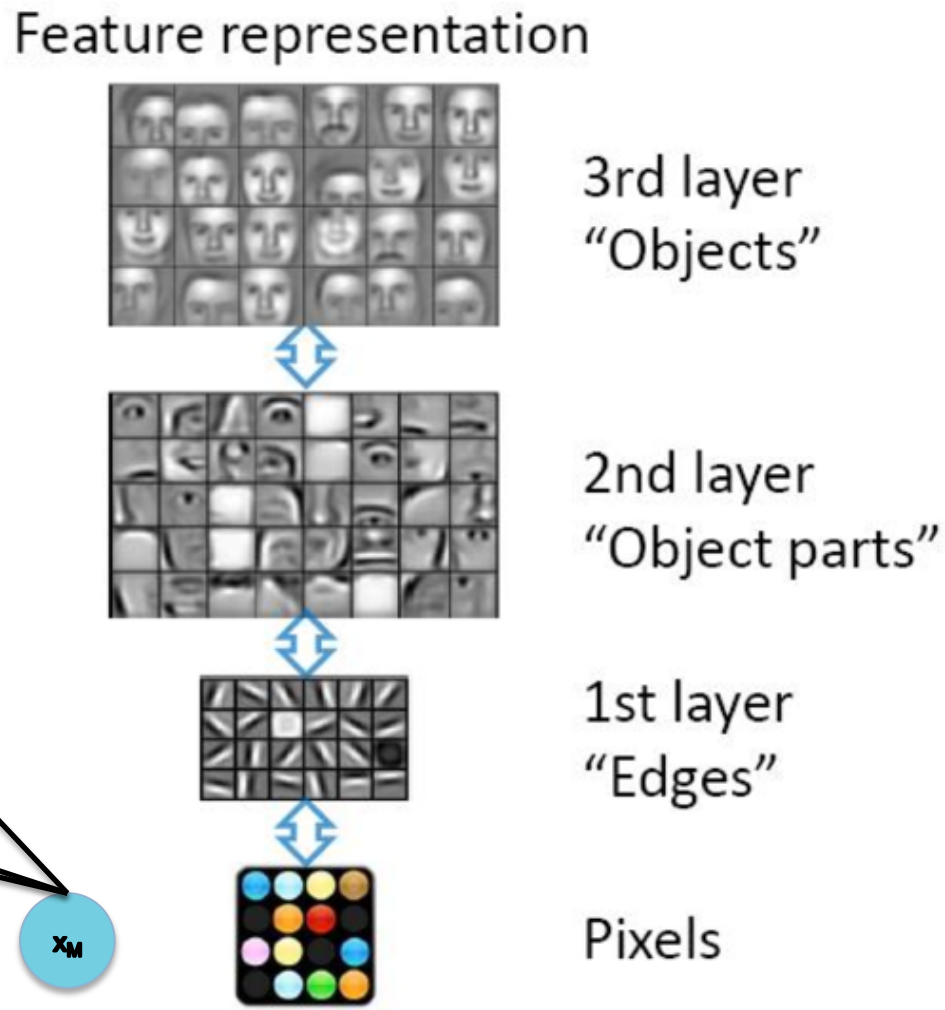
- Deep Network can build up increasingly higher levels of abstraction
- Lines, parts, regions



Example from Honglak Lee (NIPS 2010)

Different Levels of Abstraction

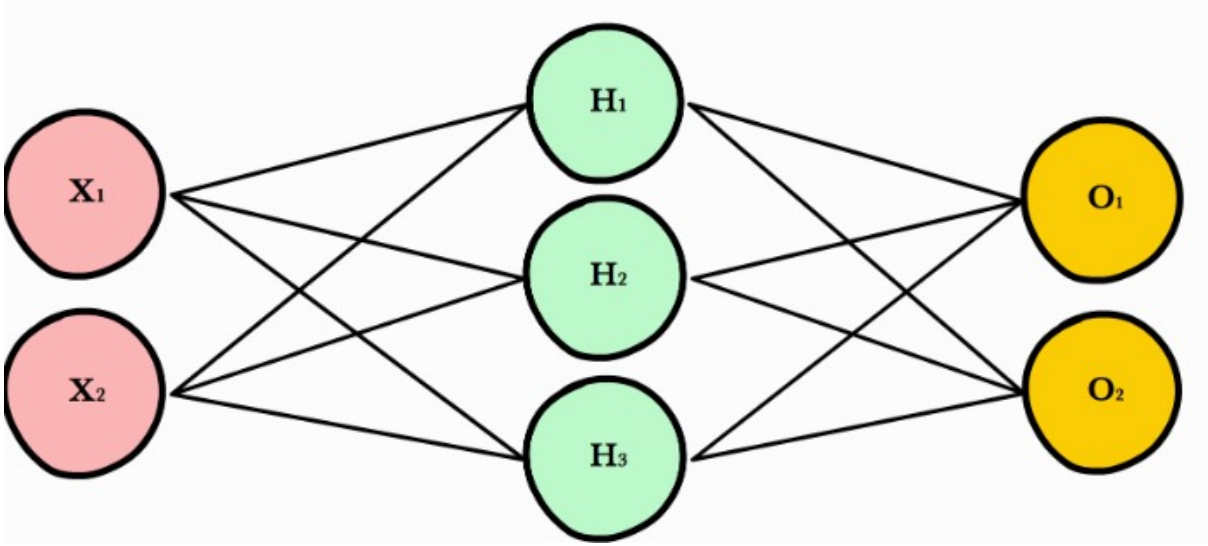


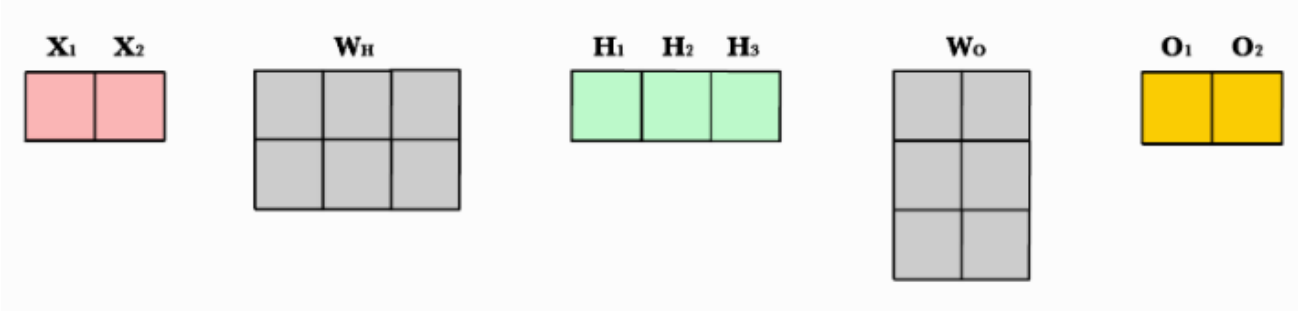


Example from Honglak Lee (NIPS 2010)

Working with Matrices : Forward Computation

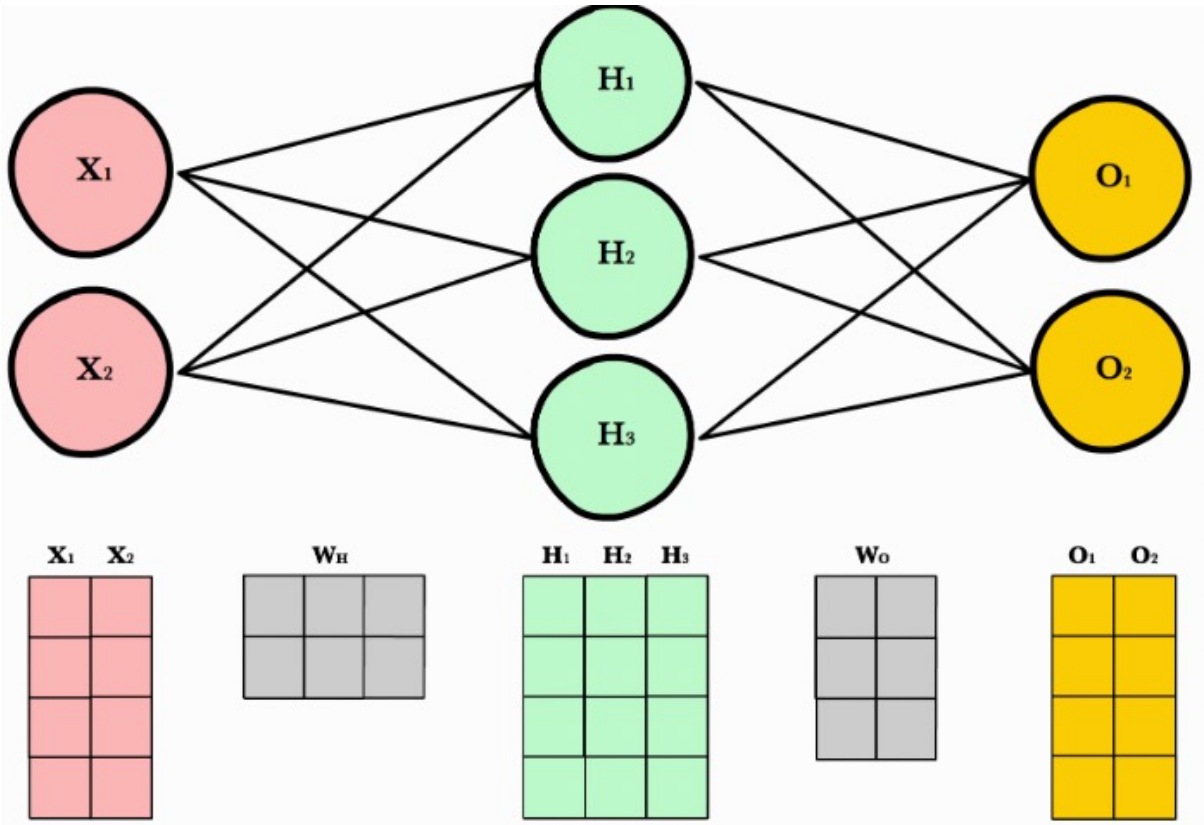
For 1 instance/sample:





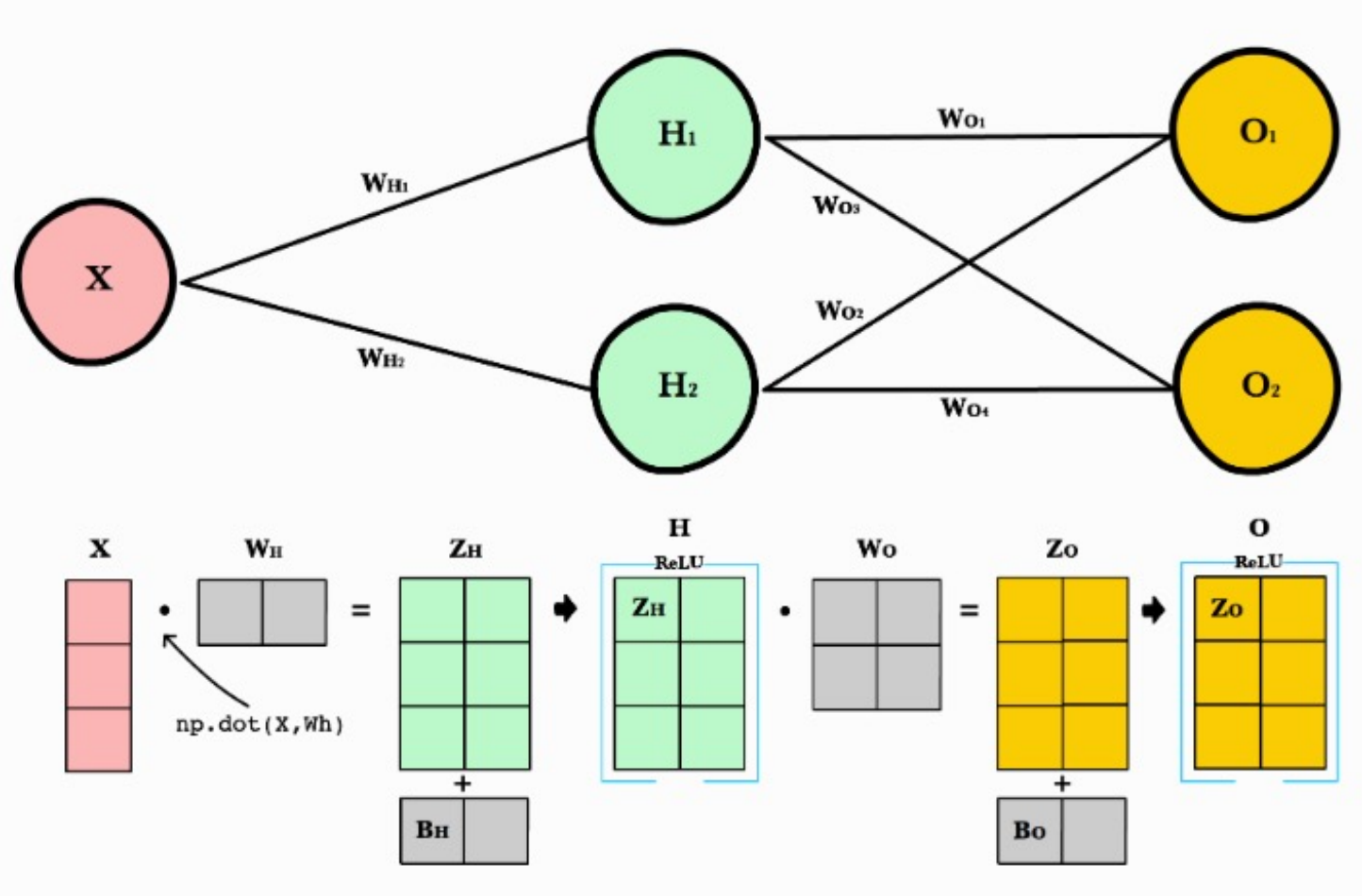
Working with Matrices : Forward Computation

For multiple instances/samples:



Working with Matrices: Forward Computation

For multiple instances/samples with bias:



Applying Neural Network

Expample Problem

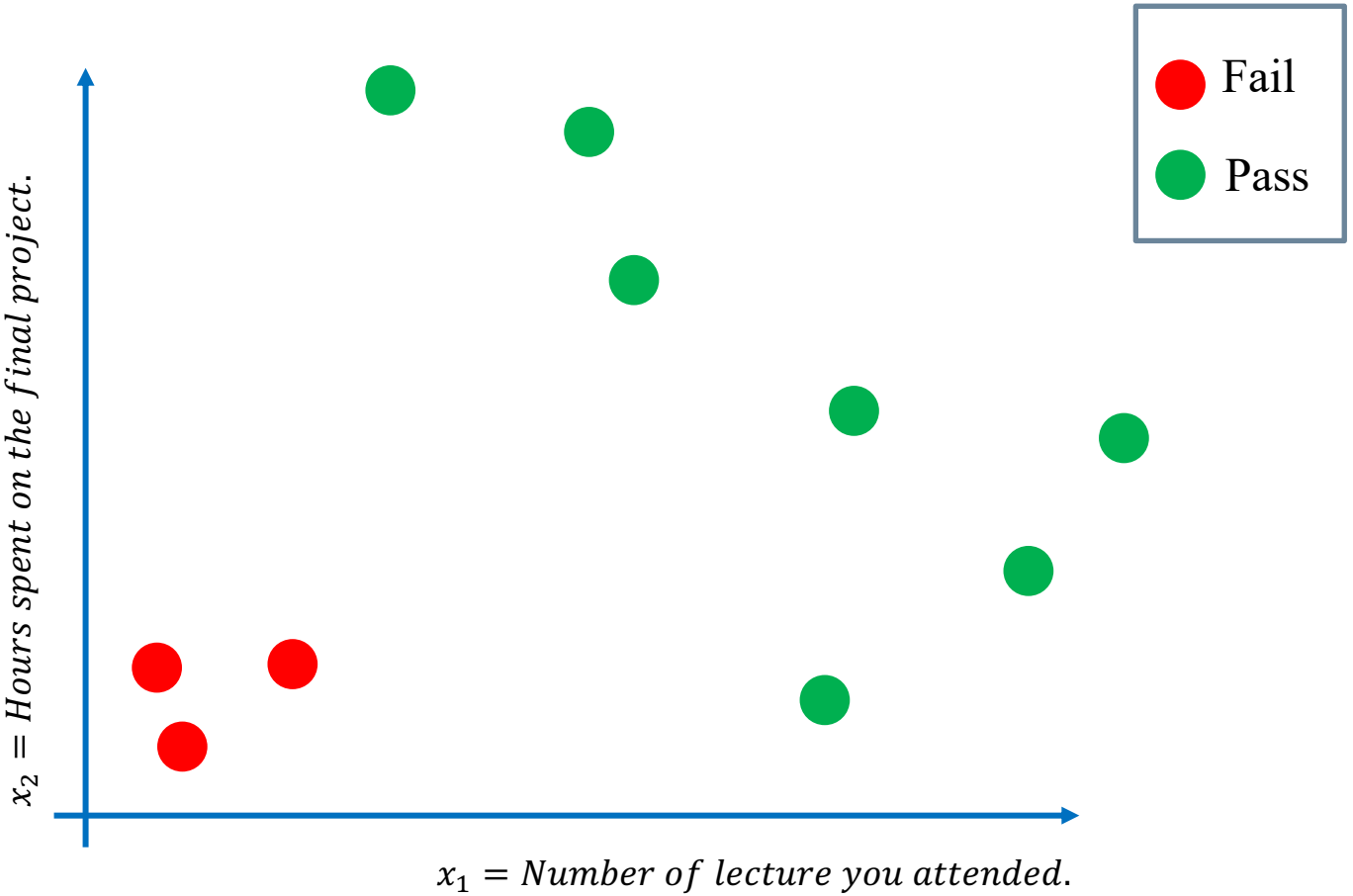
Will I pass this class?

Let's consider **two features** in the model:

$$x_1 = Number\ of\ lecture\ you\ attended.$$

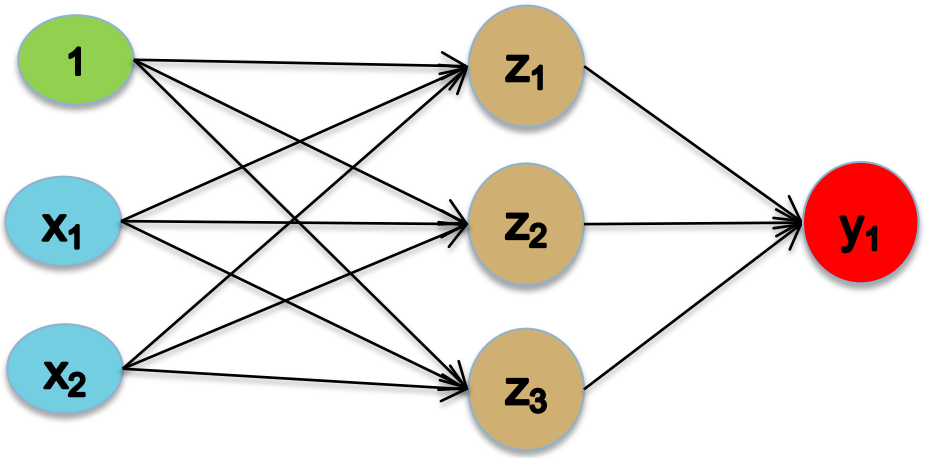
$$x_2 = Hours\ spent\ on\ the\ final\ project.$$

Expample Problem (cont.)



Expample Problem: Quantifying loss

$$x^{(1)} = \begin{matrix} x_1 & x_2 \\ [2 & 3] \end{matrix}$$



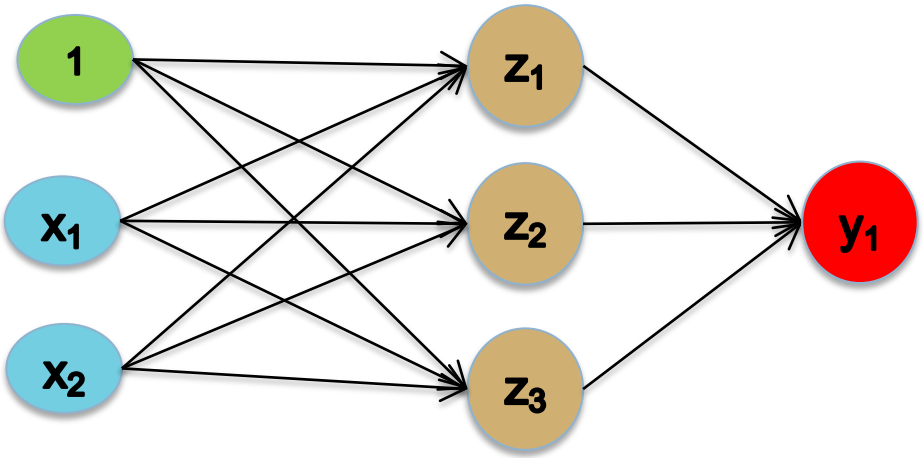
Predicted: 0.9

Actual value: 0

Callculating the loss: $\mathcal{L}(f_\Theta(x^{(1)}), y^{(1)})$

Predicted Actual value

Expample Problem: Quantifying loss



$$X = \begin{matrix} x_1 & x_2 \end{matrix} \begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 6 & 3 \\ 15 & 3 \\ 15 & 4 \\ 16 & 5 \\ 17 & 6 \\ 18 & 3 \\ 8 & 15 \\ 9 & 16 \end{bmatrix}$$

$$f_\Theta(X) = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.1 \\ 0.6 \\ 0.7 \\ 0.7 \\ 0.6 \\ 0.7 \\ 0.5 \\ 0.65 \end{bmatrix} \quad \text{Predicted}$$

$$Y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \text{Actual}$$

Callculating the loss: $\frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(f_\Theta(x^{(i)}), y^{(i)})$

Predicted Actual

Loss function

Binary cross entropy loss

- It can be used with models (classification) that output a probability between 0 and 1.

$$\mathcal{L}(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log f_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - f_\theta(x^{(i)})]$$

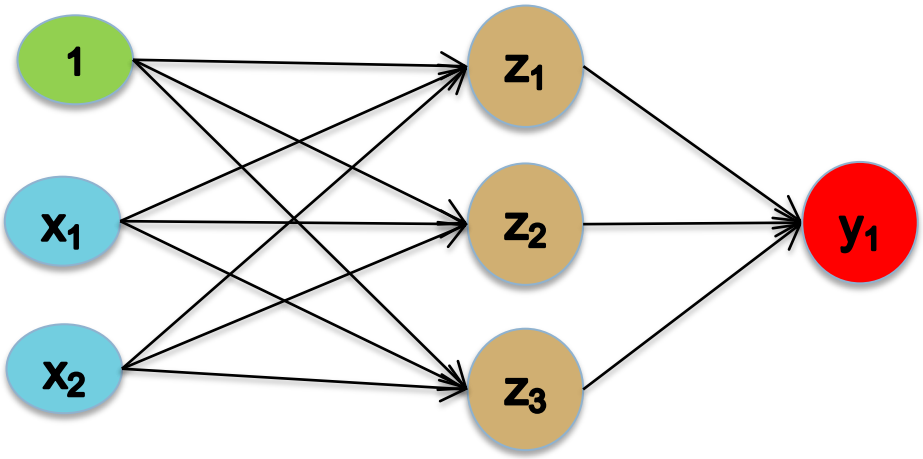
Mean squared error loss

- It can be used with **regression models** that output continuous real numbers.

$$\mathcal{L}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (f_\theta(x^{(i)}) - y^{(i)})^2$$

Binary cross entropy loss

$$X = \begin{matrix} x_1 & x_2 \end{matrix} \begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 6 & 3 \\ 15 & 3 \\ 15 & 4 \\ 16 & 5 \\ 17 & 6 \\ 18 & 3 \\ 8 & 15 \\ 9 & 16 \end{bmatrix}$$

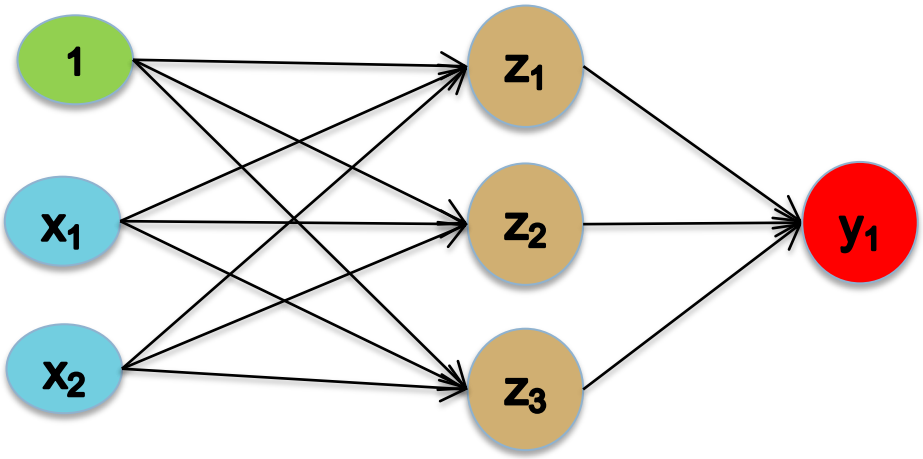


$$f_\Theta(X) = \begin{matrix} \text{Predicted} \end{matrix} \begin{bmatrix} 0.1 \\ 0.2 \\ 0.1 \\ 0.6 \\ 0.7 \\ 0.7 \\ 0.6 \\ 0.7 \\ 0.5 \\ 0.65 \end{bmatrix} \quad Y = \begin{matrix} \text{Actual} \end{matrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\mathcal{L}(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} \underbrace{y^{(i)}}_{\text{Actual}} \log \underbrace{f_\theta(x^{(i)})}_{\text{Predicted}} + \underbrace{(1 - y^{(i)})}_{\text{Actual}} \log(1 - \underbrace{f_\theta(x^{(i)})}_{\text{Predicted}})]$$

Mean squared error loss

$$X = \begin{matrix} x_1 & x_2 \end{matrix} \begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 6 & 3 \\ 15 & 3 \\ 15 & 4 \\ 16 & 5 \\ 17 & 6 \\ 18 & 3 \\ 8 & 15 \\ 9 & 16 \end{bmatrix}$$



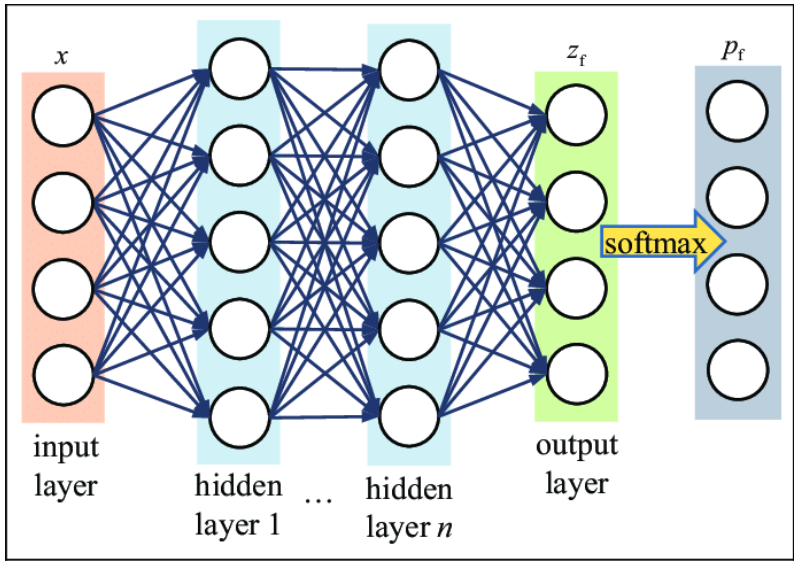
$$f_\Theta(X) = \begin{matrix} \text{Predicted} \\ \begin{bmatrix} 0.1 \\ 0.2 \\ 0.1 \\ 0.6 \\ 0.7 \\ 0.7 \\ 0.6 \\ 0.7 \\ 0.5 \\ 0.65 \end{bmatrix} \end{matrix} \qquad Y = \begin{matrix} \text{Actual} \\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \end{matrix}$$

$$\mathcal{L}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\underbrace{f_\theta(x^{(i)})}_{\text{Predicted}} - \underbrace{y^{(i)}}_{\text{Actual}})^2$$

Softmax

Softmax is a mathematical function that converts a vector of numbers into a vector of probabilities.

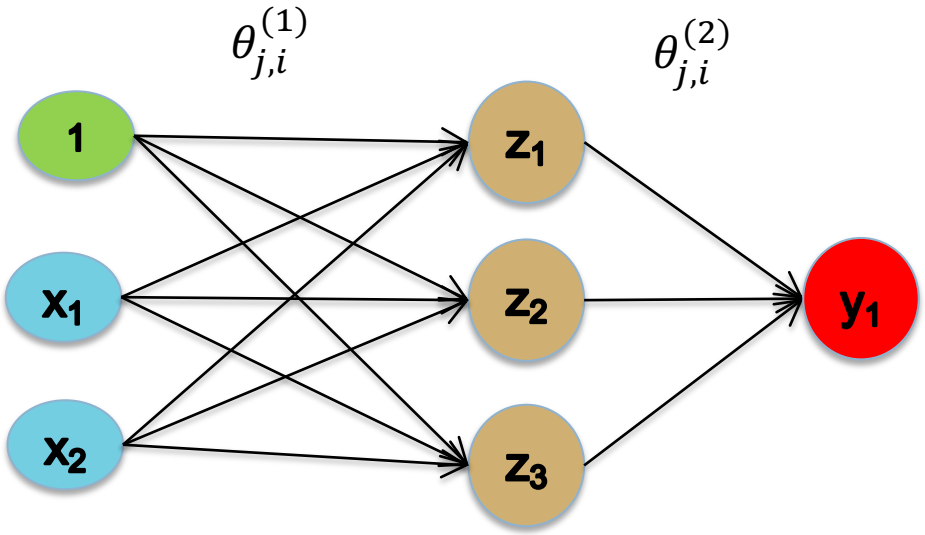
$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$



$$\boldsymbol{z} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} \quad \begin{matrix} 1.1 \\ 2.2 \\ 0.2 \\ -1.7 \end{matrix} \quad s_i = \frac{e^{z_i}}{\sum_l e^{z_l}} \quad \begin{matrix} 0.224 \\ 0.672 \\ 0.091 \\ 0.013 \end{matrix} \quad \boldsymbol{s} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix}$$

Backpropagation (Training)

Loss Optimization



$$X = \begin{matrix} x_1 & x_2 \end{matrix} \begin{bmatrix} 2 & 3 \\ 1 & 2 \\ 6 & 3 \\ 15 & 3 \\ 15 & 4 \\ 16 & 5 \\ 17 & 6 \\ 18 & 3 \\ 8 & 15 \\ 9 & 16 \end{bmatrix}$$

Predicted Actual

$$f_\Theta(X) = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.1 \\ 0.6 \\ 0.7 \\ 0.7 \\ 0.6 \\ 0.7 \\ 0.5 \\ 0.65 \end{bmatrix} \quad Y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

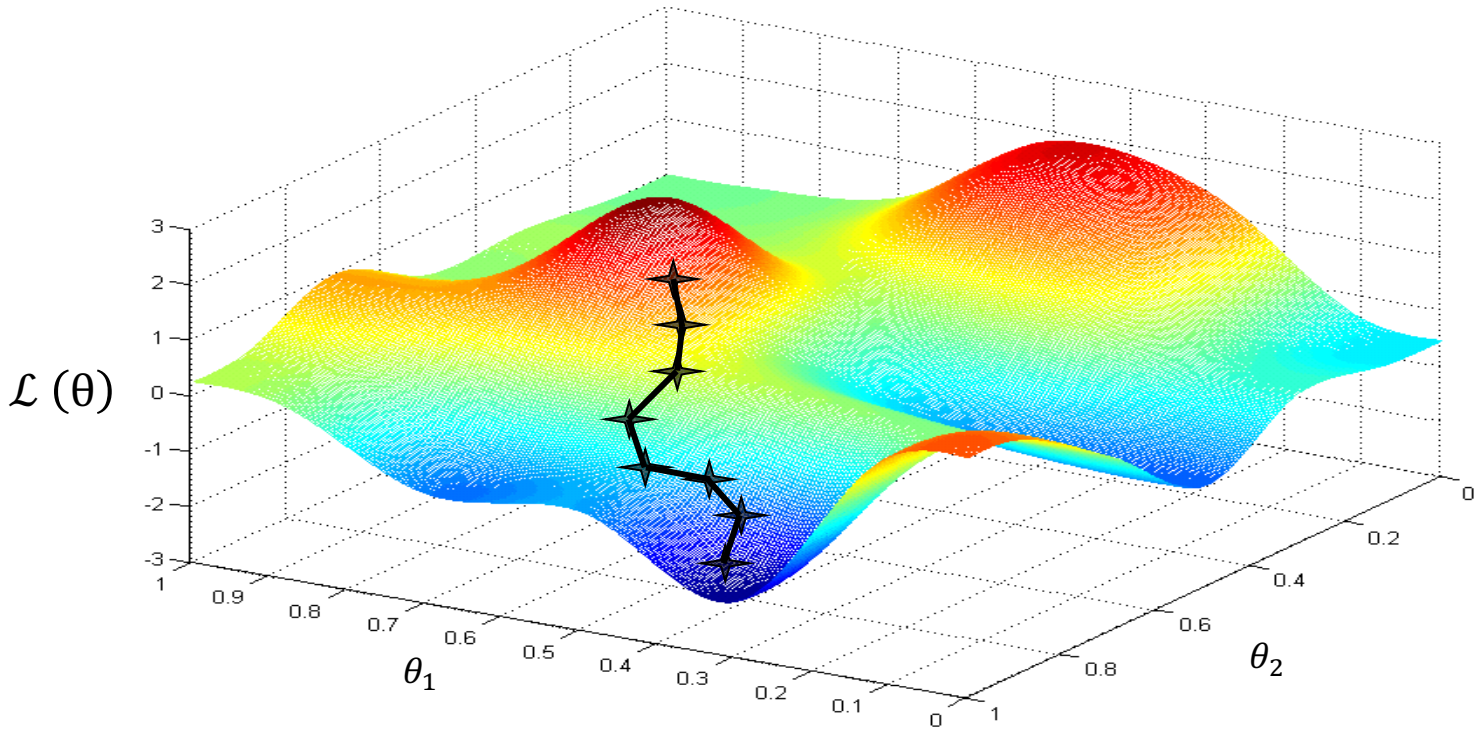
We want to find the network weights that achives the lowest loss.

$$\Theta^* = arg\min_\Theta \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f_\Theta(x^{(i)}), y^{(i)})$$

Predicted Actual

$$\Theta^* = arg\min_\Theta \mathcal{L}(\Theta) \qquad where\ \Theta = \{\Theta^{(1)}, \Theta^{(2)}, \dots.\}$$

Intuition picture of gradient descent



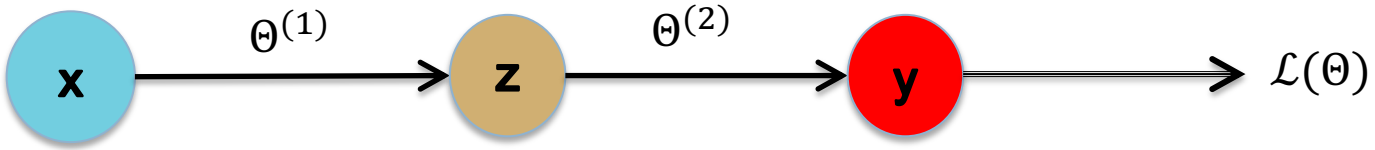
Starting at some points on the surface of the lost fucntion.
Take a step in the direction of steepest descent.
Each step changes parameter θ to reduce L(θ) until end up at a local minimum.

Gradient descent

Algorithm

1. Initilize weight randomly
2. For Loop (until convergence)
    1. Compute gradient $\frac{\partial \mathcal{L}(\Theta)}{\partial \Theta}$
    2. Update weights. $\Theta = \Theta - \alpha \frac{\partial \mathcal{L}(\Theta)}{\partial \Theta}$
3. Return weights

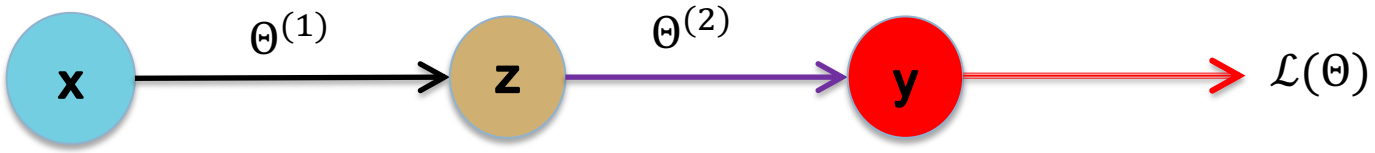
Computing Gradient: Backpropagation



$$\frac{\partial \mathcal{L}(\Theta)}{\partial \Theta^{(2)}} \qquad \frac{\partial \mathcal{L}(\Theta)}{\partial \Theta^{(1)}}$$

$where\ \Theta = \{\Theta^{(1)}, \Theta^{(2)}\}$

Computing Gradient: Backpropagation

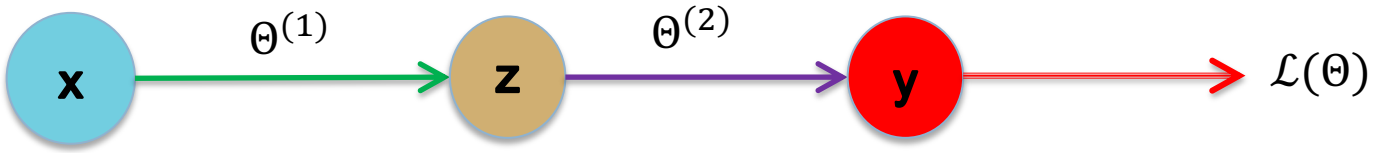


$$\frac{\partial \mathcal{L}(\Theta)}{\partial \Theta^{(1)}} = ?$$

$$\frac{\partial \mathcal{L}(\Theta)}{\partial \Theta^{(2)}} = \frac{\partial \mathcal{L}(\Theta)}{\partial y} * \frac{\partial y}{\partial \Theta^{(2)}}$$

$$where\ \Theta = \{\Theta^{(1)}, \Theta^{(2)}\}$$

Computing Gradient: Backpropagation



$$\frac{\partial \mathcal{L}(\Theta)}{\partial \Theta^{(1)}} = \frac{\partial \mathcal{L}(\Theta)}{\partial y} * \frac{\partial y}{\partial z} * \frac{\partial z}{\partial \Theta^{(1)}}$$

$$\frac{\partial \mathcal{L}(\Theta)}{\partial \Theta^{(2)}} = \frac{\partial \mathcal{L}(\Theta)}{\partial y} * \frac{\partial y}{\partial \Theta^{(2)}}$$

Repeat this calculation for **every weight in the network** using gradient from latter layer.

$where\ \Theta = \{\Theta^{(1)}, \Theta^{(2)}\}$

A matrix example

Forward computation

$$z_1 = XW_1$$
$$h_1 = \text{ReLU}(z_1)$$
$$\hat{y} = h_1 W_2$$
$$L = ||\hat{y}||_2^2$$

Backpropagation

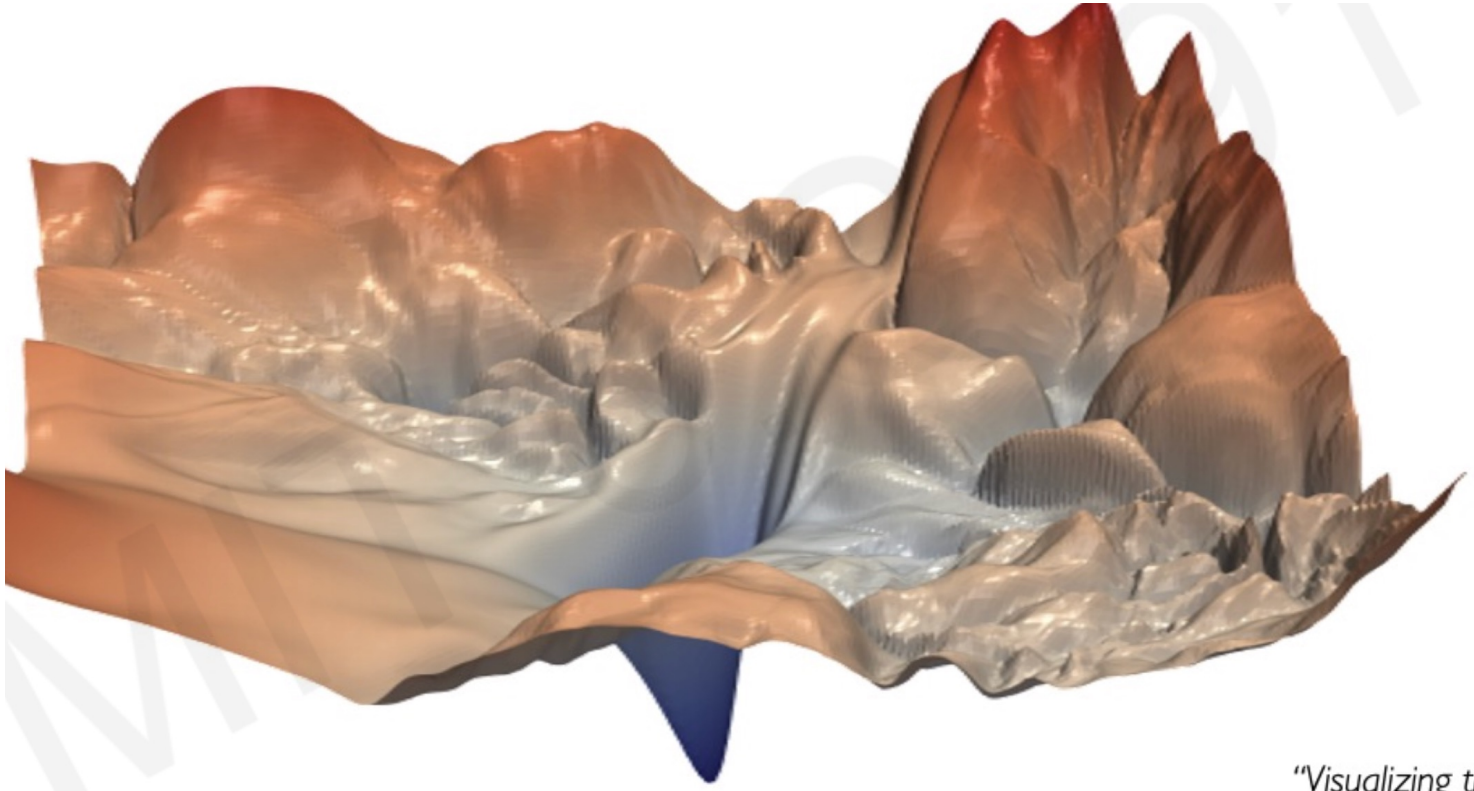
$$\frac{\partial L}{\partial \hat{y}} = 2\hat{y}$$
$$\boxed{\frac{\partial L}{\partial W_2} = h_1^\top \frac{\partial L}{\partial \hat{y}}}$$
$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial \hat{y}} W_2^\top$$
$$\frac{\partial h_1}{\partial z_1} = \frac{\partial L}{\partial h_1} \circ I[h_1 > 0]$$
$$\boxed{\frac{\partial z_1}{\partial W_1} = x^\top \frac{\partial h_1}{\partial z_1}}$$

Exmaple of impementation

```
import numpy as np

# forward prop
z_1 = np.dot(X, W_1)
h_1 = np.maximum(z_1, 0)
y_hat = np.dot(h_1, W_2)
L = np.sum(y_hat**2)

# backward prop
dy_hat = 2.0*y_hat

dW2 = h_1.T.dot(dy_hat)

dh1 = dy_hat.dot(W_2.T)

dz1 = dh1.copy()
dz1[z1 < 0] = 0

dW1 = X.T.dot(dz1)
```

Visualizing the loss landscape

Training Neural Network is difficult.



"Visualizing the loss landscape of neural nets". Dec 2017.

Gradient descent

Algorithm

1. Initilize weight randomly
2. For Loop (until convergence)
    1. Compute gradient $\frac{\partial \mathcal{L}(\Theta)}{\partial \Theta}$
    2. Update weights. $\Theta = \Theta - \alpha \frac{\partial \mathcal{L}(\Theta)}{\partial \Theta}$
3. Return weights

Learning rate(lr)

- Small lr - converges slowly and get stuck in a local minima.
- Large lr – becomes unstable and diverge.
- How to find a stable lr?

Learning rate

Idea 1: Try different learning rate and find a stable one.

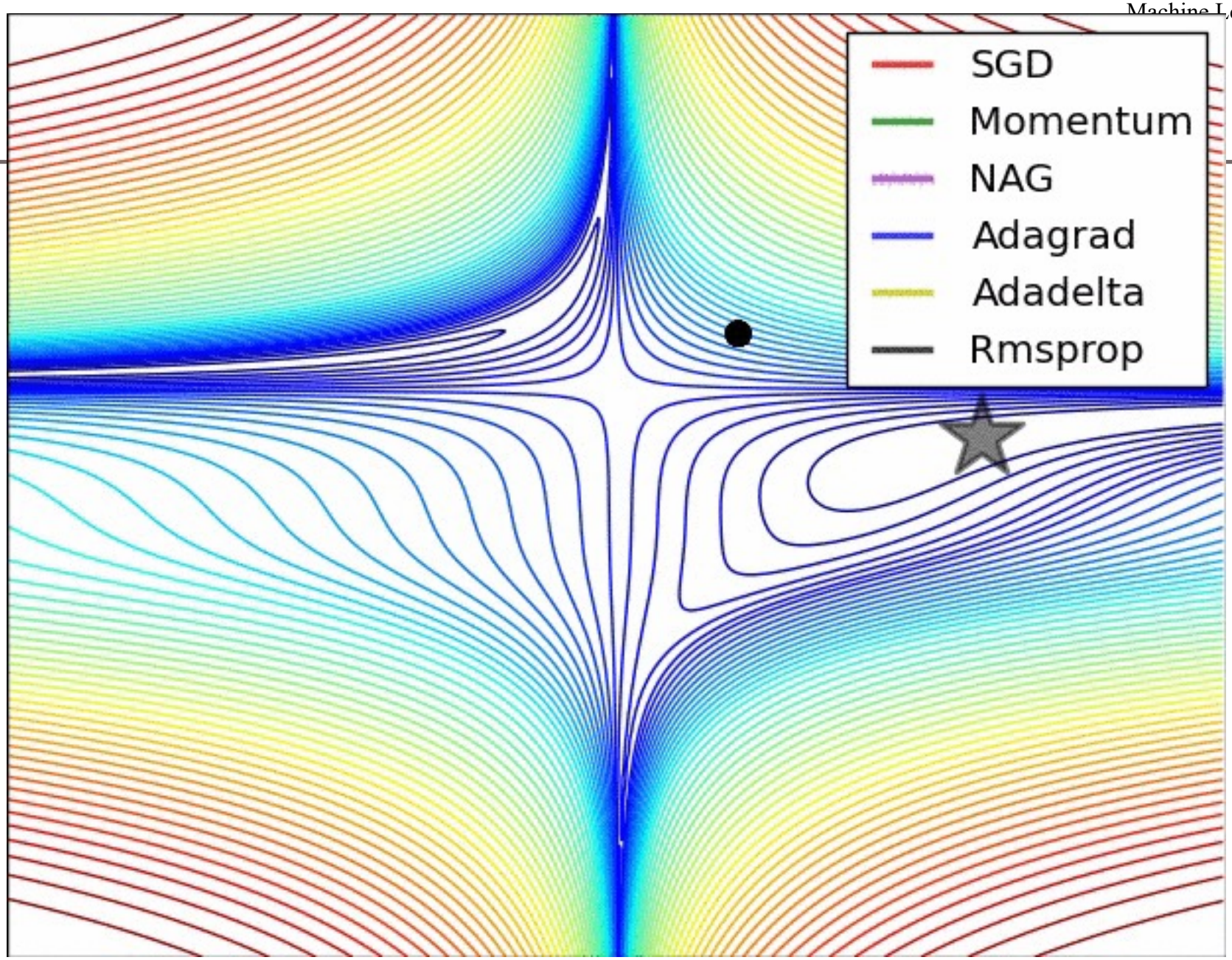
Idea 2: Design a adaptive learning rate approach (more smarter).

Algorithms:

SGD, Adam, Adadelta, Adagrad, RMSProb

Learning rate are no longer fixed.
Can be larger and smaller depending on:

- How large gradient is
- Size of perticular weight
- How fast learning is happening.
- etc


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop

Training

Gradient descent

Algorithm

1. Initilize weight randomly
2. For Loop (until convergence)
    1. Compute gradient $\frac{\partial \mathcal{L}(\Theta)}{\partial \Theta}$
    2. Update weights. $\Theta = \Theta - \alpha \frac{\partial \mathcal{L}(\Theta)}{\partial \Theta}$
3. Return weights

Can be very computaionaly intensive to compute (use all samples).

Stochastic Gradient descent (SGD)

Algorithm

1. Initilize weight randomly
2. For Loop (until convergence)
    1. Pick a single sample i;
    2. Compute gradient $\frac{\partial \mathcal{L}_i(\Theta)}{\partial \Theta}$
    3. Update weights. $\Theta = \Theta - \alpha \frac{\partial \mathcal{L}(\Theta)}{\partial \Theta}$
3. Return weights

Easy to compute but very noisy (stochastic).

Mini-batch SGD

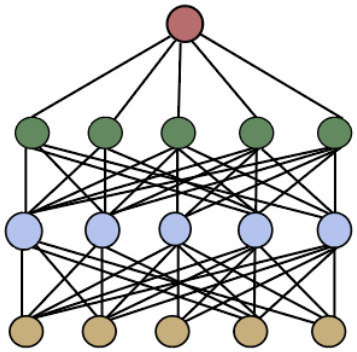
Algorithm

1. Initilize weight randomly
2. For Loop (until convergence)
    1. Pick batch of K samples ;
    2. Compute gradient $\frac{\partial \mathcal{L}(\Theta)}{\partial \Theta} = \frac{1}{B}\sum_{i=1}^{K} \frac{\partial \mathcal{L}_i(\Theta)}{\partial \Theta}$
    3. Update weights. $\Theta = \Theta - \alpha \frac{\partial \mathcal{L}(\Theta)}{\partial \Theta}$
3. Return weights

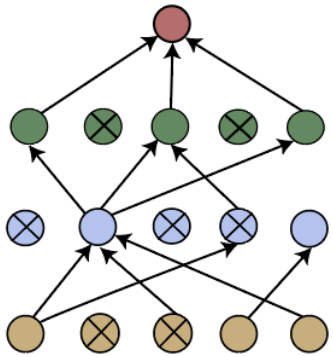
Fast to compute and much better estimate of the true gradient.

Overtiffting issue

- Regularization
  - Use dropout: it randomly select some neurons output (50%) set to 0.

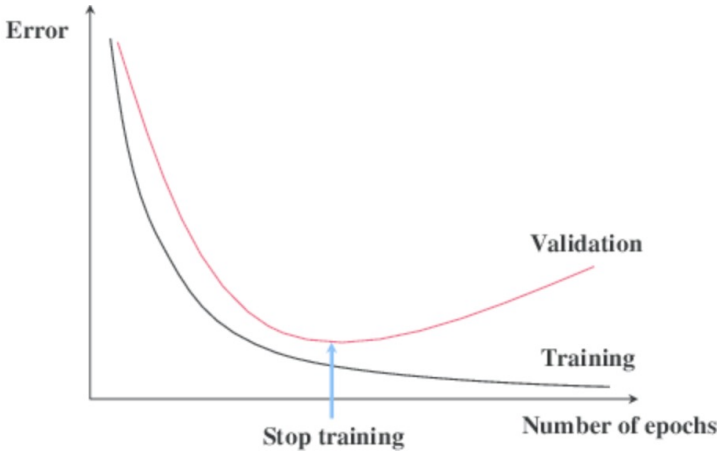


Standard Neural Network



After Applying Dropout

- Early stopping



Implenentation

Practice

Neural network from scratch

```
# training iteration
for iter in range(iter_n):

    loss = 0
    acc = 0
    #for each sample
    for i, x in enumerate(x_train1):

        #forward computation
        z = np.dot(x, W_1)
        h1 = sigmoid(z)

        one2 = np.ones(1)
        h1_one = np.append(h1,one2)
        predicted = softmax(np.dot(h1_one,W_2))

        #accuracy
        p = np.argmax(predicted)
        yhat = np.argmax(y_train1[i])
        if p == yhat:
            acc += 1


        #backpropagation
        loss += 1/2*sum((predicted - y_train1[i])**2)

        delta2 = (predicted - y_train1[i]).reshape(1,outputsize)
        W2_df = np.dot(h1_one.reshape(hn+1,1), delta2)

        delta1 = np.dot(delta2, W_2.T)[0][0:hn]
        der = (delta1 * der_sigmoid(h1))

#         print(x.shape)
#         print(der.shape)
        W1_df = np.dot(x.reshape(inputsize+1,1),der.reshape(hn,1).T)

        #update with SGD
        W_1 -= lr * W1_df
        W_2 -= lr * W2_df

    # Print the loss every 10 epochs
    if iter % 10 == 0:
        print(iter, "loss: " ,loss, "acc: ", acc/len(x_train))
    iter_loss.append(loss)
```

Practice

Neural network using Tensorflow/Keras

```
# Define the model architecture
model = keras.Sequential(
    [
        layers.Flatten(input_shape=(inputsize,)),
        layers.Dense(128, activation="sigmoid"),
        layers.Dense(outputsize, activation="softmax"),
    ]
)

# Compile the model
model.compile(
    optimizer="sgd",
    loss="mean_squared_error",
    metrics=["accuracy"],
)

# Train the model
model.fit(x_train, y_train, epochs=500, validation_split=0.2)
```

```
Epoch 495/500
5/5 [==============================] - 0s 8ms/step - loss: 0.2500 - accurac
y: 0.5750
```

- Thank you!