Introduction to Deep Learning
Chapter 2: Neural Networks

Pao-Ann Hsiung

National Chung Cheng University

Contents

Contents

Basics of Neural Networks

線性
- The Perceptron and its Learning Rule (Frank Rosenblatt, 1957)
- Adaptive Linear Neuron and Delta Rule (Widrow & Hoff, 1960)

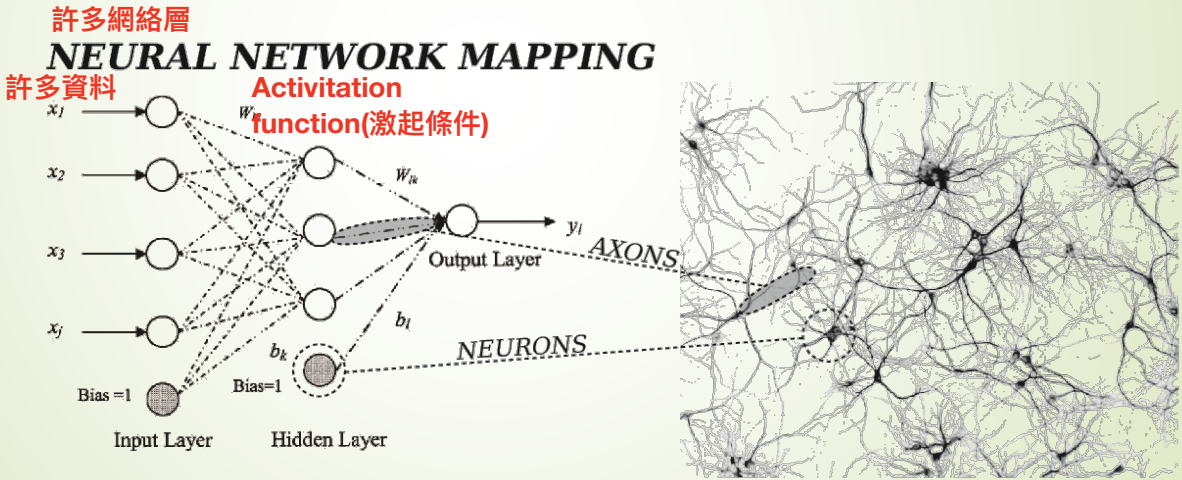
非線性
- Logistic Regression and Gradient Descent

Basics of Neural Networks

- The Perceptron and its Learning Rule (Frank Rosenblatt, 1957)
- Adaptive Linear Neuron and Delta Rule (Widrow & Hoff, 1960)
- Logistic Regression and Gradient Descent

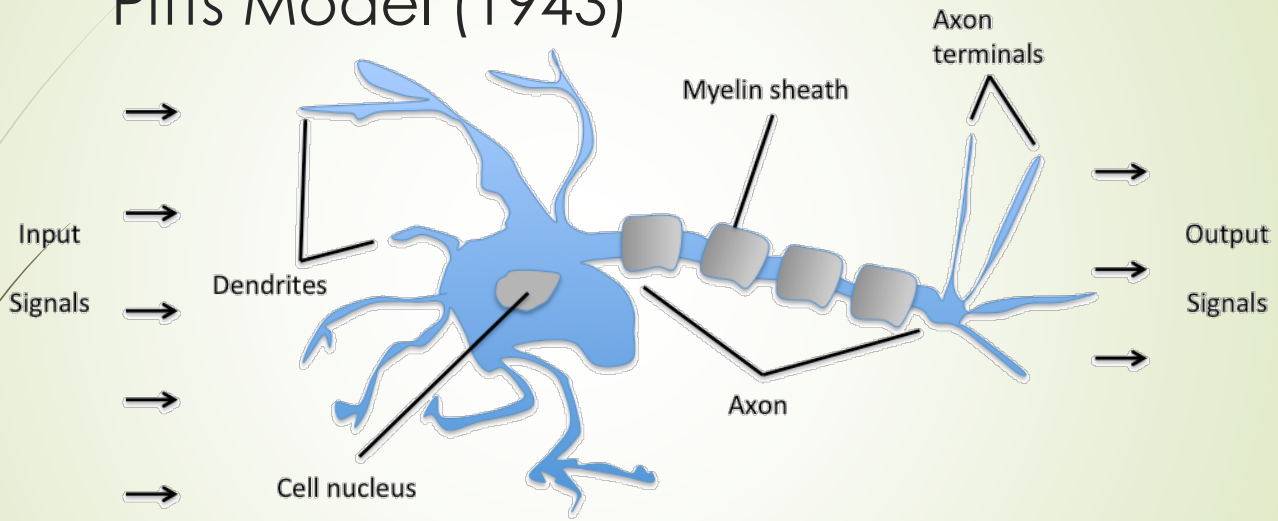
Basics of Neural Networks

- Biologically inspired (akin to the neurons in a brain)



http://rpi-cloudreassembly.transvercity.net/2012/11/04/neural-network-mapping-analysis-from-above/

Artificial Neurons and the McCulloch-Pitts Model (1943)

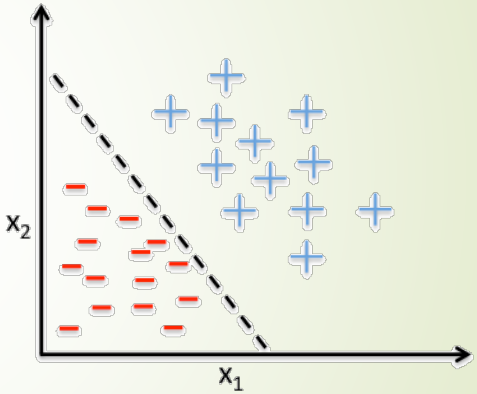


Schematic of a biological neuron.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, 1943.

Frank Rosenblatt's Perceptron (1957)

- Supervised learning
- Single-layer
- Binary linear classifier
- To predict to which of 2 possible categories, a certain data point belongs on a set of input variables

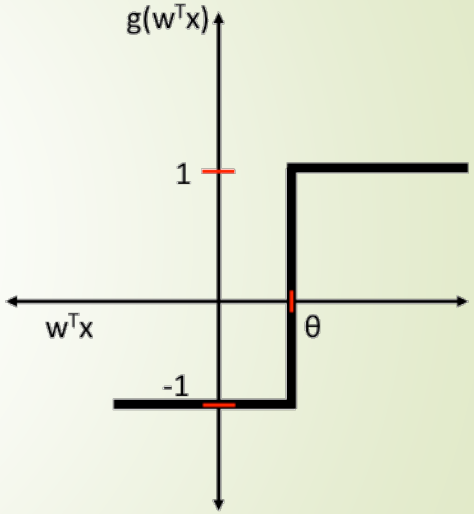


Example of a linear decision boundary for binary classification.

F. Rosenblatt. The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, 1957.

Unit Step Function

- Positive class: +1
- Negative class: −1
- Activation function: g(z) = 1 if z ≥ θ; −1 o/w
  - where z is a linear combination of input values x and weights w, that is, 線性組合
    $z = w_1x_1 + w_2x_2 + \ldots + w_mx_m = \sum_{j=1}^{m} x_j w_j = \mathbf{w}^T\mathbf{x}$
  - $\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$ is the weight vector 權重
  - $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ is an m-dimensional sample 資料 from the training data set

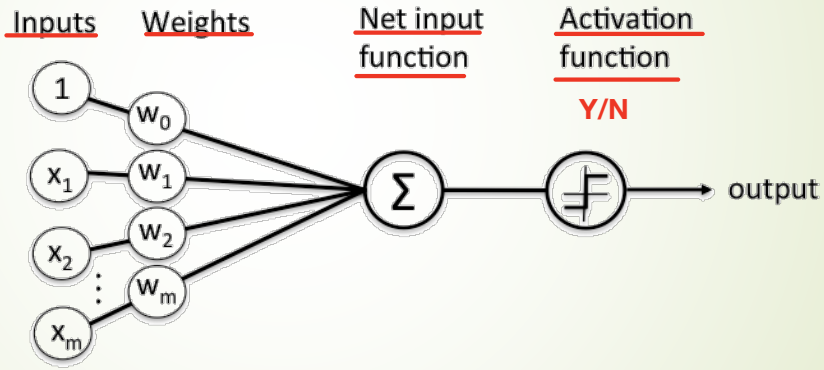


Unit step function.

Simplified Unit Step Function

- To simplify calculations, move θ to the origin such that the activation function becomes
- $g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$

Perceptron Learning Rule



Schematic of Rosenblatt's perceptron.

Rosenblatt's Perceptron Learning Rule

- Initialize the weights to 0 or small random numbers.
- For each training sample $x^{(i)}$:
  - Calculate the output value $y^{(i)} = g(z^{(i)})$
  - Update the weights as follows:

逐漸改變權重 $w_j := w_j + \eta(y'^{(i)} - y^{(i)})x_j^{(i)}$

小數 樣本 預測與實際差異

where η is the learning rate, $0.0 < \eta < 1$,
$y'^{(i)}$ is is the actual true class label, and
$y^{(i)}$ is the predicted class label.

Perceptron Rule in Python

- Classify the flowers in the **Iris** dataset using the perceptron rule
- Iris dataset from UCI Machine Learning Repository

More complete version:
https://github.com/rasbt/mlxtend/blob/master/mlxtend/classifier/perceptron.py

```
import numpy as np

class Perceptron(object):
    def __init__(self, eta=0.01, epochs=50):
        self.eta = eta
        self.epochs = epochs

    def train(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.errors_ = []
        for _ in range(self.epochs):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] +=  update * xi
                self.w_[0] +=  update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Classify 2 flower species: Setosa and Versicolor using sepal length and petal length

- import pandas as pd
- df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=None)

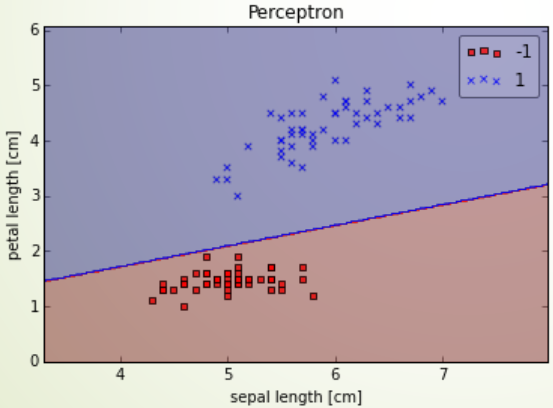
- # setosa and versicolor
- y = df.iloc[0:100, 4].values
- y = np.where(y == 'Iris-setosa', -1, 1)

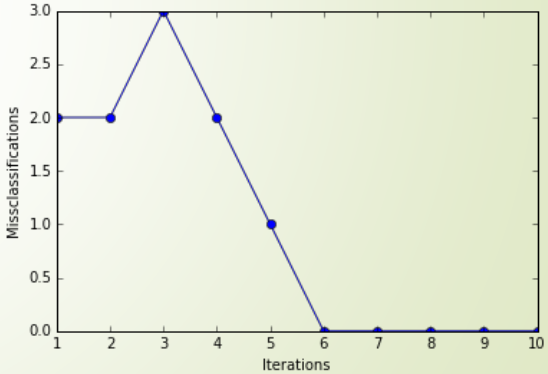
- # sepal length and petal length
- X = df.iloc[0:100, [0,2]].values

```
%matplotlib inline
import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions

ppn = Perceptron(epochs=10, eta=0.1)

ppn.train(X, y)
print('Weights: %s' % ppn.w_)
plot_decision_regions(X, y, clf=ppn)
plt.title('Perceptron')
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.show()

plt.plot(range(1, len(ppn.errors_)+1), ppn.errors_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Misclassifications')
plt.show()
```

Results

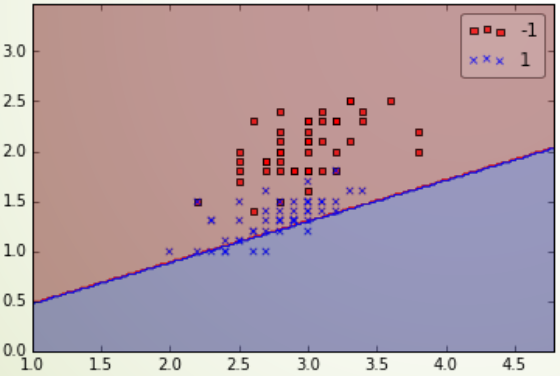
- Perceptron converges after 6th iteration
- Weights: [-0.4 -0.68 1.82]

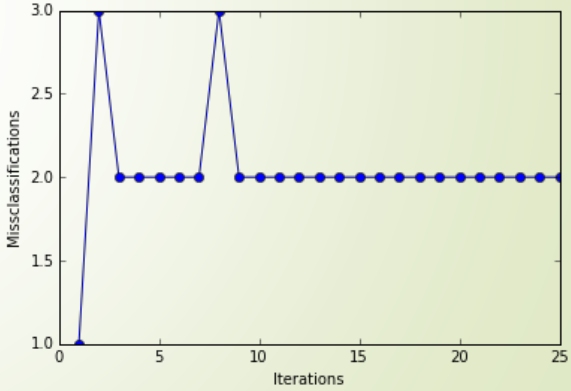




Problems with Perceptron

- The 2 classes must be separable by a linear hyperplane
- If not, then the perceptron algorithm does NOT converge!
- Example





http://sebastianraschka.com/Articles/2015_singlelayer_neurons.html#problems-with-perceptrons

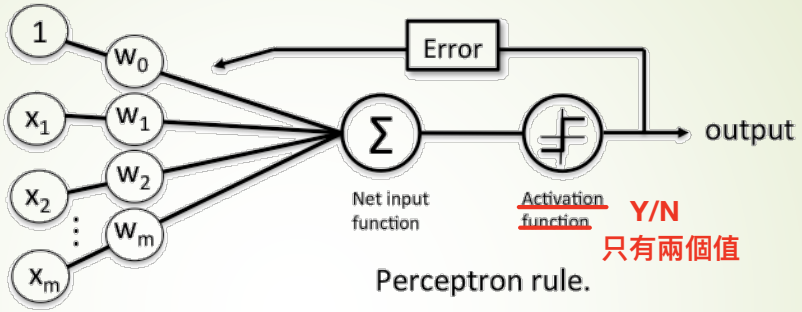
Basics of Neural Networks

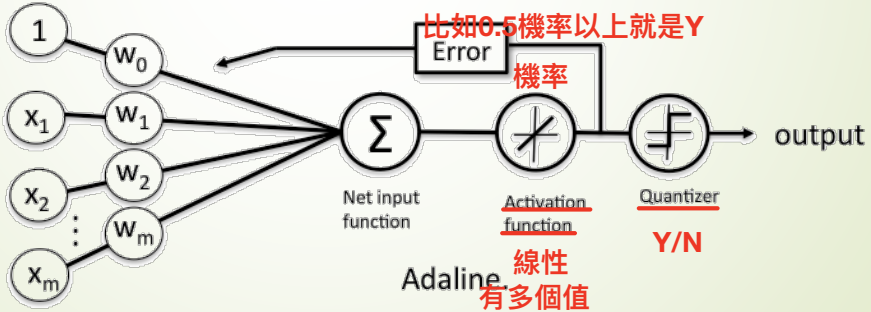
- The Perceptron and its Learning Rule (Frank Rosenblatt, 1957)
- Adaptive Linear Neuron and Delta Rule (Widrow & Hoff, 1960)
- Logistic Regression and Gradient Descent

Adaptive Linear Neurons and the Delta Rule (1960)

- Bernard Widrow and Tedd Hoff proposed Adaptive Linear Neurons (Adaline)
- Linear activation function: $g(z) = z$.
- It is differentiable, so we can define a cost function and minimize it!

B. Widrow et al. Adaptive ”Adaline” neuron using chemical ”memistors”. Number Technical Report 1553-2. Stanford Electron. Labs., Stanford, CA, October 1960.


1
w_0
x_1
w_1
x_2
w_2
w_m
x_m
Error
Σ
output
Net input function
Activation function
Y/N
只有兩個值
Perceptron rule.

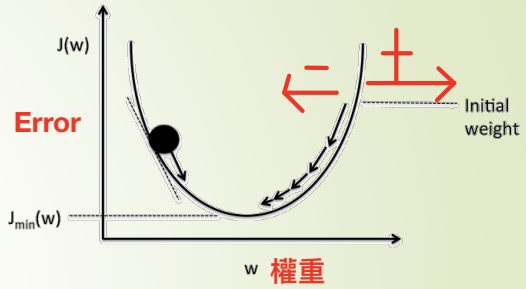

比如0.5機率以上就是Y
1
w_0
x_1
w_1
x_2
w_2
w_m
x_m
Error
機率
Σ
output
Net input function
Activation function
Quantizer
Y/N
線性
有多個值
Adaline.

Gradient Descent

https://youtu.be/b4Vyma9wPHo

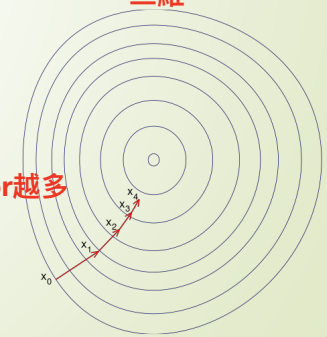
- **Gradient Descent**
  - A first-order iterative optimization algorithm for finding the minimum of a function
  - Take steps proportional to the **negative of the gradient** of the function at the current point

單維



Schematic of gradient descent.

二維



越外面error越多

由外而進

Gradient Descent of a Cost Function

- Cost function: sum of squared errors (SSE)
  - $J(w) = \frac{1}{2}\sum_i (y'^{(i)} - y^{(i)})^2$
- To minimize SSE, we can use "gradient descent" 降低SSE
- A step in the opposite direction of gradient 反向

$$\Delta w = -\alpha \nabla J(w)$$ 由外而內慢慢前進

where α is the learning rate, $0 < \alpha < 1$

- Thus, we need to compute the partial derivative of the cost function for each weight in the weight vector,

$$\Delta w_j = -\alpha \frac{\partial J}{\partial w_j}$$ 找出每個向量的權重

Partial derivative computations

Derivative x (SSE)

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i (y'^{(i)} - y^{(i)})^2$$

$$= \frac{1}{2}\sum_i \frac{\partial}{\partial w_j}(y'^{(i)} - y^{(i)})^2$$

$$= \frac{1}{2}\sum_i 2(y'^{(i)} - y^{(i)})\frac{\partial}{\partial w_j}(y'^{(i)} - y^{(i)})$$

$$= \sum_i (y'^{(i)} - y^{(i)})\frac{\partial}{\partial w_j}(y'^{(i)} - \sum_j w_j x_j^{(i)})$$

$$= \sum_i (y'^{(i)} - y^{(i)})(-x_j^{(i)})$$

Loss

$z^{(i)} = y(y'^{(i)}) = \hat{y}$

Updating weights using gradient descent

- A step in gradient descent:
  - $\Delta w_j = -\alpha \frac{\partial J}{\partial w_j} = -\alpha \sum_i \left(y'^{(i)} - y^{(i)}\right)\left(-x_j^{(i)}\right) = \alpha \sum_i (y'^{(i)} - y^{(i)}) x_j^{(i)}$
- Update weight vector:
  - 新 舊
  - $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$
- Differences with the perceptron rule
  - The output $y^{(i)}$ is a real number, not a class label as in perceptron learning rule.
  - Weight update is based on "all samples in the training set" (Batch GD)

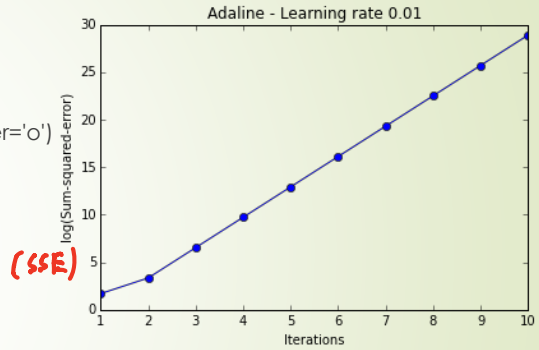
Gradient Descent Rule in Python

```
import numpy as np

class AdalineGD(object):
    def __init__(self, alpha=0.01, epochs=50):
        self.alpha = alpha
        self.epochs = epochs
    def train(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []
        for i in range(self.epochs):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.alpha * X.T.dot(errors)
            self.w_[0] += self.alpha * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self
```

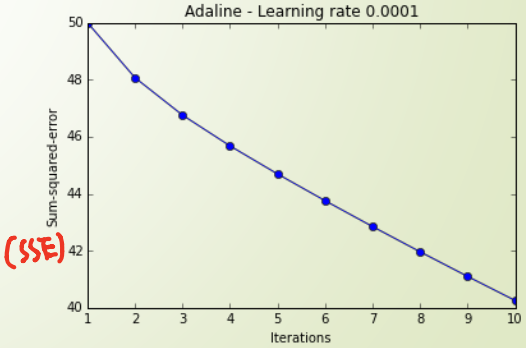
```
def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]
    def activation(self, X):
        return self.net_input(X)
    def predict(self, X):
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

Different Learning Rates: 0.01 vs. 0.0001

```
ada = AdalineGD(epochs=10, alpha=0.01).train(X, y)
plt.plot(range(1, len(ada.cost_)+1), np.log10(ada.cost_), marker='o')
plt.xlabel('Iterations')
plt.ylabel('log(Sum-squared-error)')
plt.title('Adaline - Learning rate 0.01')
plt.show()
```

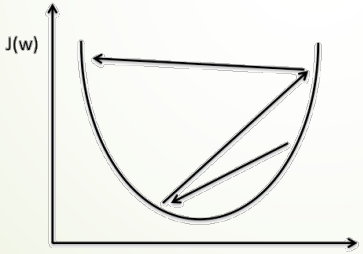



```
ada = AdalineGD(epochs=10, alpha=0.0001).train(X, y)
plt.plot(range(1, len(ada.cost_)+1), ada.cost_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Sum-squared-error')
plt.title('Adaline - Learning rate 0.0001')
plt.show()
```



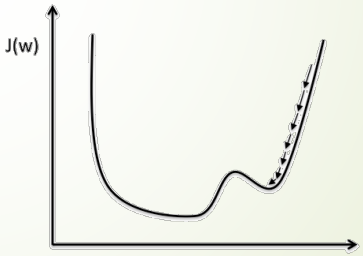
Problems with learning rates

- If the learning rate is TOO LARGE gradient descent will overshoot the minima and diverge.
- If the learning rate is too small, gradient descent will require too many epochs to converge and can become trapped in local minima more easily.



Too large

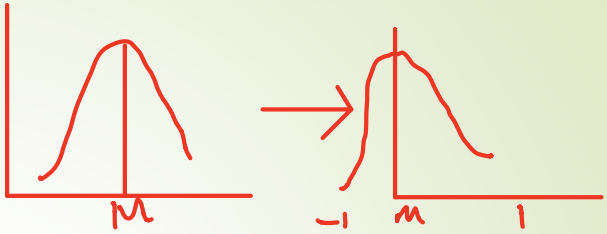
Large learning rate: Overshooting.



Too small

Small learning rate: Many iterations until convergence and trapping in local minima.

Feature Scaling



聚集

- If features are scaled on the same scale, gradient descent converges faster and prevents weights from becoming too small (weight decay).

值的範圍可能相差很大會影響結果

- Common way for feature scaling

原始資料特徵-平均資料特徵

$$x_{j,std} = \frac{x_j - \mu_j}{\sigma_j}$$

where μ_j is the sample mean of the feature x_j and σ_j the standard deviation.

- After standardization, the features will have unit variance and centered around mean zero.

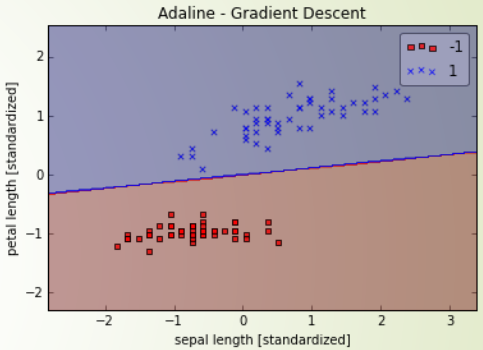
Feature Scaling in Python

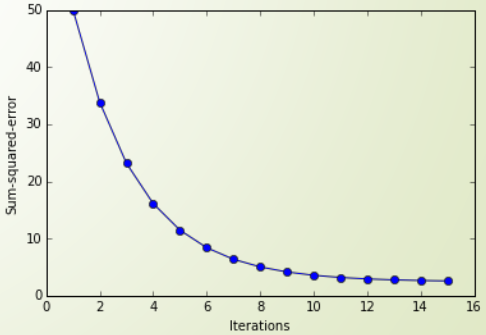
```
# standardize features
X_std = np.copy(X)
X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

兩個特徵

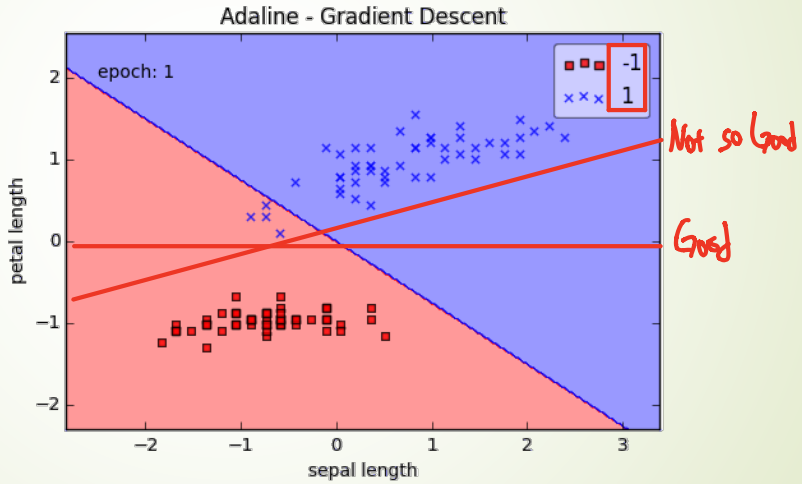
Using feature scaling in Adaline

```
%matplotlib inline
import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions

ada = AdalineGD(epochs=15, eta=0.01)

ada.train(X_std, y)
plot_decision_regions(X_std, y, clf=ada)
plt.title('Adaline - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.show()

plt.plot(range(1, len( ada.cost_)+1), ada.cost_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Sum-squared-error')
plt.show()
```





Adaline Gradient Descent



Online Learning via Stochastic Gradient Descent

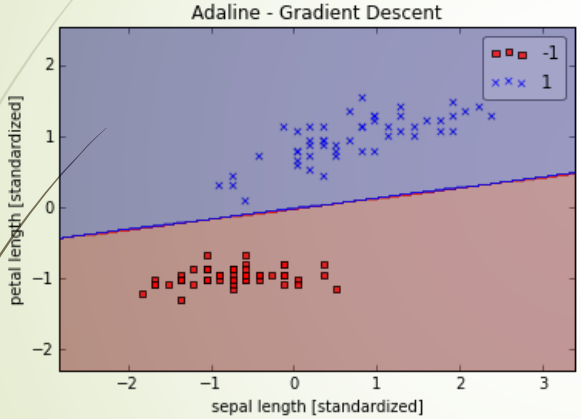
- Batch Gradient Descent (BGD)
  - Cost function is minimized based on the complete training dataset (all samples)
- Stochastic Gradient Descent (SGD)
  - Weights are incrementally updated after each individual training sample
  - Converges faster than BGD since weights are updated immediately after each training sample
  - Computationally more efficient, especially for large datasets
- Mini-batch Gradient Descent (MGD)
  - Compromise between BGD and SGD, dataset is divided into mini-batches
  - Smoother convergence than SGD

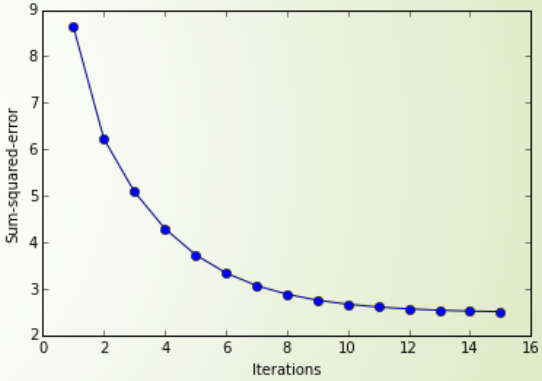
Adaline with SGD in Python

```
import numpy as np

class AdalineSGD(object):
    def __init__(self, alpha=0.01, epochs=50):
        self.alpha = alpha
        self.epochs = epochs
    def train(self, X, y, reinitialize_weights=True):
        if reinitialize_weights:
            self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []
        for i in range(self.epochs):
            for xi, target in zip(X, y):
                output = self.net_input(xi)
                error = (target - output)
                self.w_[1:] += self.alpha * xi.dot(error)
                self.w_[0] += self.alpha * error
            cost = ((y - self.activation(X))**2).sum() / 2.0
            self.cost_.append(cost)
        return self
    def net_input(self, X):
            return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        return self.net_input(X)

    def predict(self, X):
        return np.where(self.activation(X) >= 0.0,
1, -1)
```

每次都update Weight

Adaline with SGD Results

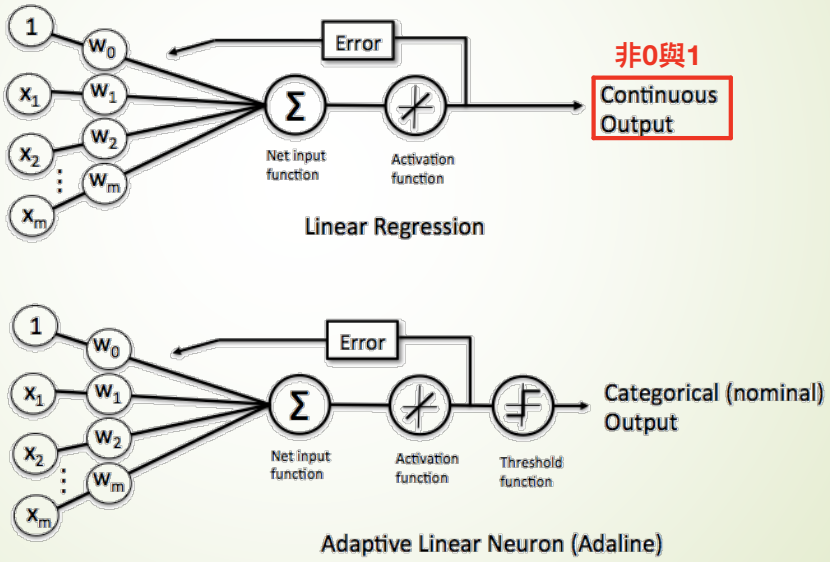




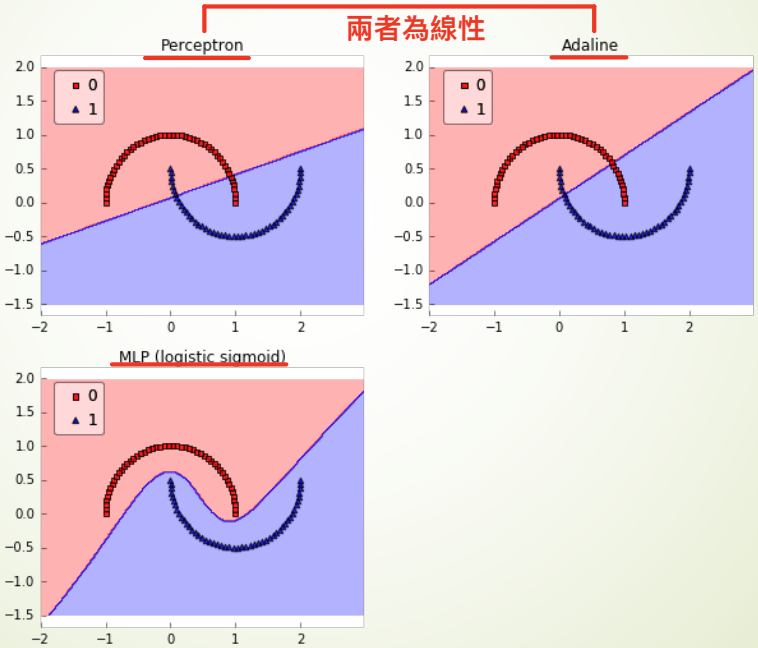
Basics of Neural Networks

- The Perceptron and its Learning Rule (Frank Rosenblatt, 1957)
- Adaptive Linear Neuron and Delta Rule (Widrow & Hoff, 1960)
- Logistic Regression and Gradient Descent

Linear Regression vs. Adaline



Perceptron vs. Adaline vs. Multi-Layer Perceptrons (Logistic Regression)



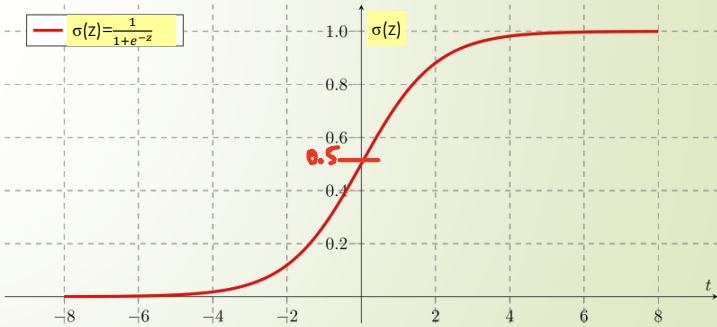
Logistic Regression

"logit" = "log odds"

$odds = \frac{P(event)}{1 - P(event)}$

- Definition:
  - Given input $x \in \mathcal{R}^{n_x}$, calculate the probability $\hat{y} = P(y = 1|x)$, $0 \leq \hat{y} \leq 1$.
- Parameters:
  - Weights: $w \in \mathcal{R}^{n_x}$
  - Bias: $b \in \mathcal{R}$
- Output:
  - $\hat{y} = \sigma(z) = \sigma(w^T x + b)$

    where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid activation function



If z is large positive number, σ(z) → 1
If z is small negative number, σ(z) → 0

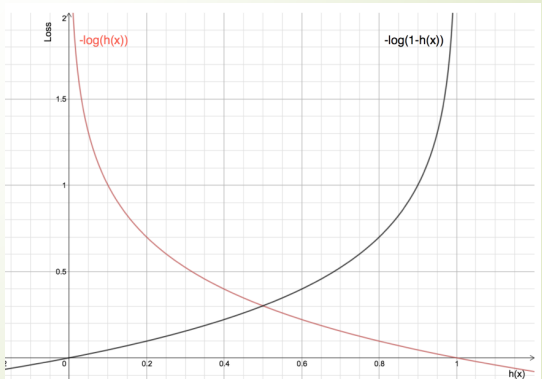
Logistic Regression Cost Function

- For i^{th} input $x^{(i)}$, $\hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$ where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$
  預期 實際
- For each labeled data $(x^{(i)}, y^{(i)})$, we could like $\hat{y}^{(i)} \approx y^{(i)}$, where $\hat{y}^{(i)}$is the predicted output and $y^{(i)}$is the actual expected ground truth value.
- Loss (error)function for each input is defined using Cross-Entropy or Log Loss

  Loss $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$

- Intuition
  - If y=1, $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$
    Y ➨ Need large $\hat{y}$
  - If y=0, $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$
    N ➨ Need small $\hat{y}$

Estimate Parameter Vector



Note: we do not use sum of squared errors because it will be not convex in logistic regression

Logistic Regression Cost Function

- Cost function is the average of all cross-entropy losses

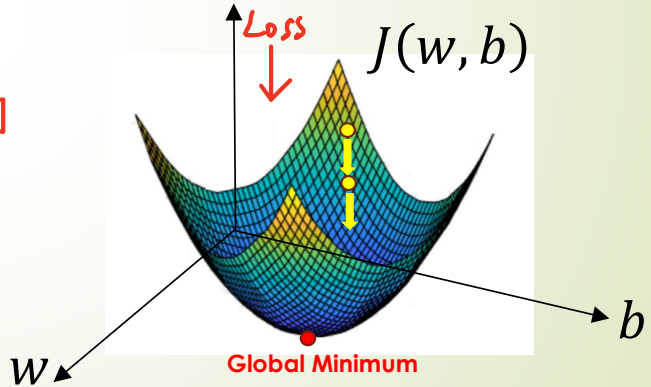
$$
\begin{aligned}
&\mathcal{J}(w, b) \\
&= \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \hat{y}^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right) \right]
\end{aligned}
$$

- Goal:
  - Find vectors **w** and **b** that minimize the cost function (total loss)
- Logistic regression can be viewed as a small neural network!

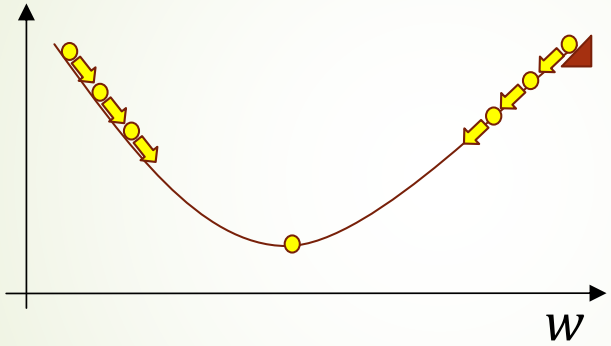
Gradient Descent for Logistic Regression https://www.youtube.com/watch?v=YMJtsYlp4kg

- $\hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$
- $\mathcal{J}(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})\right]$

- Find w, b that minimize $\mathcal{J}(w,b)$



Gradient Descent for Logistic Regression



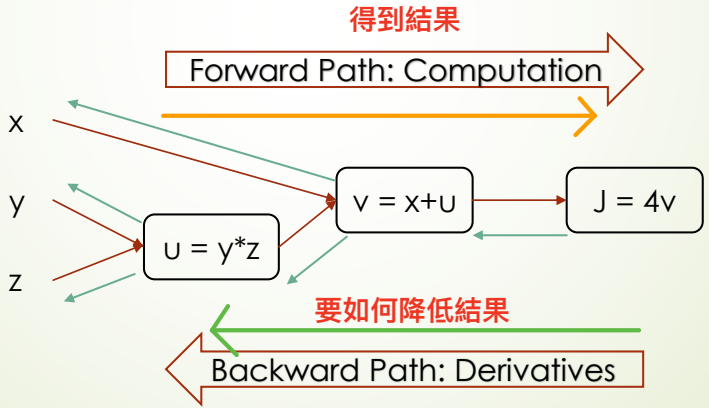
Repeat {

$$w := w - \propto \frac{\delta J(w)}{\delta w}$$

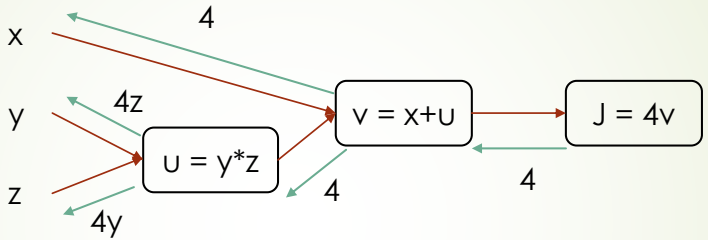
}

Computation Graph

- A graph that depicts all the computations required for a function in a forward path
- For example: J(x, y, z) = 4(x + yz)



Computing Derivatives



$$\frac{\delta J}{\delta v} = 4$$

$$\frac{\delta J}{\delta x} = \frac{\delta J}{\delta v}\frac{\delta v}{\delta x} = 4 \times 1 = 4$$

$$\frac{\delta J}{\delta u} = \frac{\delta J}{\delta v}\frac{\delta v}{\delta u} = 4 \times 1 = 4$$

$$\frac{\delta J}{\delta y} = \frac{\delta J}{\delta u}\frac{\delta u}{\delta y} = 4 \times z = 4z$$

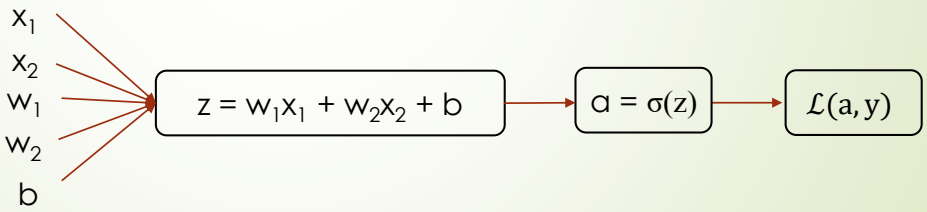
$$\frac{\delta J}{\delta z} = \frac{\delta J}{\delta u}\frac{\delta u}{\delta z} = 4 \times y = 4y$$

Logistic Regression Computation Graph

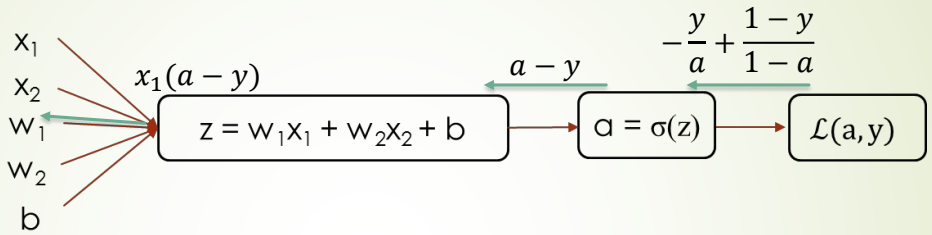
$$z = w^T x + b$$

$$\hat{y} = a = \sigma(z)$$

$$\mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$$



Logistic Regression Derivatives



- $\frac{\delta L(a,y)}{\delta a} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)$
- $\frac{\delta L(a,y)}{\delta z} = \frac{\delta L(a,y)}{\delta a}\frac{\delta a}{\delta z} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)\left(a(1-a)\right) = a - y$
- $\frac{\delta L(a,y)}{\delta w_1} = \frac{\delta L(a,y)}{\delta a}\frac{\delta a}{\delta z}\frac{\delta z}{\delta w_1} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)a(1-a)x_1 = x_1(a-y) = x_1\frac{\delta L(a,y)}{\delta z}$

Logistic Regression on *m* examples

$J = 0;\ dw_1 = 0;\ dw_2 = 0;\ db = 0;$

For i = 1 to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J /= m;\quad dw_1 /= m;\quad dw_2 /= m;\quad db /= m;$

全部計算sample後才更新weigh

Update weights:

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

Vectorization (demo)

```
import numpy as np
a = np.array([1,2,3,4])
print(a)
```

```
import time
a = np.random.rand(1000000)
b = np.random.rand(1000000)
tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized version:" + str(1000*(toc-tic) + "ms")
```

```
c = 0
tic = time.time()
for i = range(1000000):
        c += a[i]*b[i]
toc = time.time()
print(c)
print("For loop:" + str(1000*(toc-tic) + "ms")
```

Neural network programming guide

- Whenever possible, avoid explicit for loops!
- Instead of:

```
u = np.zeros((n,1))
for i in range(n):
    u[i] = math.exp(v[i])
```

- Use vectors:

```
u = np.exp(v)
```

- Other functions in numpy: np.log(v), np.abs(v), np.maximum(v,0)
  Other element-wise computations: v**2, 1/v

Vectorizing Logistic Regression

dw = np.zeros((nx, 1))

J = 0; ~~$dw_1 = 0;\ dw_2 = 0;$~~ db = 0;

For i = 1 to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

~~$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$~~

~~$dw_2 \mathrel{+}= x_1^{(i)} dz^{(i)}$~~

$dw \mathrel{+}= x^{(i)} dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

J /= m, ~~$dw_1 /= m,\ dw_2 /= m,$~~ db /= m

dw /= m

One for loop is thus removed!!!

Vectorizing Logistic Regression

- $X = [x^{(1)}\ x^{(2)}\ \dots\ x^{(m)}]$
- $Z = [z^{(1)}\ z^{(2)}\ \dots\ z^{(m)}] = w^T X + [b\ b\ \dots\ b] =$ np.dot(w.T, X) + b
- $A = [a^{(1)}\ a^{(2)}\ \dots\ a^{(m)}] =$ sigmoid(Z)

Vectorizing Gradient Descent

- $dz^{(1)} = a^{(1)} - y^{(1)}$, $dz^{(2)} = a^{(2)} - y^{(2)}$, … (all m examples)
- $dZ = [dz^{(1)}\ dz^{(2)}\ \dots\ dz^{(m)}]$
- $A = [a^{(1)}\ \dots\ a^{(m)}]$
- $Y = [y^{(1)}\ \dots\ y^{(m)}]$
- $dZ = A - Y = [a^{(1)}-y^{(1)}\ \dots\ a^{(m)}-y^{(m)}]$
- $db = \frac{1}{m}\sum_{i=1}^{m} dz^{(i)} = \frac{1}{m} np.sum(dZ)$
- $dw = \frac{1}{m} X dZ^T$

Vectorizing Gradient Descent

J = 0; dw = np.zeros((nx,1)); db = 0;

For i = 1 to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw \mathrel{+}= x^{(i)} * dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

J /= m, dw /= m, db /= m

$Z = w^T X + b$ = np.dot(w.T, X) + b
A = sigmoid(Z)

$dZ = A - Y$
$dw = 1/m \; X \; dZ^T$
db = 1/m np.sum(dZ)

$w := w - \alpha \, dw$
$b := b - \alpha \, db$

Vectorized Version: Without any for loop!

Note on Python/numpy vectors

- Don't use rank 1 arrays of shape (n,)

  a = np.random.randn(5)

  print(a)

  [0.50290632 -0.2 …]

  print(a.shape)

  (5,)

- Use column vectors of shape (n, 1) or row vectors of shape (1, n)

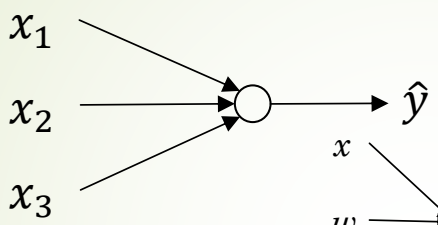
  a = np.random.randn(5,1)

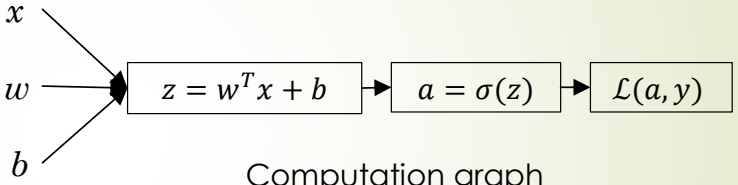
Contents

- Basics of Neural Networks
- Shallow Neural Networks
- Deep Neural Networks

What is a Shallow Neural Network?

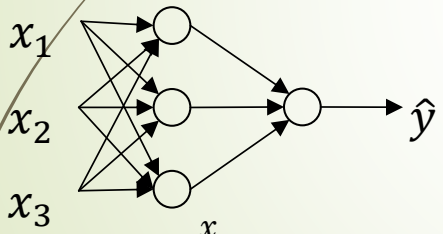
Single neuron model

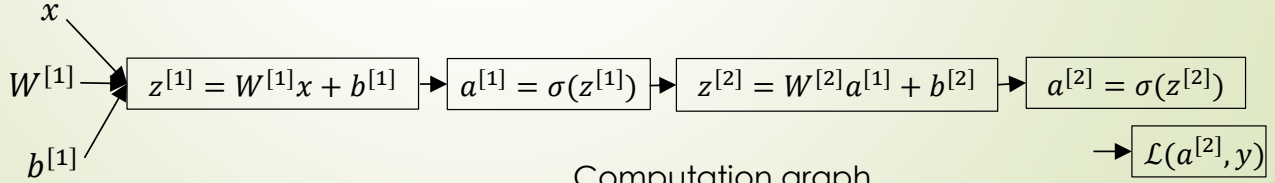




Computation graph

Multiple layer neural model

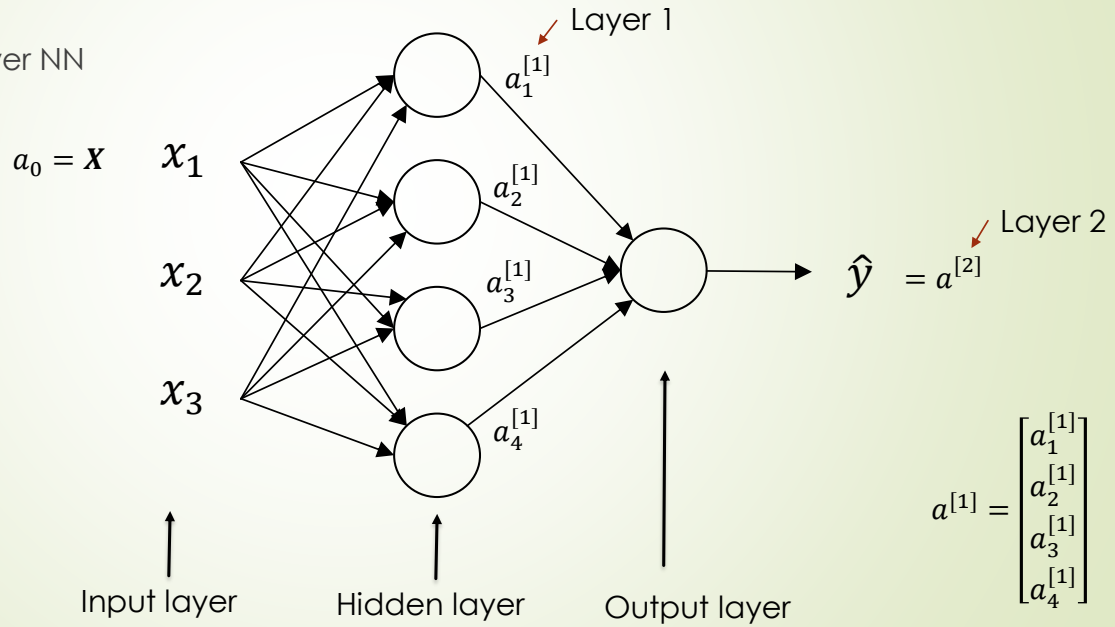




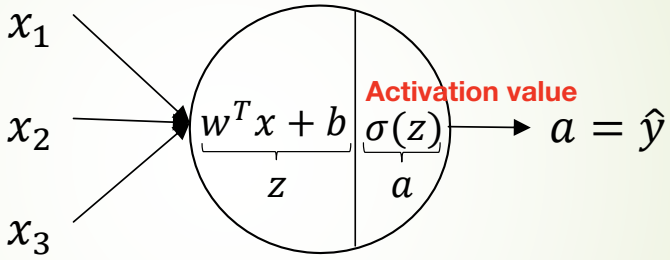
Computation graph

One hidden layer Neural Network

- 2-layer NN



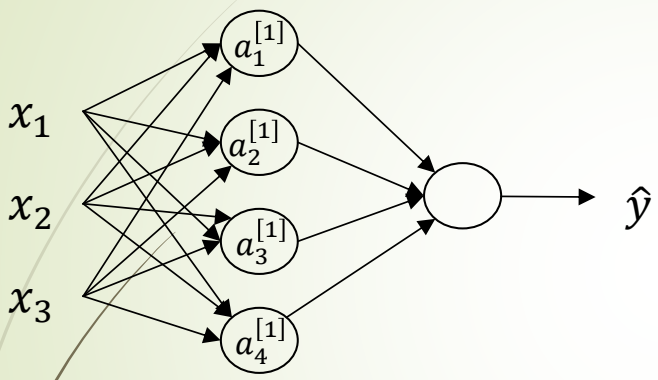
Computing NN's Output



$$z = w^T x + b$$

$$a = \sigma(z)$$

Computing NN's Output



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

Given input x:

$$z^{[1]} = W^{[1]} x + b^{[1]}$$

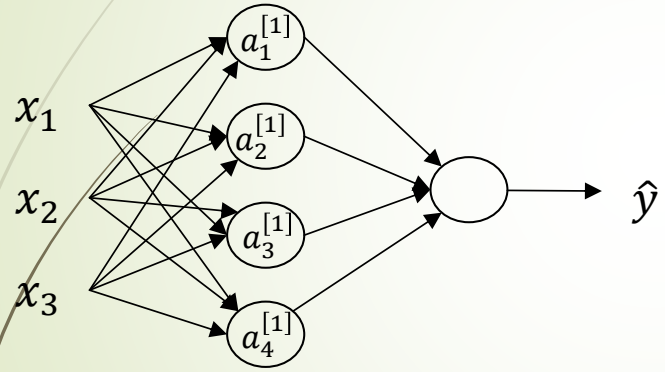
$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

Vectorizing across multiple examples

- For all m examples …




```
for i = 1 to m:
```

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

Vectorizing across multiple examples

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$\mathrm{A}^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

```
for i = 1 to m:
```

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

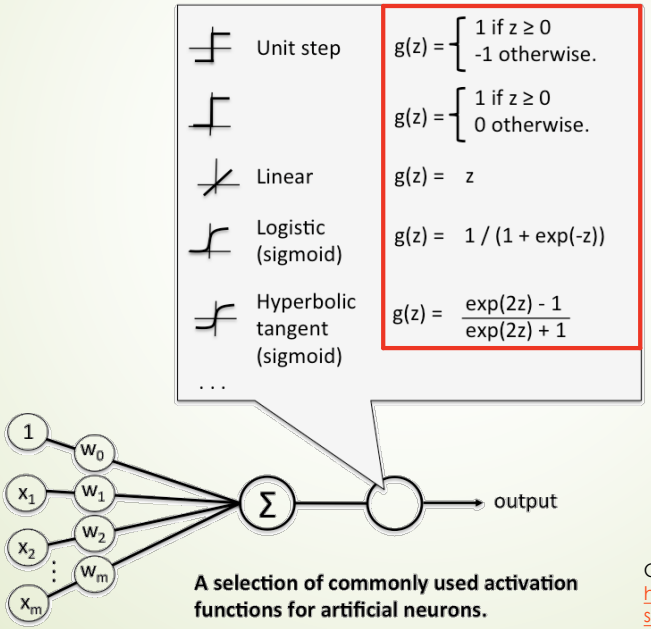
$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

Activation functions can be …

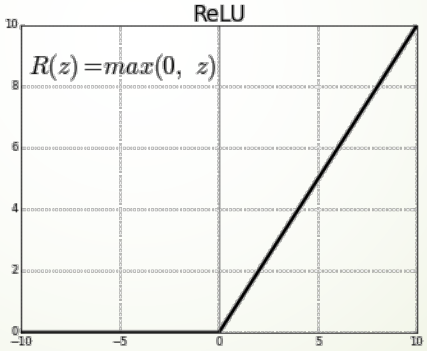


A selection of commonly used activation functions for artificial neurons.

Comprehensive List of Activation Functions:
https://stats.stackexchange.com/questions/115258/comprehensive-list-of-activation-functions-in-neural-networks-with-pros-cons

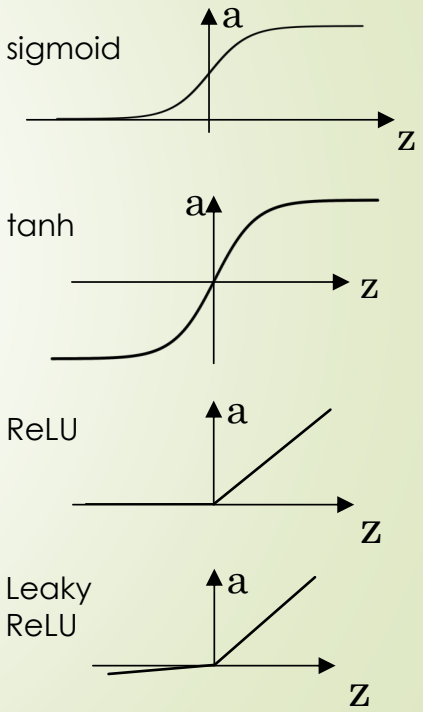
REctified Linear Unit (RELU)

- Currently, most popular activation function
- Gradient descent is much faster with ReLU



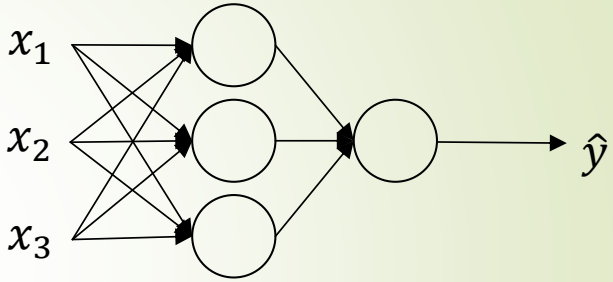
Derivatives of Activation Functions

Activation Function	Formula (g(z))	Derivative (g'z))
sigmoid	$a = \frac{1}{1+e^{-z}}$	$a(1-a)$
tanh	$a = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$1-a^2$
ReLU	$\max(0,z)$	0 if $z<0$ 1 if $z\geq 0$
Leaky ReLU	$\max(0.01z,z)$	0.01 if $z<0$ 1 if $z\geq 0$



Why non-linear activation function?

- What not linear?
- Suppose $g^{[1]}$, $g^{[2]}$ are all linear
  - $a^{[1]} = z^{[1]}$
  - $a^{[2]} = z^{[2]}$
  - $a^{[2]} = W^{[2]}a^{[1]}+b^{[2]}$
  - $= W^{[2]}(W^{[1]}X+b^{[1]})+b^{[2]}$
  - $= W^{[2]}W^{[1]}X + W^{[2]}b^{[1]}+b^{[2]}$
  - $= W'X + b'$
- All LINEAR!!!



Given x:

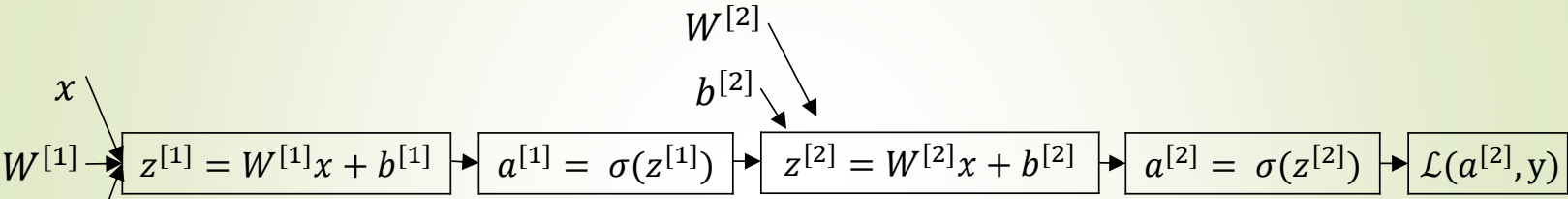
$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

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

Gradient descent for one hidden layer neural network



$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]}) \qquad dz^{[2]} = a^{[2]} - y$$

$$dW^{[1]} = dz^{[1]} x^T \qquad dW^{[2]} = dz^{[2]} a^{[1]T}$$

$$db^{[1]} = dz^{[1]} \qquad db^{[2]} = dz^{[2]}$$

Vectorizing Gradient Descent (over m examples)

One example

第二層 第二層

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]} a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$$

$$dW^{[1]} = dz^{[1]} x^{T}$$

$$db^{[1]} = dz^{[1]}$$

All example

$$dZ^{[2]} = A^{[2]} - Y$$

$$J(..) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

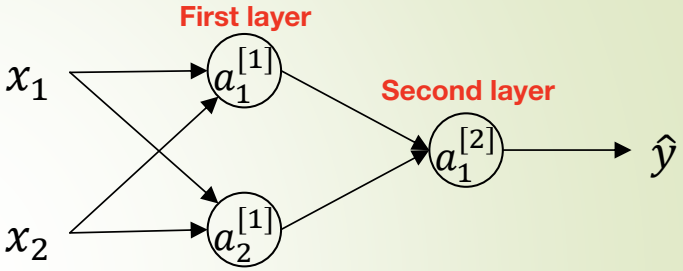
$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$$

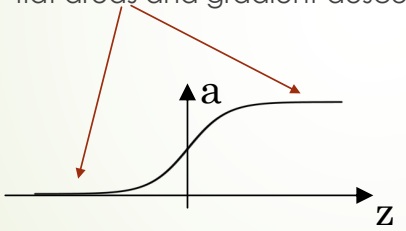
Initializing weights



- What is weights are initialized to **zero**?
- Suppose all weights are zero:
  - $W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
  - $b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
  - $a_1^{[1]} = a_2^{[1]}$
  - $dz_1^{[1]} = dz_2^{[1]}$
  - $dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$ (i.e., symmetric rows)
  - $W^{[1]} = W^{[1]} - \alpha dW$
  - $W^{[1]} = \begin{bmatrix} f & g \\ f & g \end{bmatrix}$ (i.e., symmetric rows)
- No need of TWO or more neurons … because all computations are same!
  - Do NOT initialize all weights are ZERO!!

Initialize weights RANDOMLY!

- W[1] = np.random.randn((2,2))***0.01**
  - **Small** random values are suggested!
  - If too large, $Z^{[1]} = W^{[1]}X + b^{[1]}$ will also be very large and $a^{[1]} = g^{[1]}(z^{[1]})$ will be in the flat areas and gradient descent will be very, very slooooooooow….

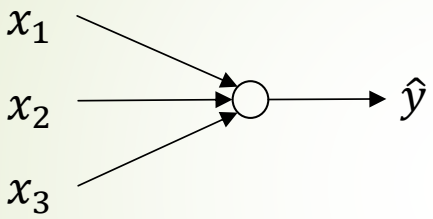


sigmoid

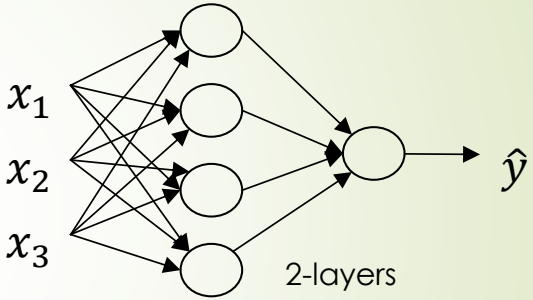
- b[1] = np.zero((2,1)) (b can be zero, no problem!)

Contents

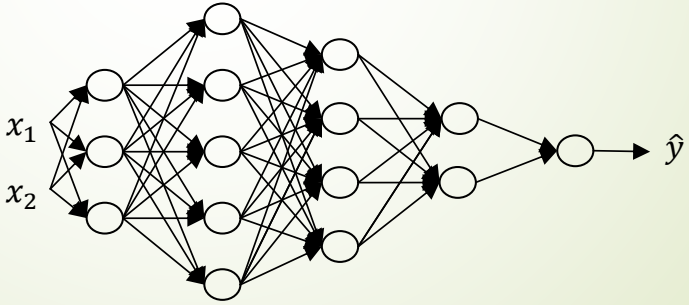
Neural Networks: 1-layer, 2-layers, …



1-layer



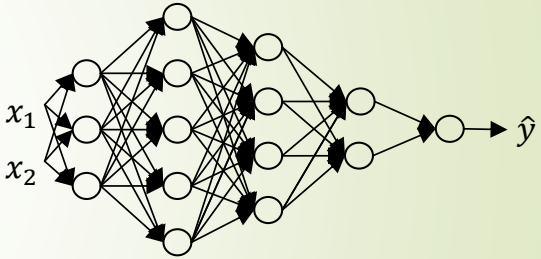
2-layers



5-layers

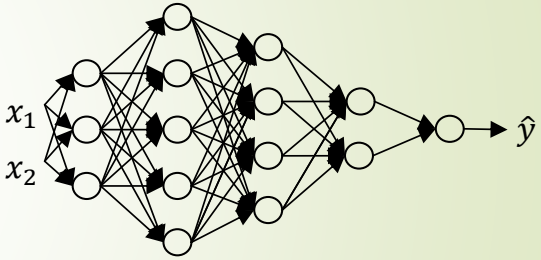
Notations in Deep Neural Networks

- #Layers = $L = 5$
- $n^{[l]}$ = #units in layer l **幾個神經元**
  - $n^{[1]} = 3$, $n^{[2]} = 5$, $n^{[3]} = 4$, $n^{[4]} = 2$, $n^{[5]} = n^{[L]} = 1$
  - $n^{[0]} = n^{[x]} = 2$ **兩個input**
- $a^{[l]} = g^{[l]}(z^{[l]})$ **Activation function**
- $W^{[l]}$ = weights for computing $z^{[l]}$
- $b^{[l]}$ = bias for computing $z^{[l]}$
- $a^{[0]} = X$ (input)
- $a^{[L]} = \hat{y}$ (prediction output)



Forward Propagation in Deep Neural Network
(Vectorized Version over all m examples)

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$
$$\dots$$
$$\hat{Y} = A^{[L]} = g^{[L]}(Z^{[L]})$$



$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

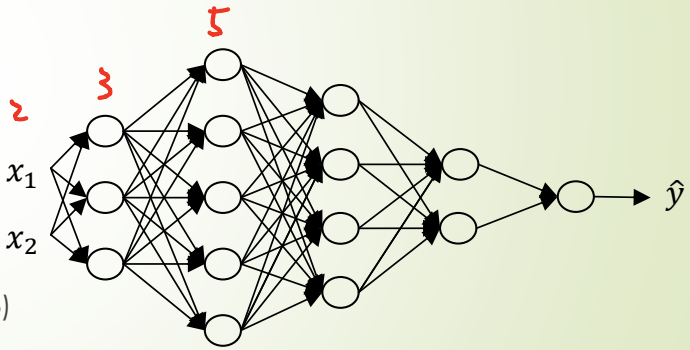
- **Still need a "for" loop over all layers 1 to L**

Getting your matrix dimensions correct

- $L = 5$
- $n^{[1]} = 3$, $n^{[2]} = 5$, $n^{[3]} = 4$, $n^{[4]} = 2$, $n^{[5]} = n^{[L]} = 1$

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

- (3, 1) = (?, ?) x (2, 1) + (3, 1)
- ($n^{[1]}$, 1) = ($n^{[1]}$, $n^{[0]}$) x ($n^{[0]}$, 1)
- Thus, dim of $W^{[1]}$ is ($n^{[1]}$, $n^{[0]}$)
- Similarly, dim of $W^{[2]}$ is ($n^{[2]}$, $n^{[1]}$)
  - In this example, dim of $W^{[2]}$ is (5, 3)
- Thus, dim of $W^{[l]}$ is $(\boldsymbol{n^{[l]}, n^{[l-1]}})$
- and dim of $b^{[l]}$ is $(\boldsymbol{n^{[l]}, 1})$



W[3]: (4, 5)
W[4]: (2, 4)
W[5]: (1, 2)

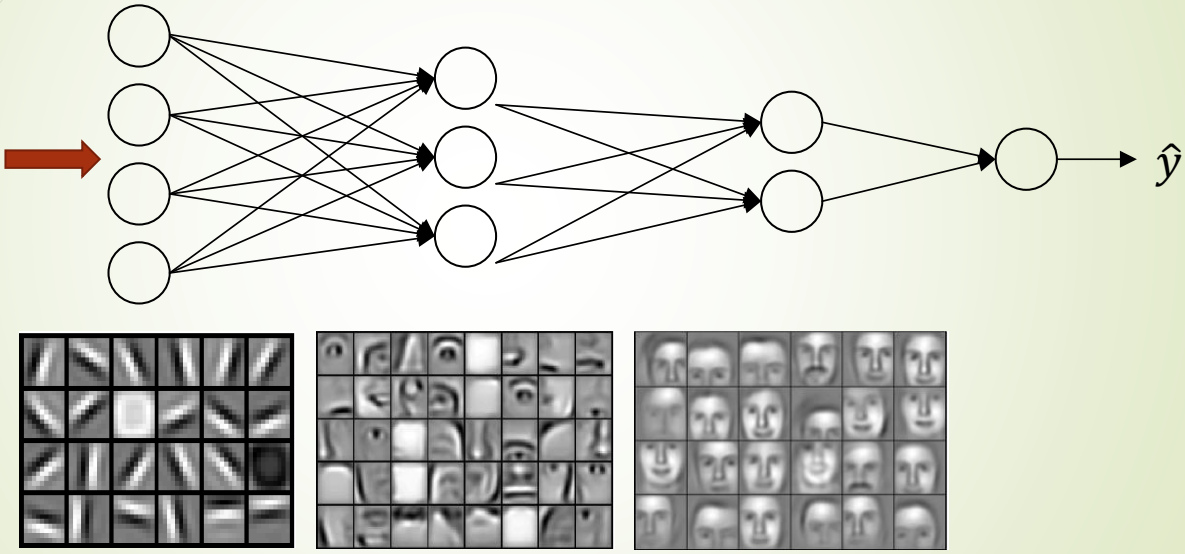
Dimensions of vectorized implementations

- For one single training example:
- $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$
- $(n^{[l]}, 1) = (n^{[l]}, n^{[l-1]}) \times (n^{[l-1]}, 1) + (n^{[l]}, 1)$

- For a vectorized implementation over m examples
- $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
- $(n^{[l]}, m) = (n^{[l]}, n^{[l-1]}) \times (n^{[l-1]}, m) + (n^{[l]}, m)$

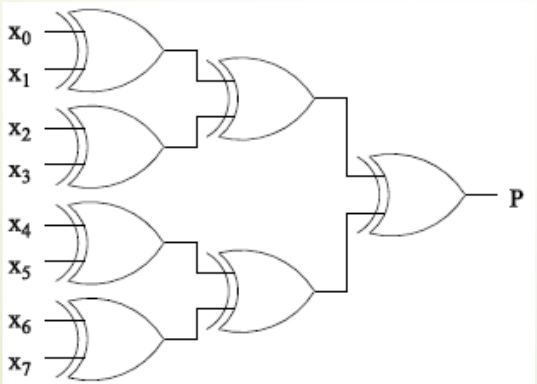
Matrix	Dimensions
$Z^{[l]}, A^{[l]}, b^{[l]}, dZ^{[l]}, dA^{[l]}, db^{[l]}$	$(n^{[l]}, m)$
$W^{[l]}, dW^{[l]}$	$(n^{[l]}, n^{[l-1]})$

Why deep networks?

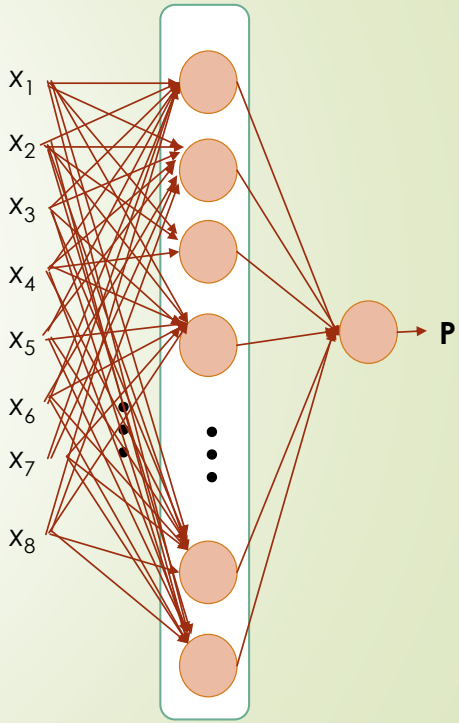


Why deep networks?

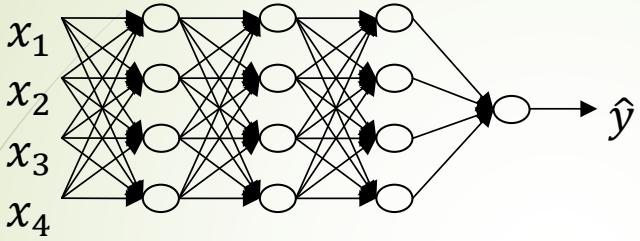
- To check even parity of an 8-bit
- Using 3 layers of XOR gates is much simple
- Using $2^8 = 256$ choices would be much more complex



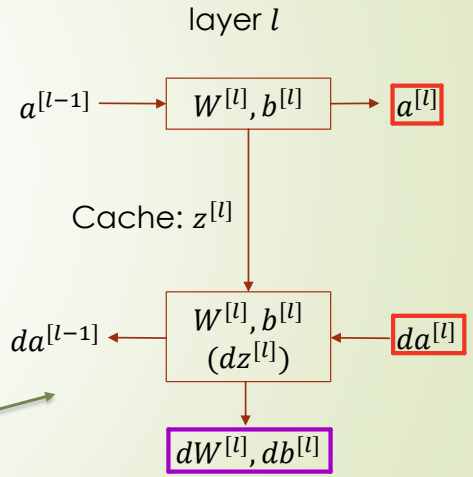
256 neurons in 1 layer



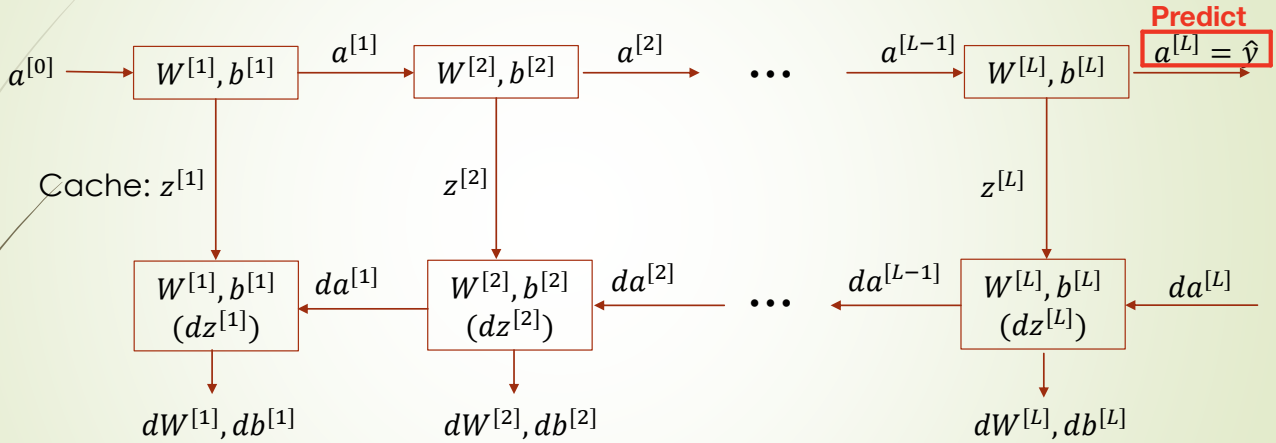
Building Blocks of Deep Neural Networks



- Focus on some layer l, parameters include $W^{[l]}$, $b^{[l]}$
- Forward Propagation
  - Input: $a^{[l-1]}$
  - Output: $a^{[l]}$
  - Method: $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$, $a^{[l]} = g^{[l]}(z^{[l]})$
  - Cache: $z^{[l]}$
- Backward Propagation
  - Input: $da^{[l]}$, $z^{[l]}$ (from cache)
  - Output: $da^{[l-1]}$, $dW^{[l]}, db^{[l]}$



Forward and backward functions



$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

Forward Propagation for DNN

- Input: $a^{[l-1]}$
- Forward Propagation:
  - For single example:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}, \quad a^{[l]} = g^{[l]}(z^{[l]})$$

  - Vectorized version:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}, \quad A^{[l]} = g^{[l]}(Z^{[l]})$$

- Output: $a^{[l]}$
- Cache: $z^{[l]}, W^{[l]}, b^{[l]}$

Backward Propagation for DNN

- Inputs: $da^{[l]}$, $z^{[l]}, W^{[l]}, b^{[l]}$ (last 3 from cache)
- Outputs: $da^{[l-1]}$, $dW^{[l]}, db^{[l]}$
- Backward Propagation:
- For **single example**

$$dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$$
$$dW^{[l]} = dz^{[l]}.a^{[l-1]}$$
$$db^{[l]} = dz^{[l]}$$
$$da^{[l-1]} = W^{[l]T} dz^{[l]}$$
$$dz^{[l]} = W^{[l+1]T} dz^{[l+1]} * g^{[l]'}(z^{[l]})$$

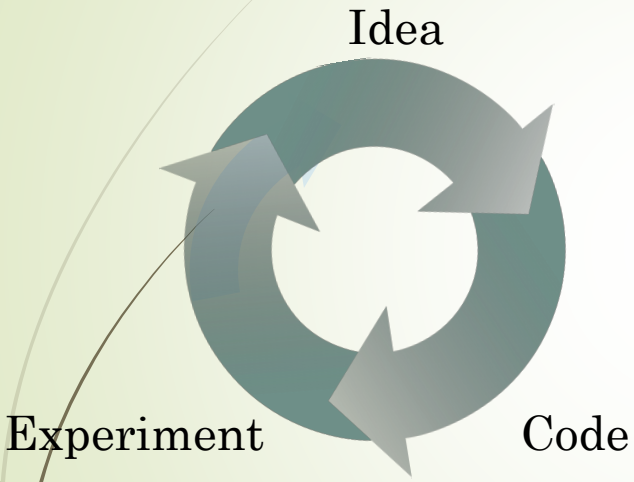
Vectorized implementation

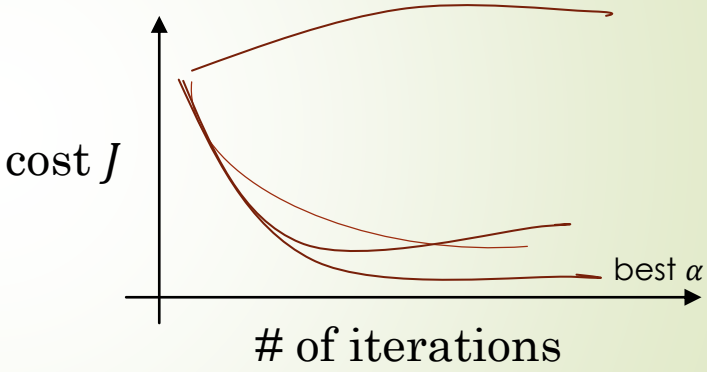
$$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$$
$$dW^{[l]} = \frac{1}{m} dZ^{[l]}.A^{[l-1]T}$$
$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis = 1, keepdims = True)$$
$$dA^{[l-1]} = W^{[l]T} dZ^{[l]}$$

Parameters vs. Hyperparameters

- Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots$
- Hyperparameters
  - Learning rate: α
  - Number of iterations
  - Number of hidden layers or L
  - Number of hidden units in each layer: $n^{[1]}, n^{[2]}, \ldots$
  - Choice of activation function: sigmoid, ReLU, tanh, etc.
  - Momentum, mini-batch size, regularization parameters, … (in the next Chapter)

Empirical Process in Applied DL





Summary of Forward/Backward Computations

$$
\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]}) \\
Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\
A^{[2]} &= g^{[2]}(Z^{[2]}) \\
&\vdots \\
A^{[L]} &= g^{[L]}\left(Z^{[L]}\right) = \hat{Y}
\end{aligned}
$$

Forward Propagation

$$
\begin{aligned}
dZ^{[L]} &= A^{[L]} - Y \\
dW^{[L]} &= \frac{1}{m} dZ^{[L]} A^{[L]^T} \\
db^{[L]} &= \frac{1}{m} np.\mathrm{sum}(\mathrm{d}Z^{[L]}, axis = 1, keepdims = True) \\
dZ^{[L-1]} &= dW^{[L]^T} dZ^{[L]} g'^{[L]}(Z^{[L-1]}) \\
&\vdots \\
dZ^{[1]} &= dW^{[L]^T} dZ^{[2]} g'^{[1]}(Z^{[1]}) \\
dW^{[1]} &= \frac{1}{m} dZ^{[1]} A^{[1]^T} \\
db^{[1]} &= \frac{1}{m} np.\mathrm{sum}(\mathrm{d}Z^{[1]}, axis = 1, keepdims = True)
\end{aligned}
$$

Backward Propagation

Next Step?

- Use Python to implement DNN Forward and Backward Propagations!!!

References

- Single-Layer Neural Networks and Gradient Descent
  - http://sebastianraschka.com/Articles/2015_singlelayer_neurons.html
- Coursera course series on Deep Learning by Andrew Ng