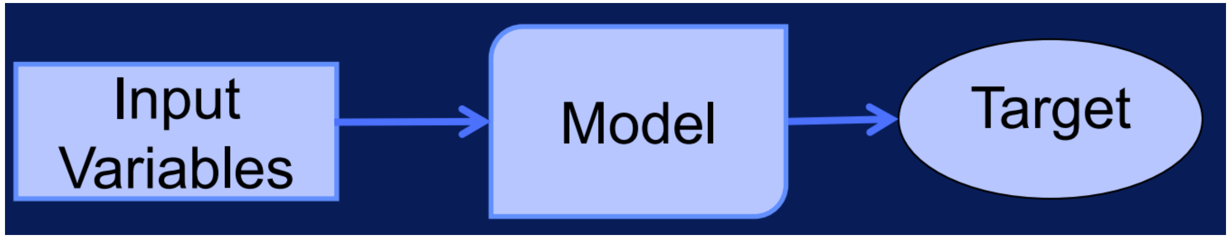
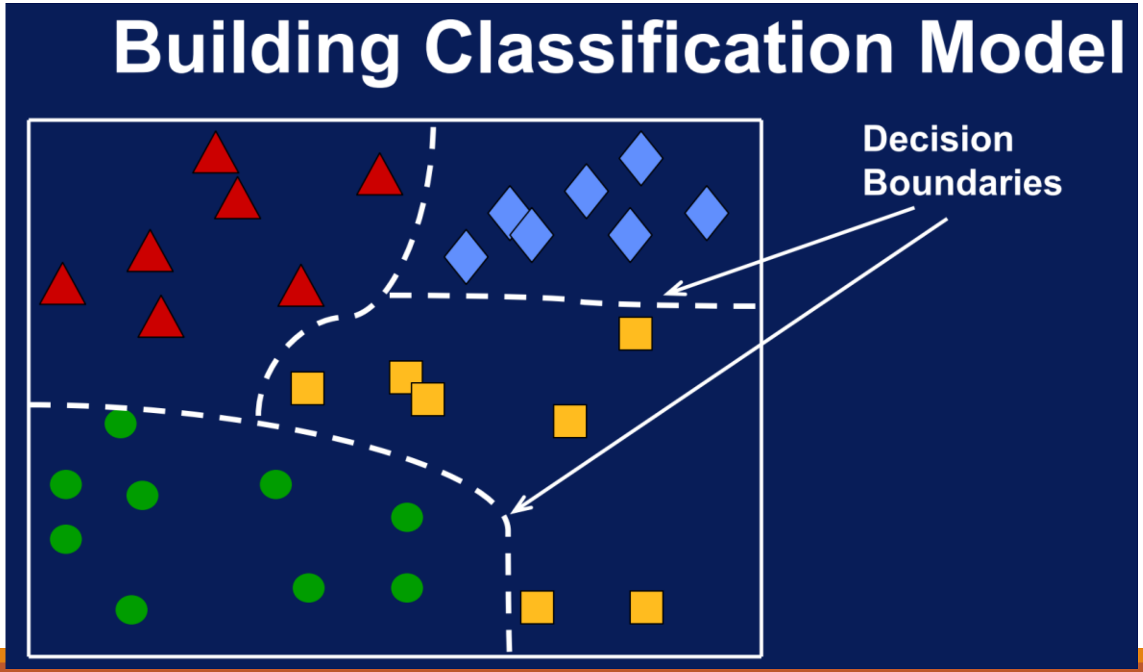
Object Classification with Machine Learning

Classification

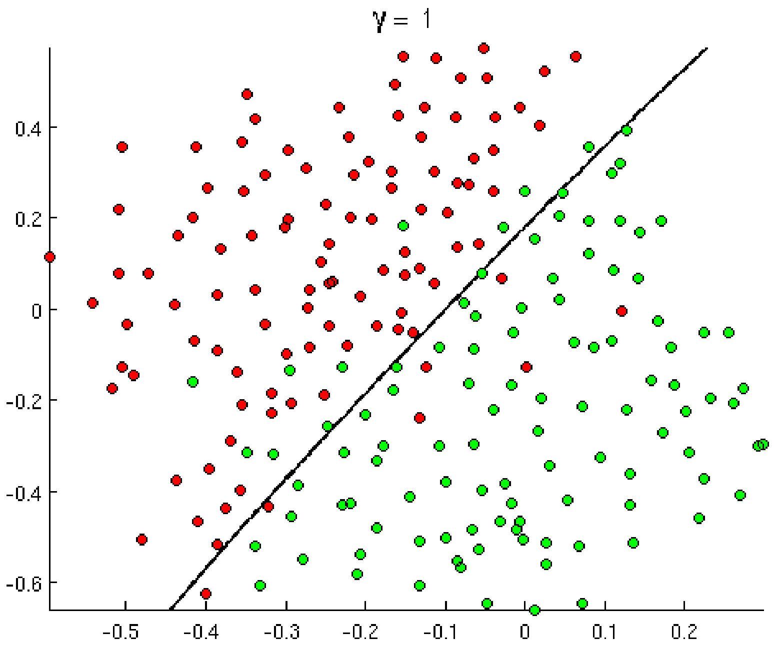
Predict: Category from input variables

Goal: Match model outputs to targets (desired outputs)

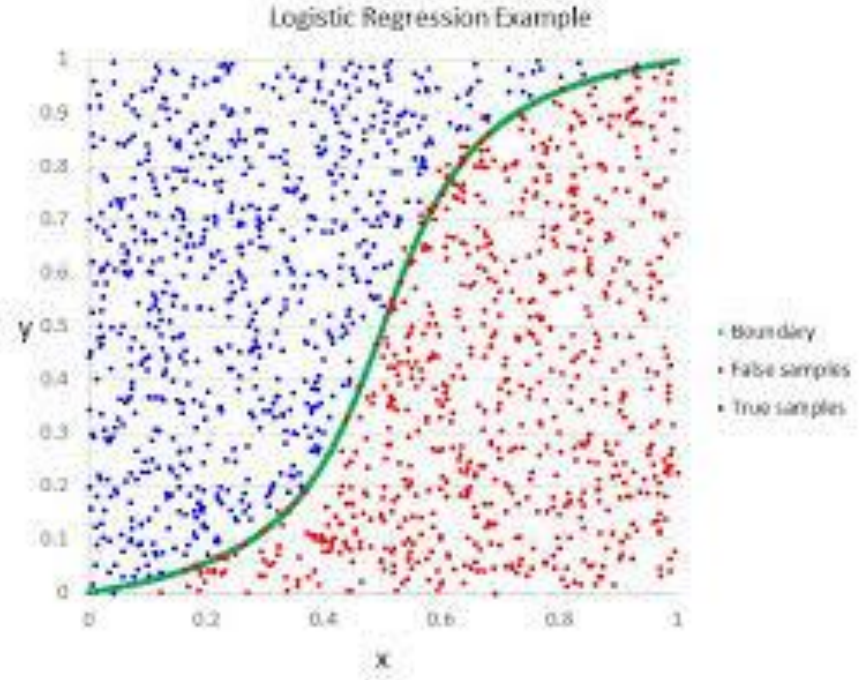




Classification (single dimension Input)

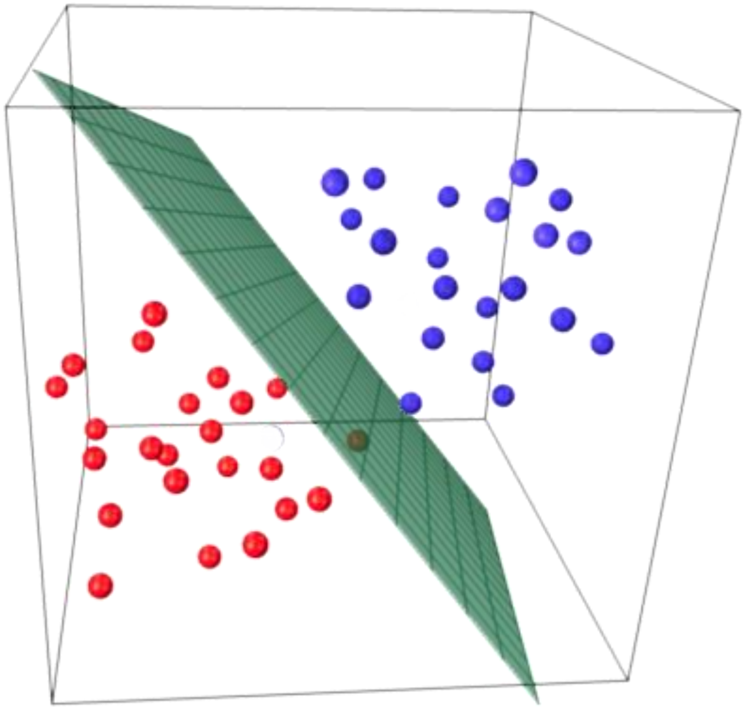


Linear Classifier

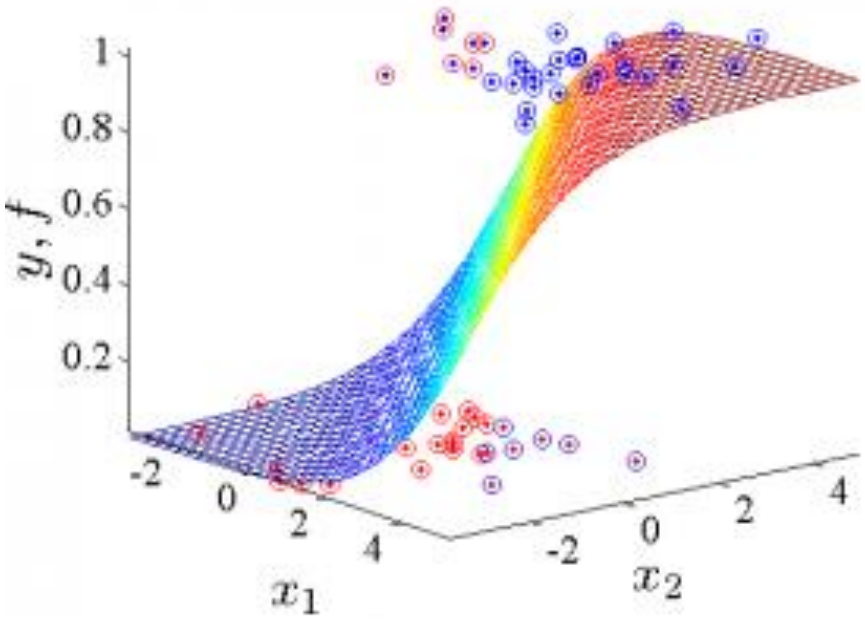


Nonlinear Classifier (Logistic / Sigmoid)

Classification (Multi-dimension Inputs)



Linear Classifier



Nonlinear Classifier
(Logistic / Sigmoid)

Clustering vs Classification

Supervised Classification	Unsupervised Clustering
• known number of classes • based on a training set • used to classify future observations	• unknown number of classes • no prior knowledge • used to understand (explore) data

Supervised Classification

SINGLE NODE MODEL	MULTI-NODE MODEL
kNN (k-Nearest Neighbor)	Neural Networks (Deep Networks)
Logistic Regression	Convolutional Neural Network
Support Vector Machine	Recurrent Neural Network ◦ Long Short Term Memory (LSTM)

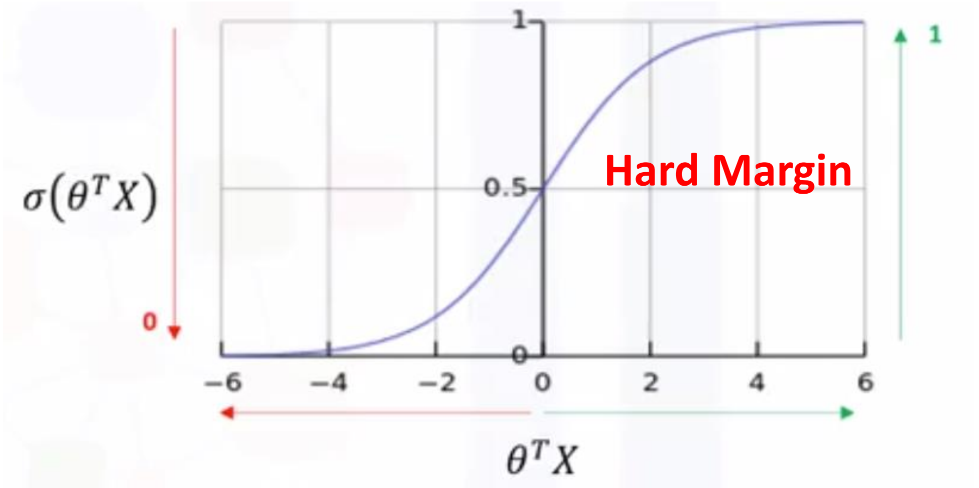
SINGLE NODE MODEL

LOGISTIC REGRESSION VS SVM

ML: Single Node Processor

LOGISTIC REGRESSION

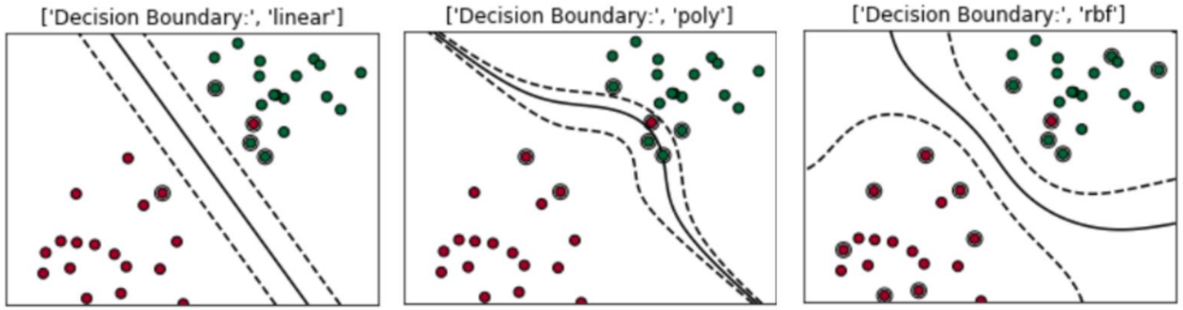
Sigmoid / Logistic Kernel Function



$$h_\theta(X) = \sigma(\theta^T X) = \frac{1}{1 + \exp(\theta^T X)} = \frac{1}{1 + \exp(\theta_1 x_1 + \theta_2\, x_2 + \theta_3\, x_3 + b)}$$

SUPPORT VECTOR MACHINE (SVM)

Class Partition Kernels: Linear / Non-Linear



$$h_\theta(X) = \theta_1 x_1 + \theta_2\, x_2 + \theta_3\, x_3 + b$$

Soft Margin

$$h_\theta(X) = exp\left(-\gamma \left\| x_i - C_j \right\|^2\right)$$

ML: Single Node Processor

Hyperplane Parameter Optimization

LOGISTIC REGRESSION

Hard Margin

Tuning Optimization Parameter:

- C: Regularization factor on
  - weight magnitude
- Kernel parameter
  - Linear -> No parameter

SUPPORT VECTOR MACHINE (SVM)

Soft Margin

Tuning Optimization Parameter:

- C: Regularization factor on
  - Soft margin (Slag variable)
- Kernel parameter
  - Linear -> no parameter
  - RBF -> γ: gamma
  - *Polynomial -: d: polynomial degree*

BEST OPTIMIZING HYPERPLANE PARAMETERS

$(\theta_0, \theta_1, \ldots, \theta_N = b, w_1, w_2, \ldots, w_N)$

Why do we need to tune these parameters?

ML: Single Node Processor (Loss Function)

Hyperplane Parameter (θ_i) Optimization

$$L(x,y) = \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2$$

Loss Function: **Error** Function

- Normally use **L2 distance** between
  - y: real output / desired output / ground truth
  - $h_\theta(X)$: Hyperplane decision kernel

Ex: Linear Hyperplane kernel

$$L(x,y) = \sum_{i=1}^{n} \left(y_i - (\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n)\right)^2$$

$$= \sum_{i=1}^{n} \left(y_i - (b + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n)\right)^2$$

Optimizing on training data only

It is possible to cause **overfitted**

Not understand general inputs

ML: Single Node Processor (Logistic)

Loss function with regularization

$$L(x, y) = \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2$$

C in Logistic

C=1/λ

Large C: less λ -> may be overfitted
Less C: Large λ -> may be underfitted

Regularization on Weight Magnitude

L1 REGULARIZATION

$$L(x, y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \boxed{\lambda \sum_{i=1}^{n} |\theta_i|}$$

L2 REGULARIZATION

$$L(x, y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \boxed{\lambda \sum_{i=1}^{n} \theta_i^2}$$

ML: Single Node Processor (Logistic)

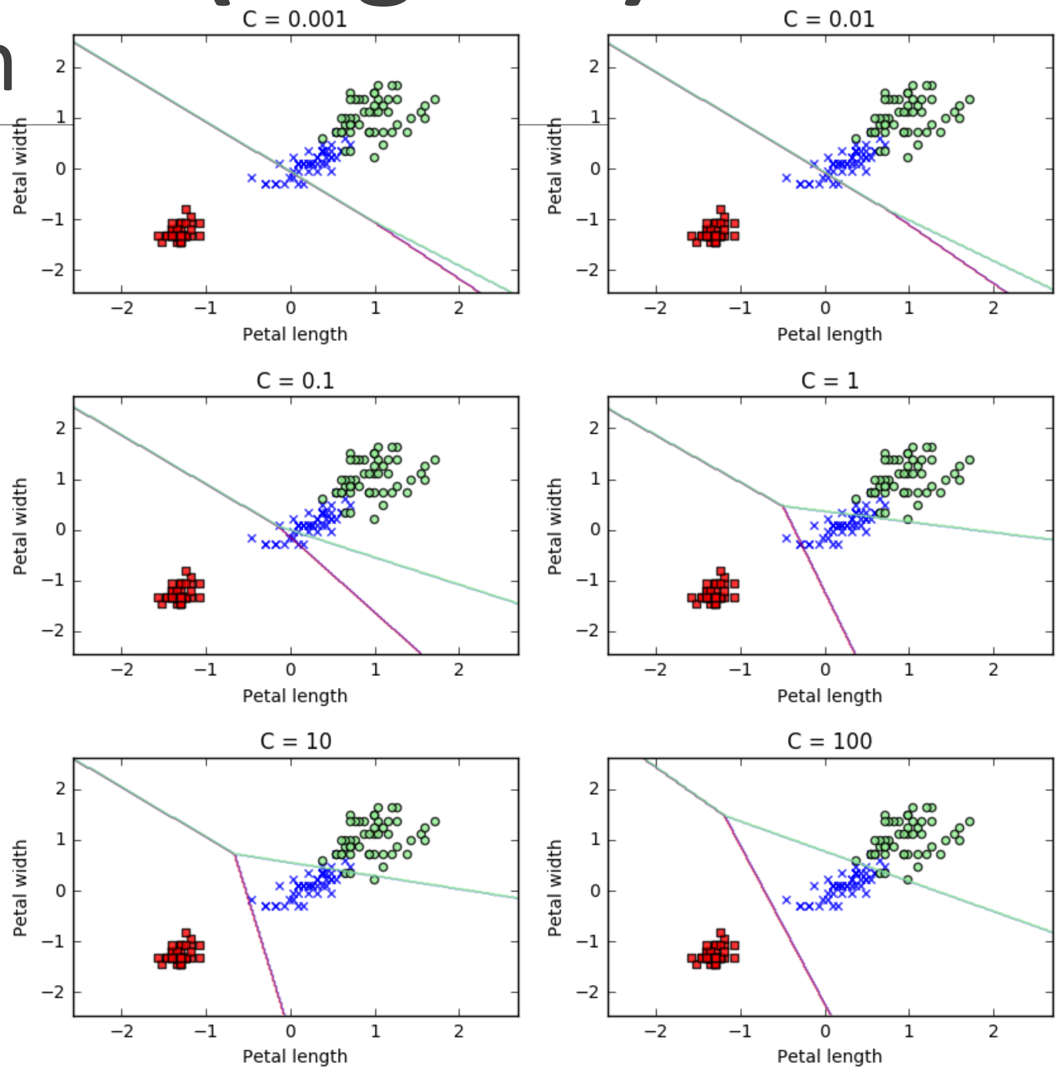
Loss function with regularization

$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \boxed{\lambda \sum_{i=1}^{n} \theta_i^2}$$

C in Logistic

C=1/λ

Large C: less λ -> may be overfitted
Less C: Large λ -> may be underfitted



ML: Single Node Processor (SVM)

Loss function with regularization

$$L(x,y) = \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2$$

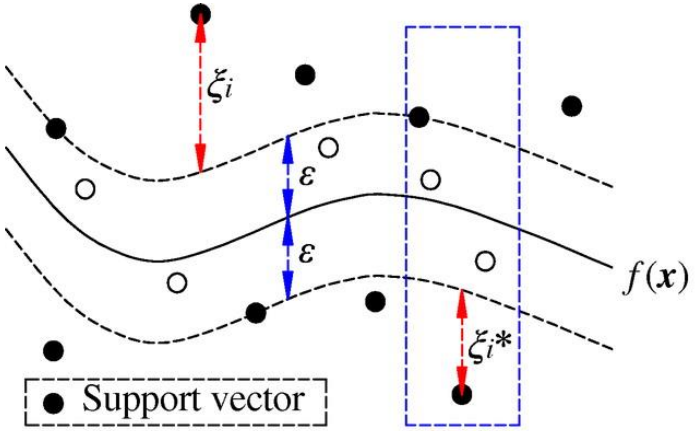
L1 Regularization

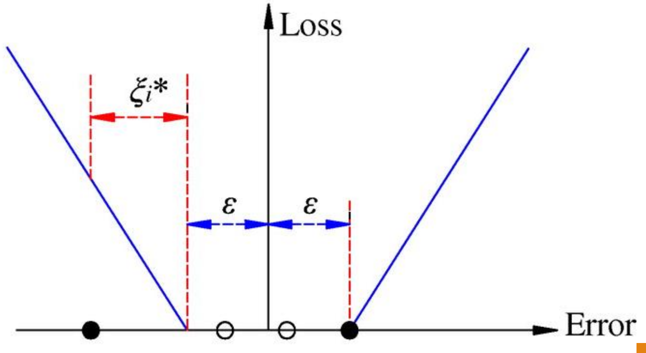
$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \boxed{C \sum_{i}^{N} \xi_i}$$

C

Large C: may be overfitted
Less C: may be underfitted

Regularization on Soft Margin





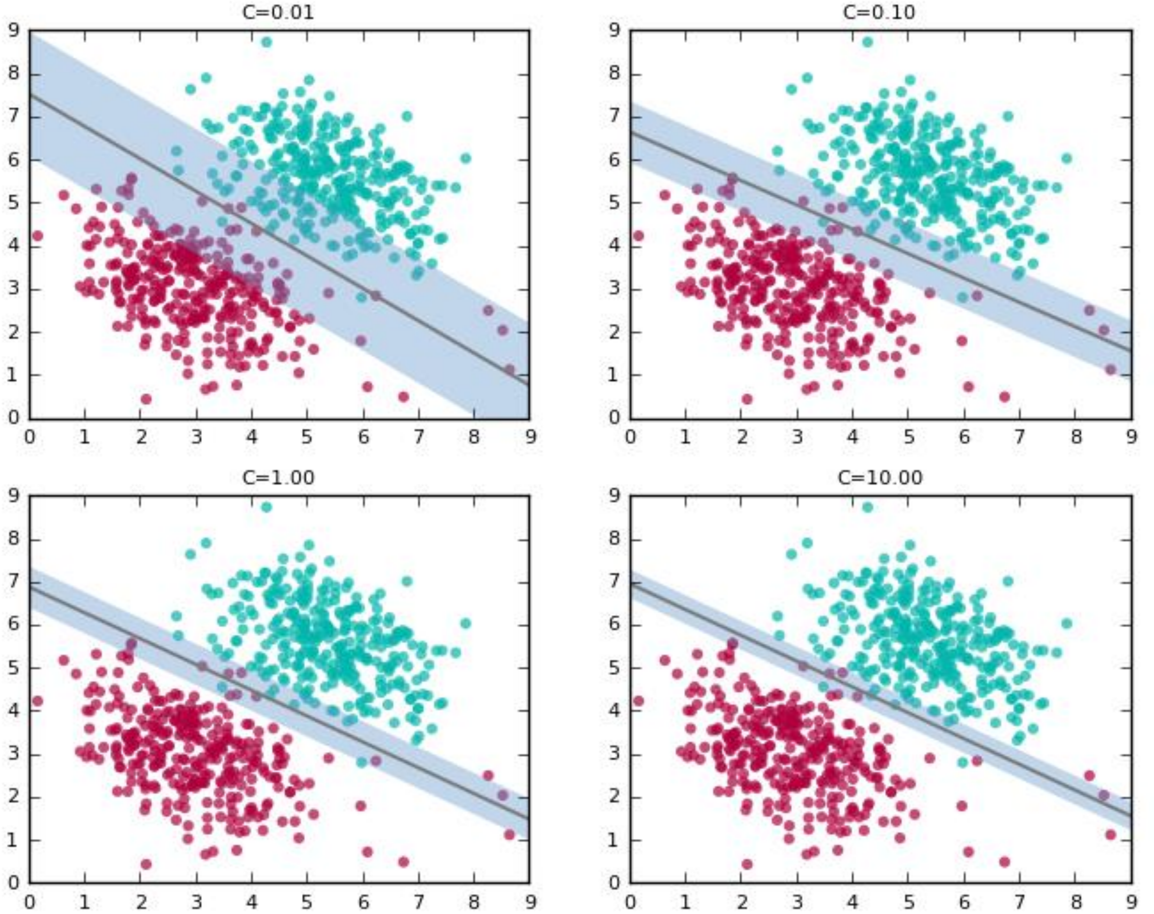
(b)

ML: Single Node Processor (SVM)

Loss function with regularization

L1 Regularization

$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \boxed{C \sum_{i}^{N} \xi_i}$$



C in SVM

Large C: may be overfitted
Less C: may be underfitted

Regularization on Soft Margin

ML: Single Node Processor (SVM)

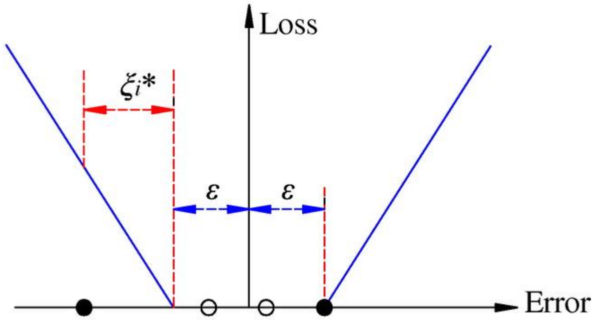
Loss function with regularization

L1 Regularization

$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \boxed{C \sum_{i}^{N} \xi_i}$$

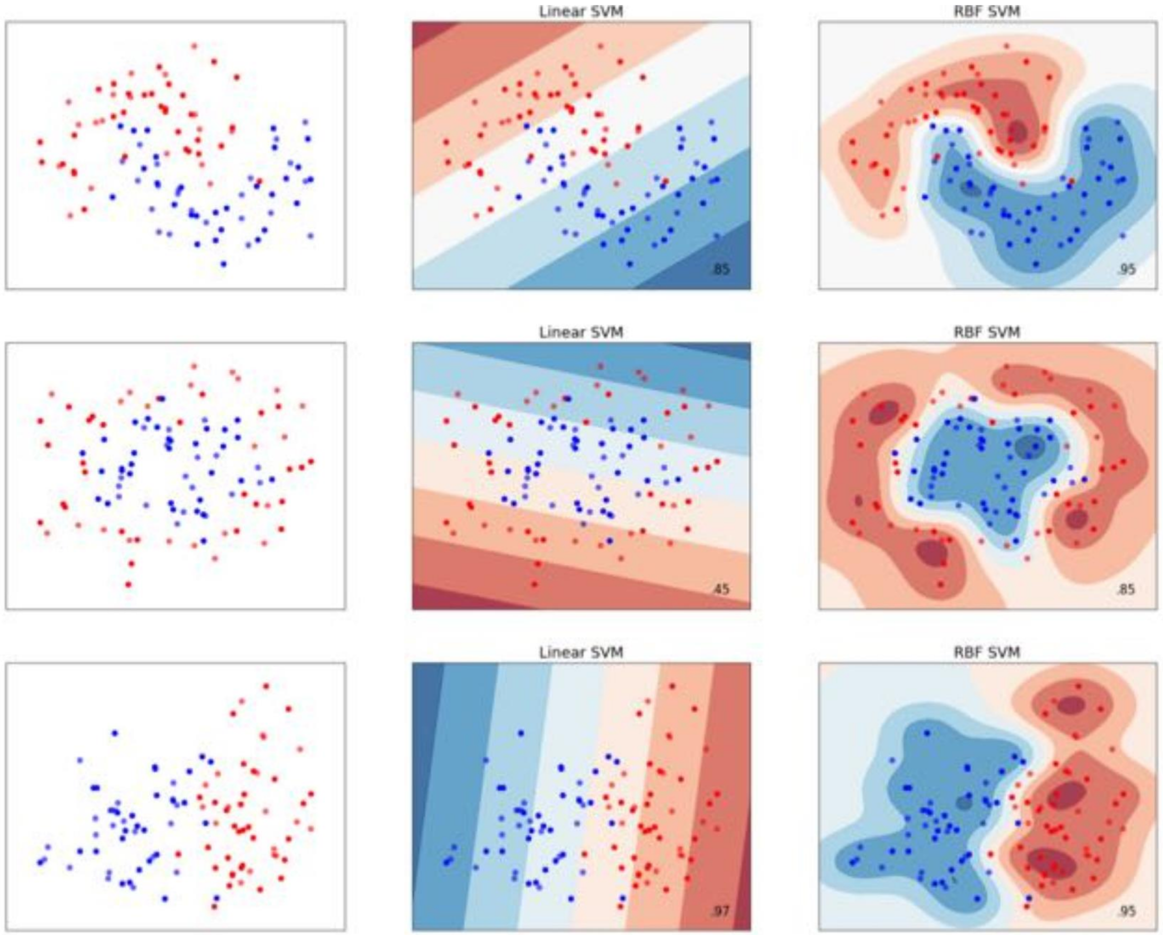
C in SVM

Large C: may be overfitted
Less C: may be underfitted



(b)

Regularization on Soft Margin



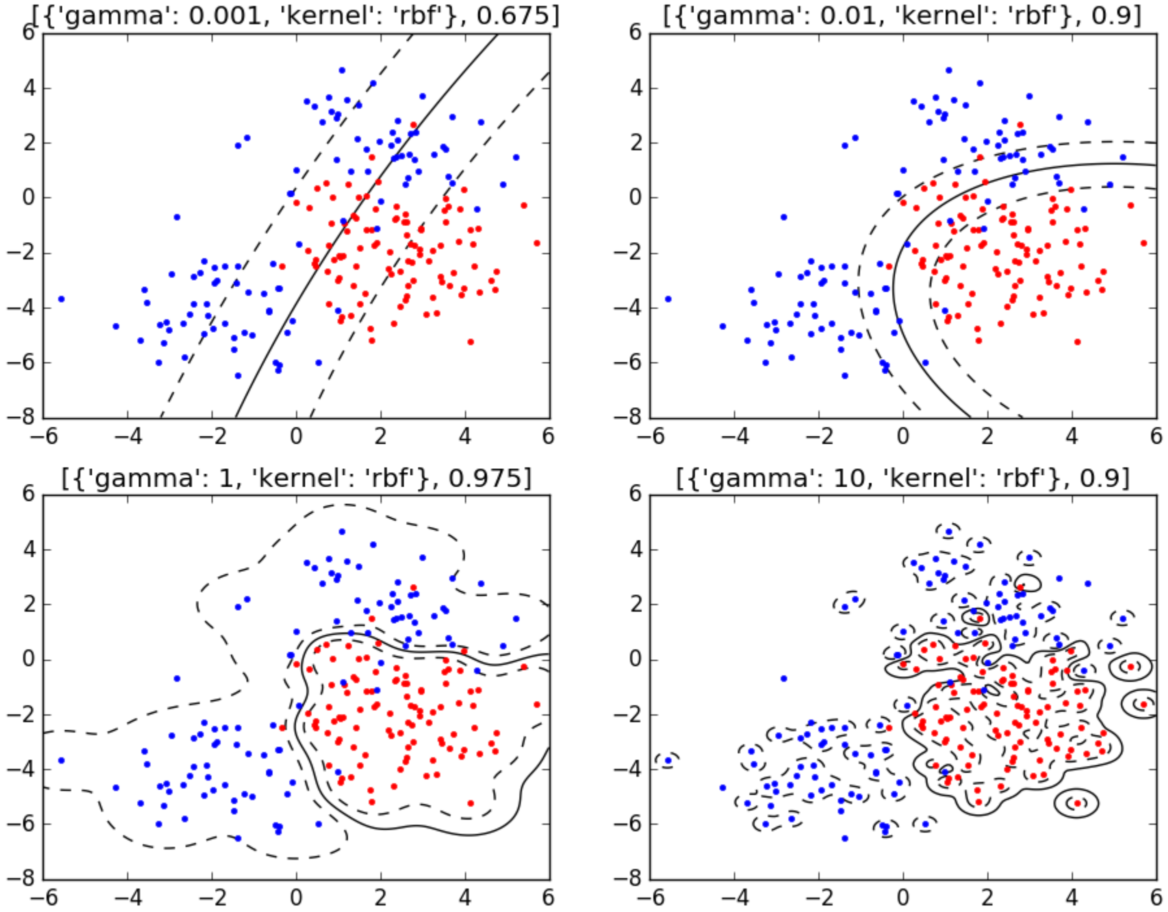
ML: Single Node Processor (SVM)

Loss function with regularization

L1 Regularization

$$L(x,y) \equiv \sum_{i=1}^{n}(y_i - \boxed{h_\theta(x_i)})^2 + C\sum_{i}^{N}\xi_i$$

$$h_\theta(X) = exp\left(-\gamma\left\|x_i - C_j\right\|^2\right)$$



Gamma (γ) in SVM

Large γ : may be overfitted
Less γ : may be underfitted

Regularization on Soft Margin

MULTI-NODE MODEL

DEEP NEURAL NET / CNN

Deep Neural Network

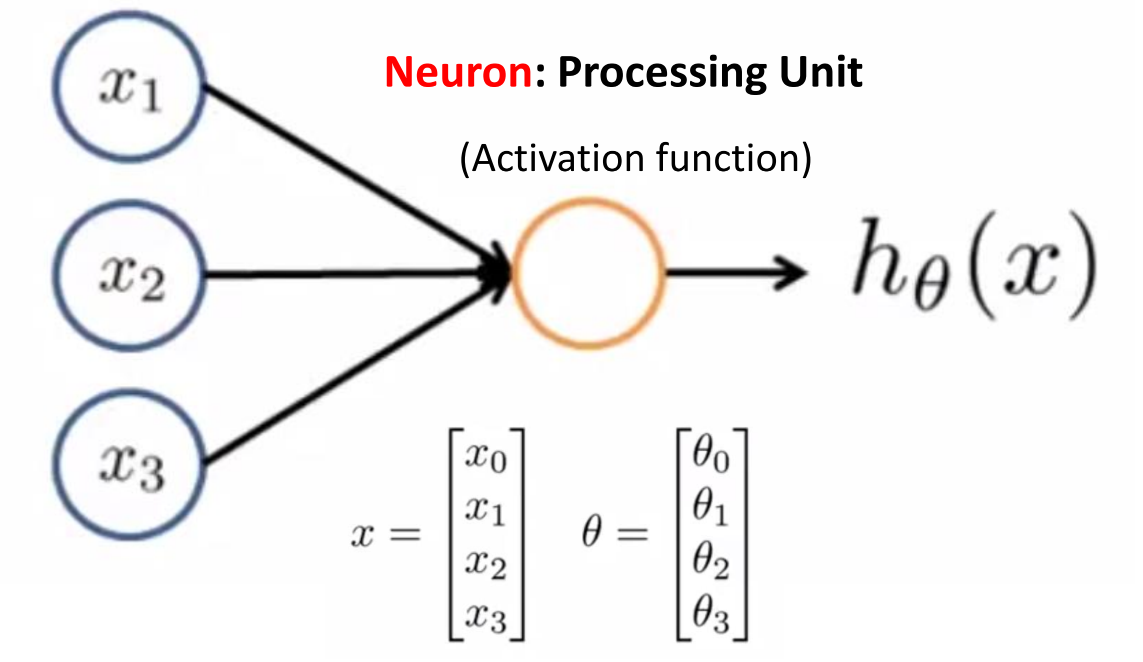
MODEL STRUCTURE

Deep Neural Network

SINGLE NODE NEURON

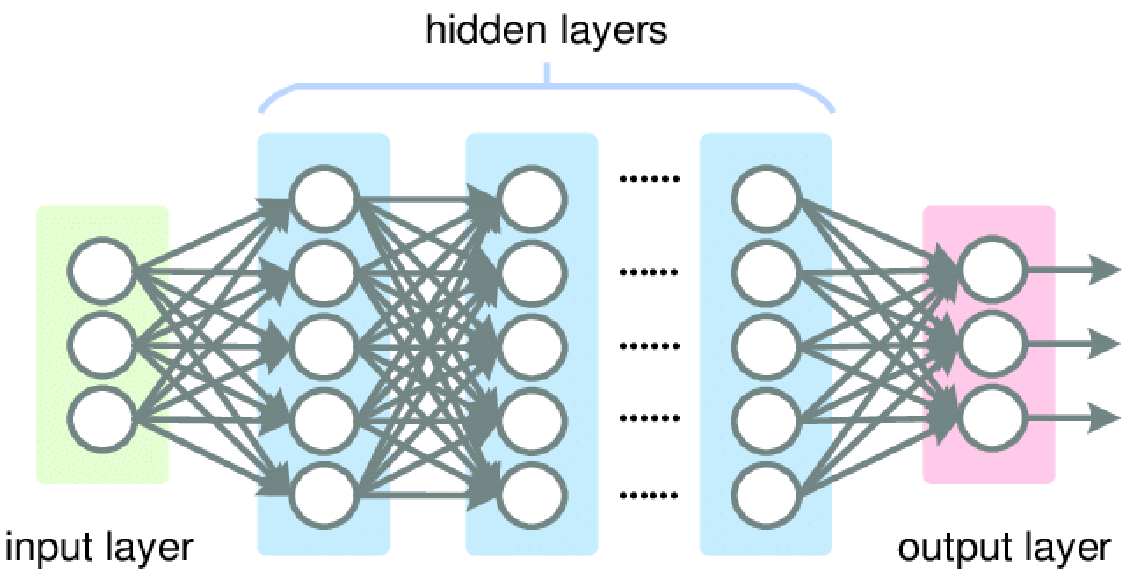
Neuron: Processing Unit

(Activation function)



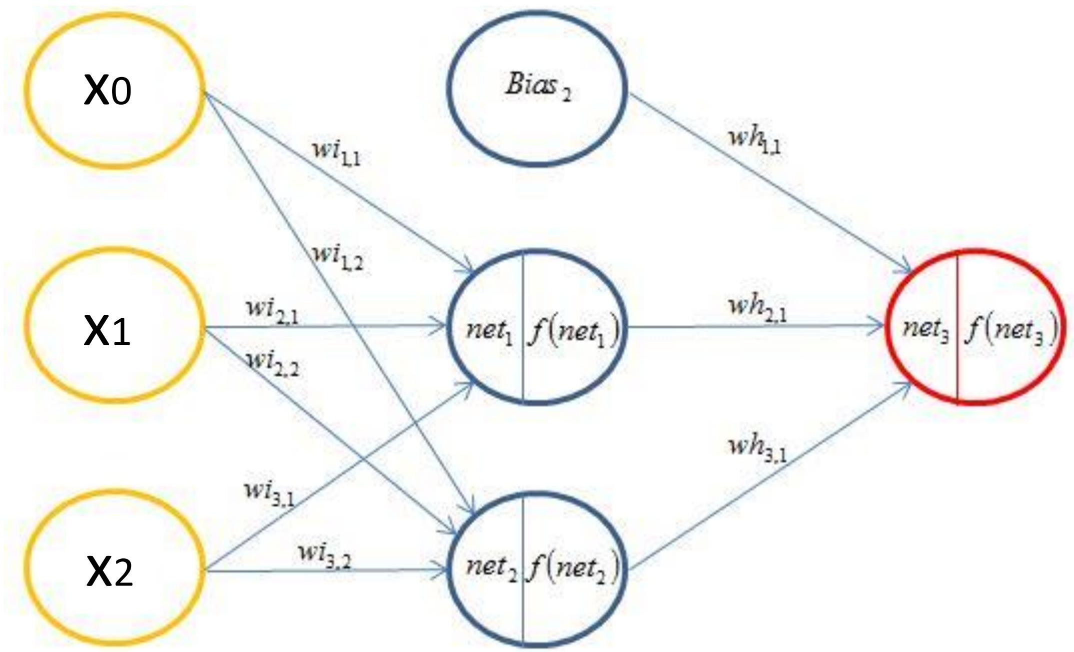
MULTI-NODE NEURON

Multi-layer perceptron / Mesh / Fully Connected



Neural network model

Neuron: Processing Unit



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

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

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

$net_i = \theta^T x = w^T x$

$f(net_i) = a_i^j$

Which activation function can we use?

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

$net_i = \theta^T x = w^T x$

$f(net_i) = a_i^j$

Activation functions

- A function used to determine output of each neuron (node)

- Types of activation functions
  - Linear Activation
  - Non linear activation
    - Sigmoid (logistic) function
    - Hyperbolic Tangent function (Tanh)
    - Rectified Linear Unit (Relu)
    - Leaky ReLU

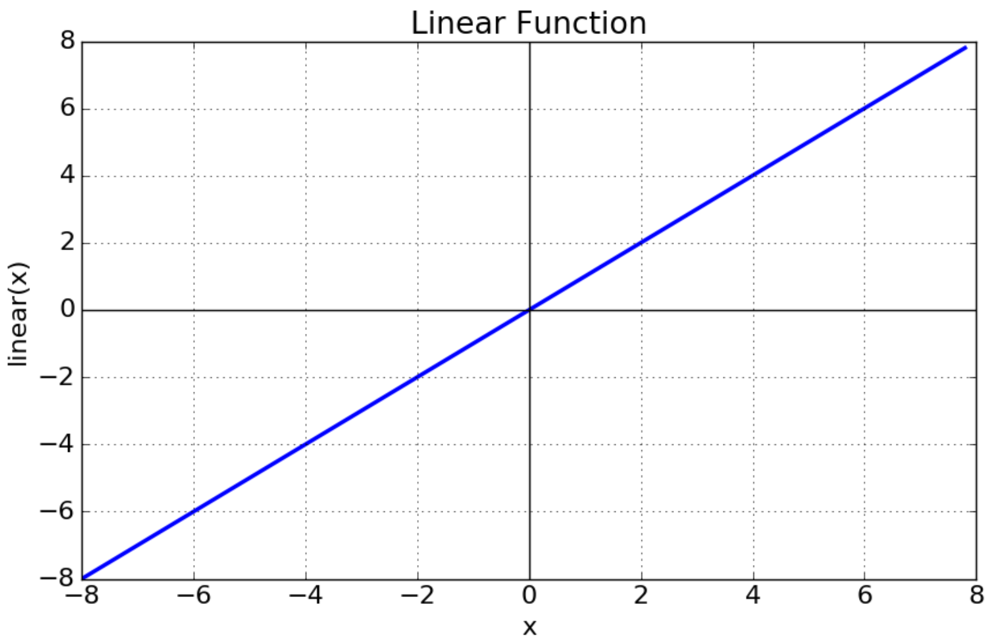
https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Linear Activation functions

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

$net_i = \theta^T x = w^T x$

$$f(net_i) = a_i^j$$



Equation : f(x) = x

Range : (-infinity to infinity)

It doesn’t help with the complexity or various parameters of usual data that is fed to the neural networks.

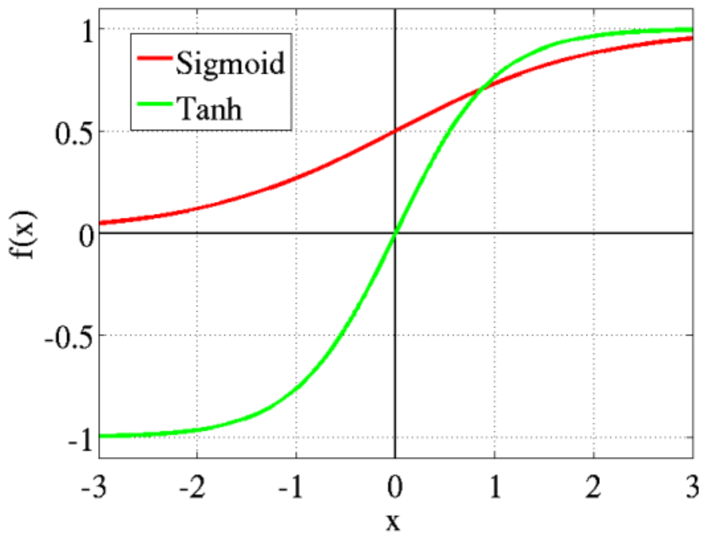
https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

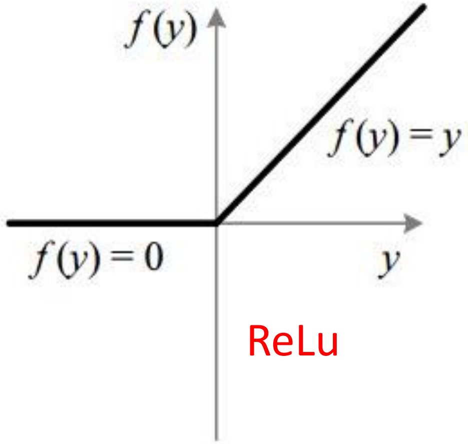
Nonlinear Activation functions

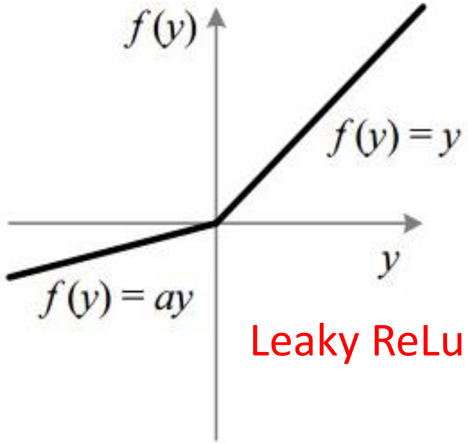
$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$net_i = \theta^T x = w^T x$

$f(net_i) = a_i^j$







Sigmoid (Logistic) : f(x) = $\frac{1}{1+e^{-z}} = \frac{1}{1+e^{-\theta^T x}}$

Tan hyperbolic : f(x) = tanh(x)

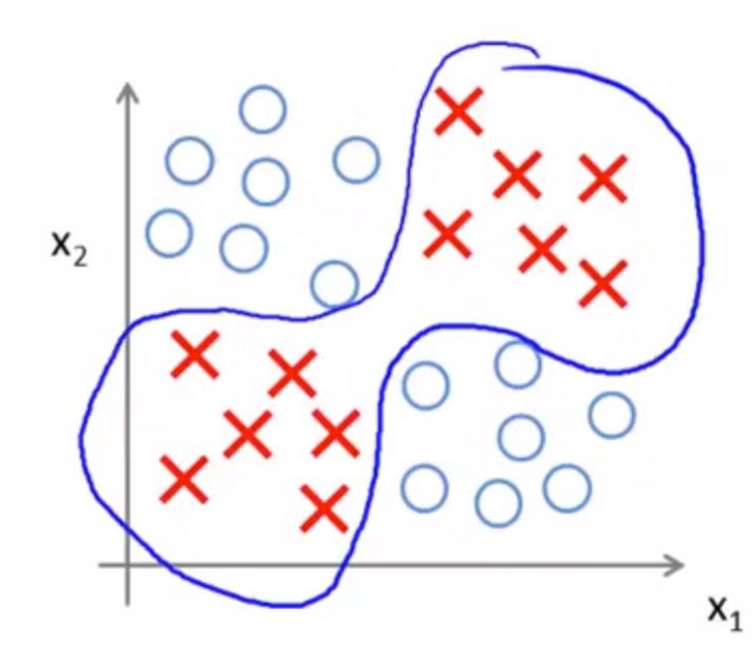
ReLU : f(z) = $\max(0, z)$

Leaky ReLU: f(y) = $\begin{cases} ay & y < 0 \\ y & y \geq 0 \end{cases}$

Neural model example

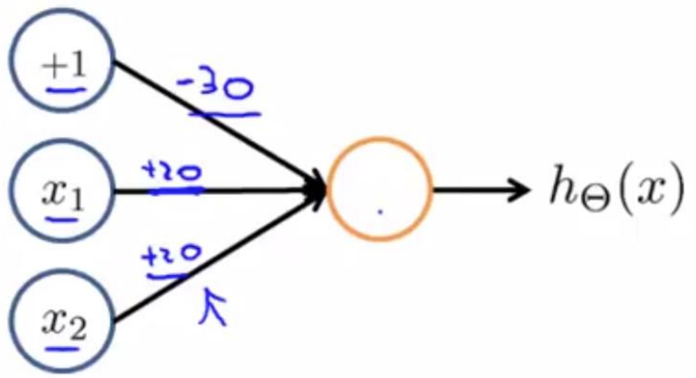
Neural model example

Sigmoid: $f(x) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-\theta^T x}}$

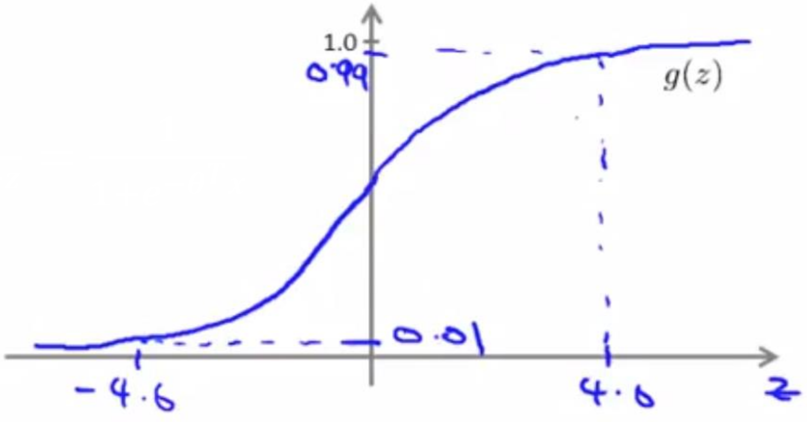


Simple example: AND

- $x_1, x_2 \in \{0, 1\}$
- $y = x_1 \text{ AND } x_2$



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

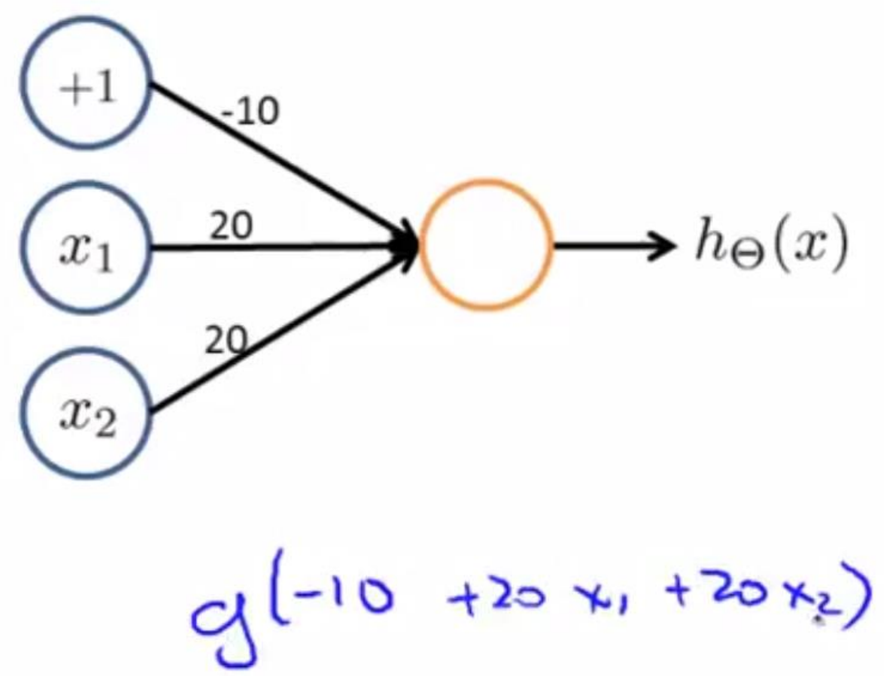


x_1	x_2	$h_\Theta(x)$
0	0	
0	1	
1	0	
1	1	

Neural model example

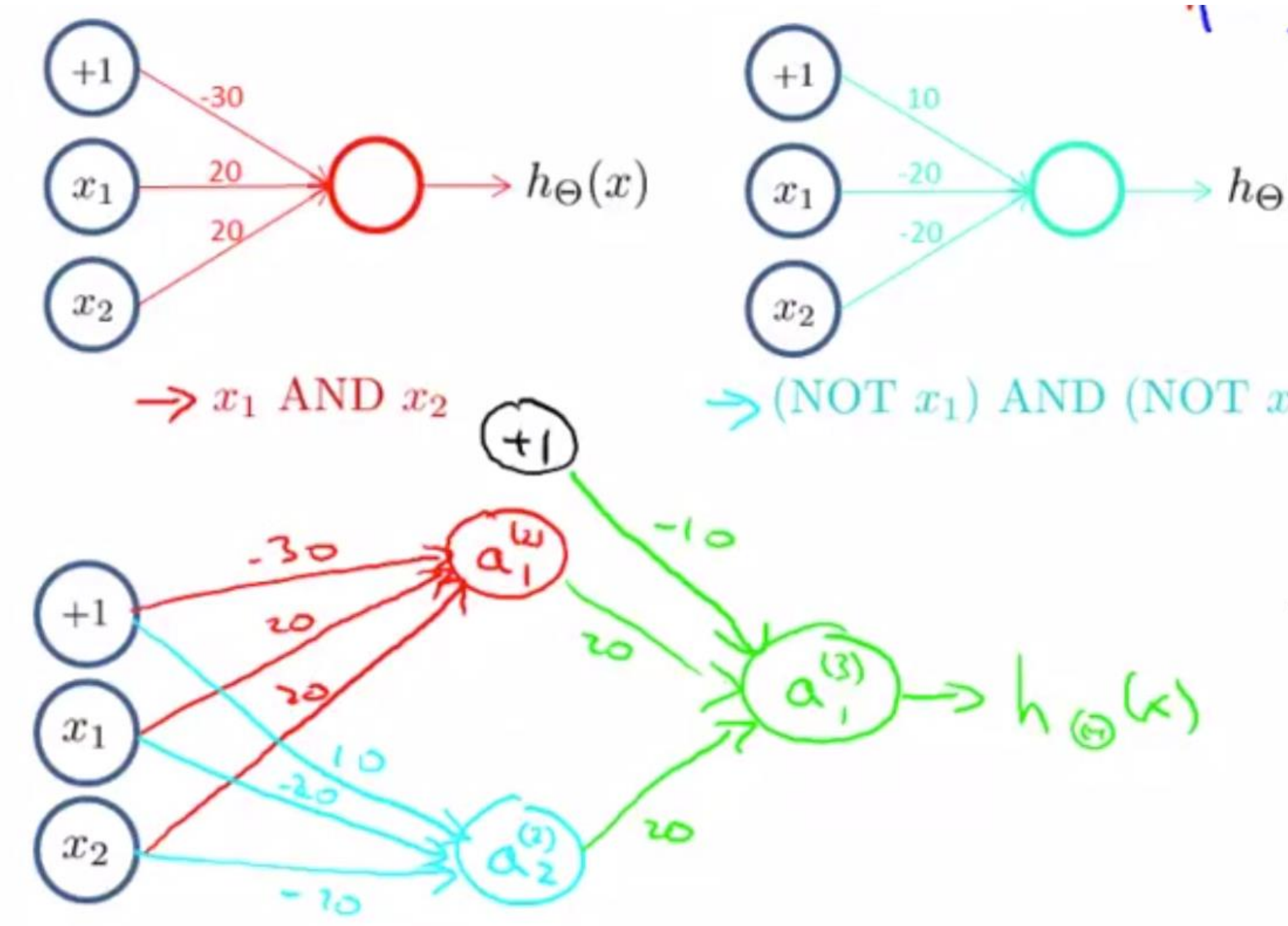
$$\text{Sigmoid: } f(x) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-\theta^T x}}$$

What is the result of this Neural Node?



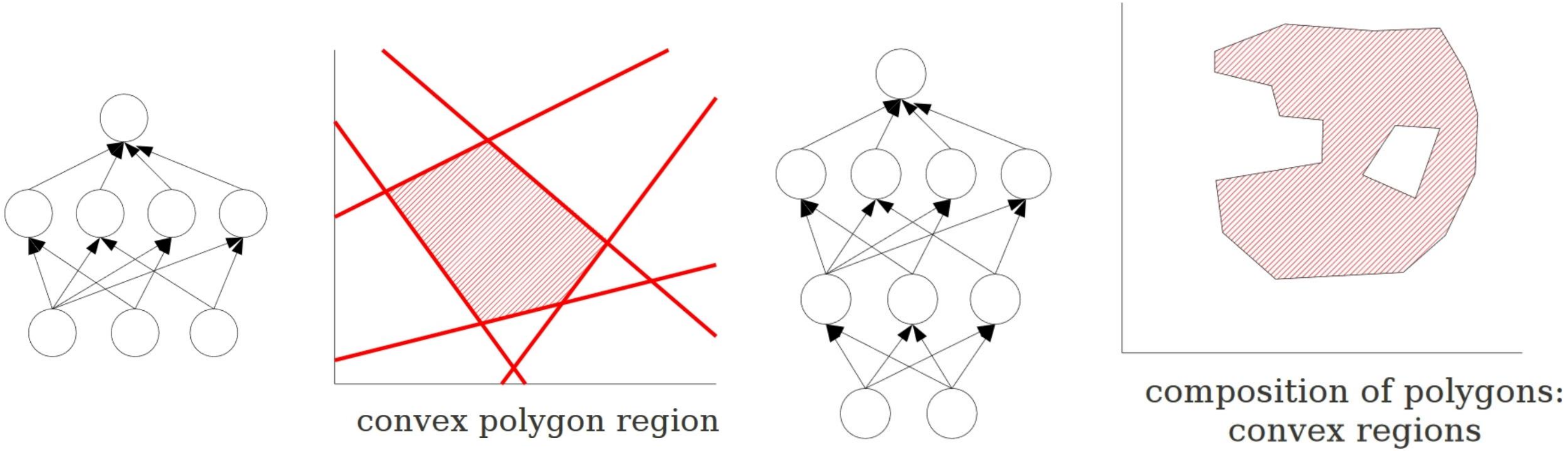
x_1	x_2	$h_\Theta(x)$
0	0	
0	1	
1	0	
1	1	

Neural model example



x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\Theta(x)$
0	0			
0	1			
1	0			
1	1			

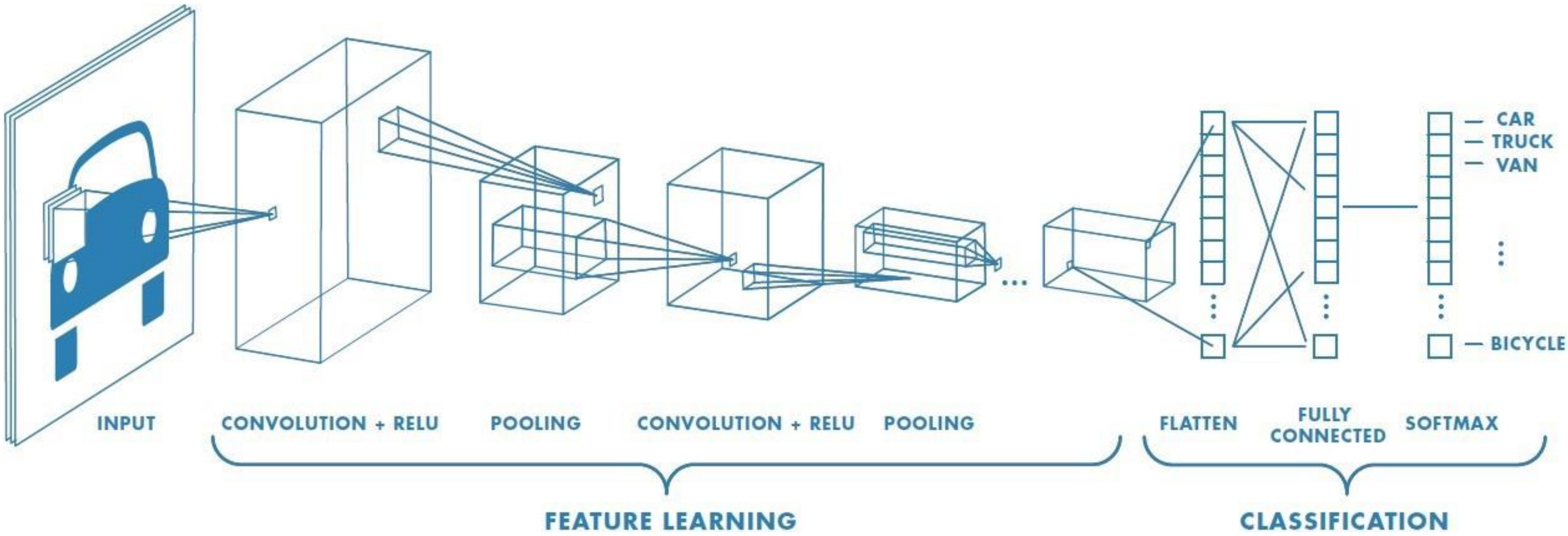
Neural model generate complex hyperplane decision



Conv Neural Network

MODEL STRUCTURE

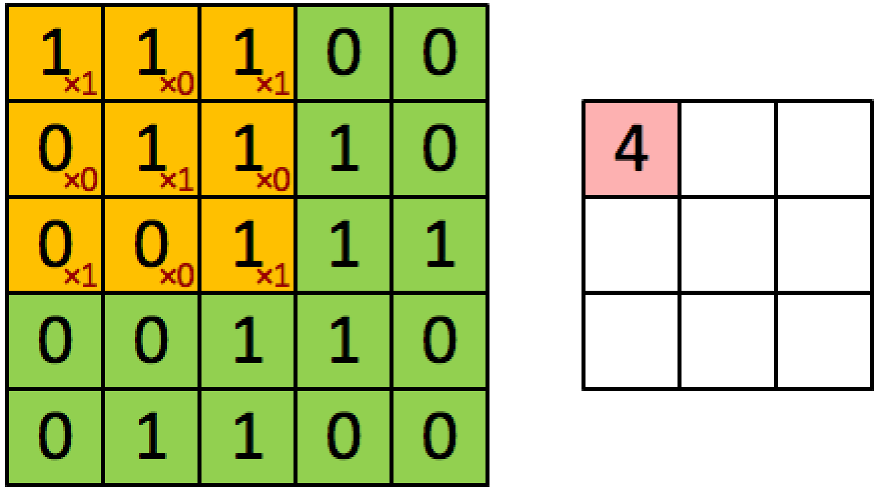
Convolutional Neural network(CNN) terminology



Convolutional Neural network(CNN) terminology

- **Convolution**
- **Convolutional mask (kernel)**
- **Edge detection mask**
- **Padding**
- **Convolution Stride**
- **Convolutions over volume**
- **Pooling**

CNN terminology: convolution



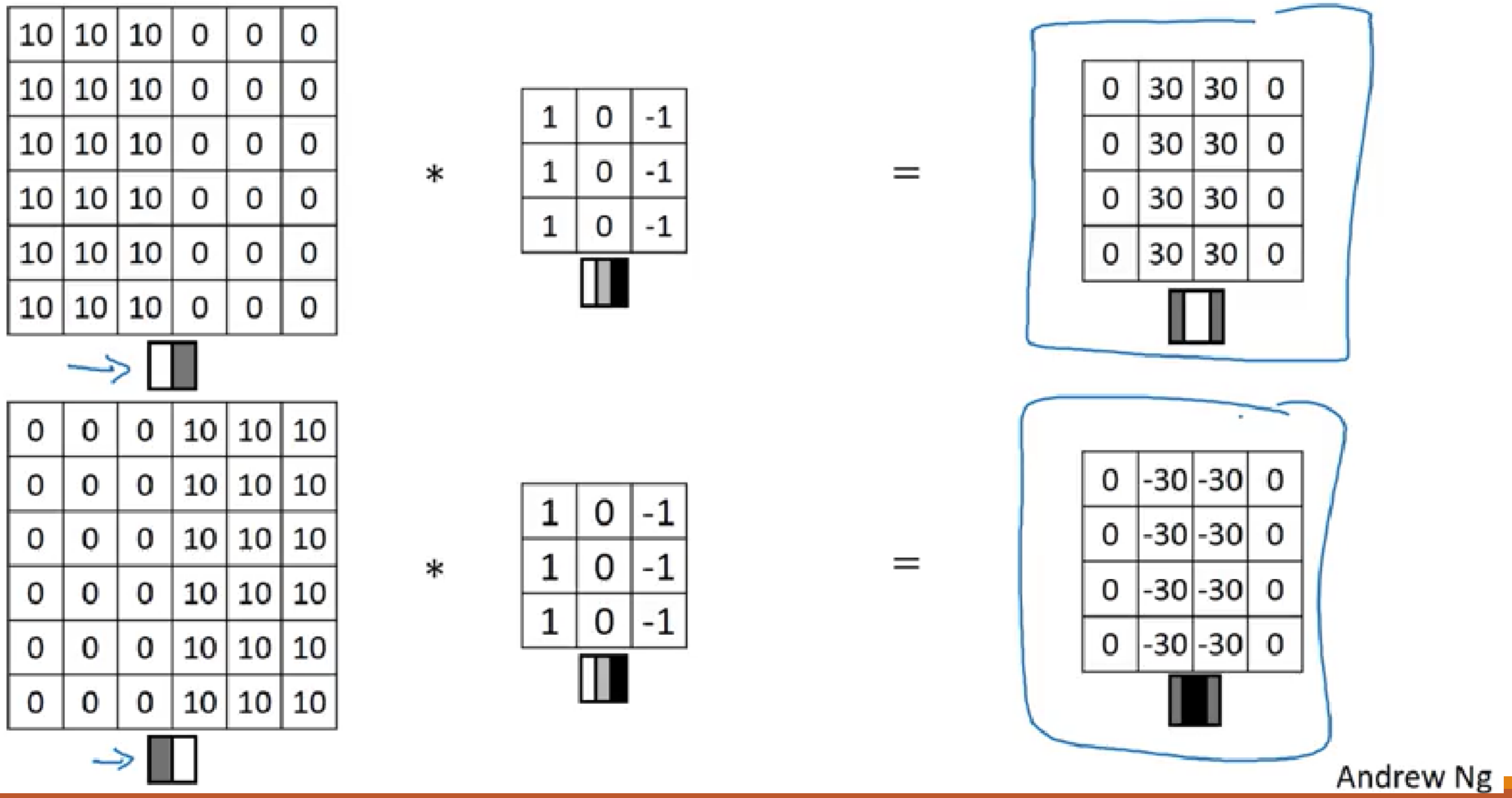
Image

Convolved Feature

Convolution

- **Operation:**
  - **Dot Product or**
  - **Weighted sum**
- **Task**
  - **Local pattern detection**
    - **Search for a particular local pattern in input**

CNN terminology: convolution

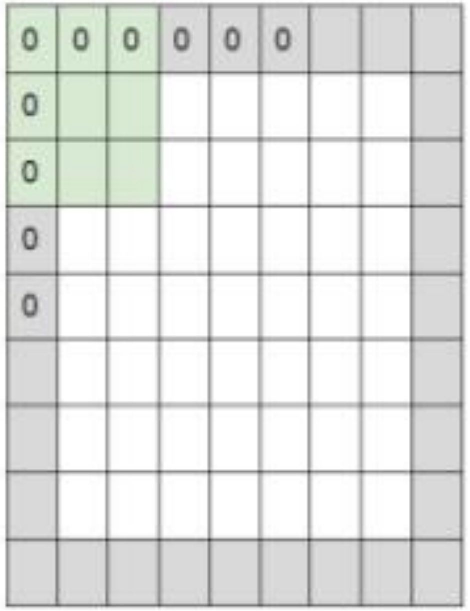


Ex.

- **Vertical Local pattern detection**
- **Sign of convolution results**
  - **According to convolutional mask**
- **Adaptive mask**
  - **Learning from data or domain problem**

Convolution **padding**

Zero Padding the border



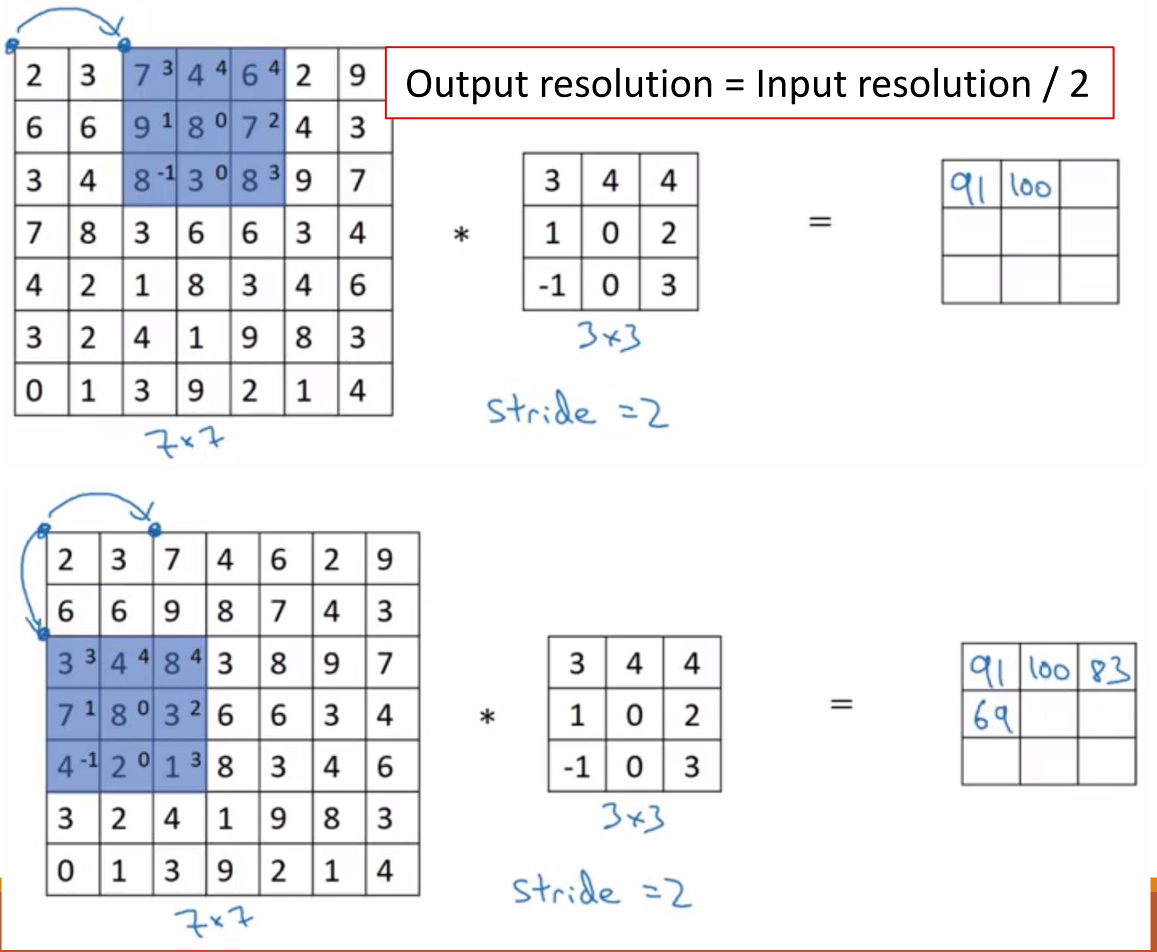
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Fig Credit: A Karpathy, CS231n

Padding
- **Boundary extending**
  - **In order to maintain convolution results the same dimension as input**
- **Padding size**
  - **Depend on mask size**
- **Padding value**
  - **Zero (mostly used)**
  - **Reflected border**
  - **Circular index**

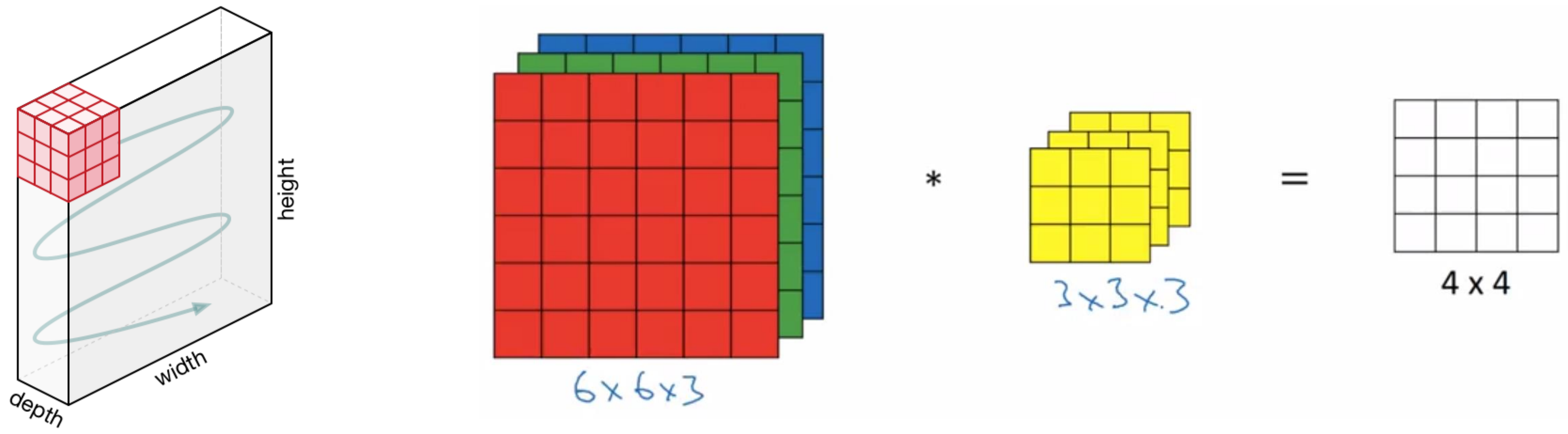
Convolution **Stride**



Stride

- Convolution resolution selection
  - Stride = 1
    - Convolution on every input position
      - Reserve output resolution = input resolution
  - Stride > 1
    - Convolution skipping
      - Output resolution < Input resolution

Convolution **over volume**



3 input planes / 3 convolutional masks / 1 output

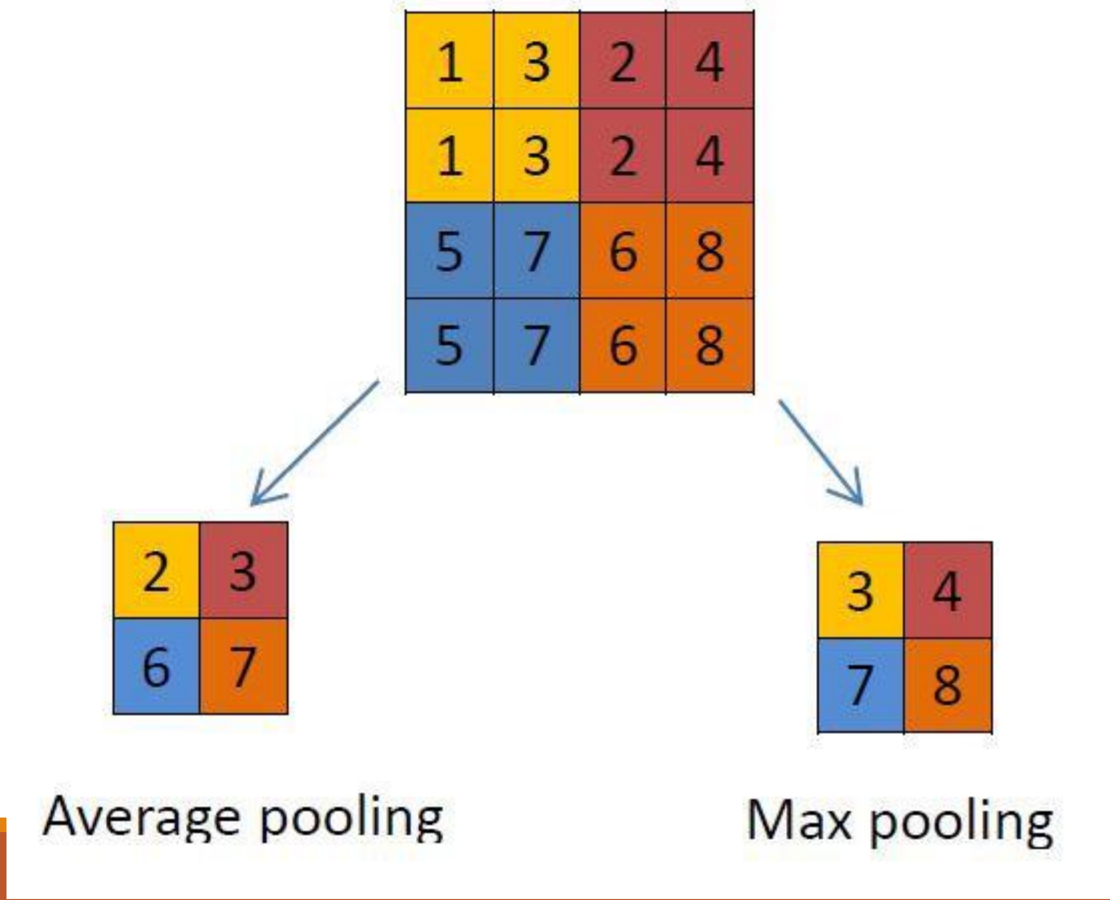
Output Results (with no zero padding example) = 4x4

- **= Red plane (1) * mask (1) + Green plane (2) * mask (2) + Blue plane (3) * mask (3)**

Sum all detected local pattern from all Red / Green / Blue planes

- **Return a single output of all detected patterns in input planes**

Convolution **Pooling**



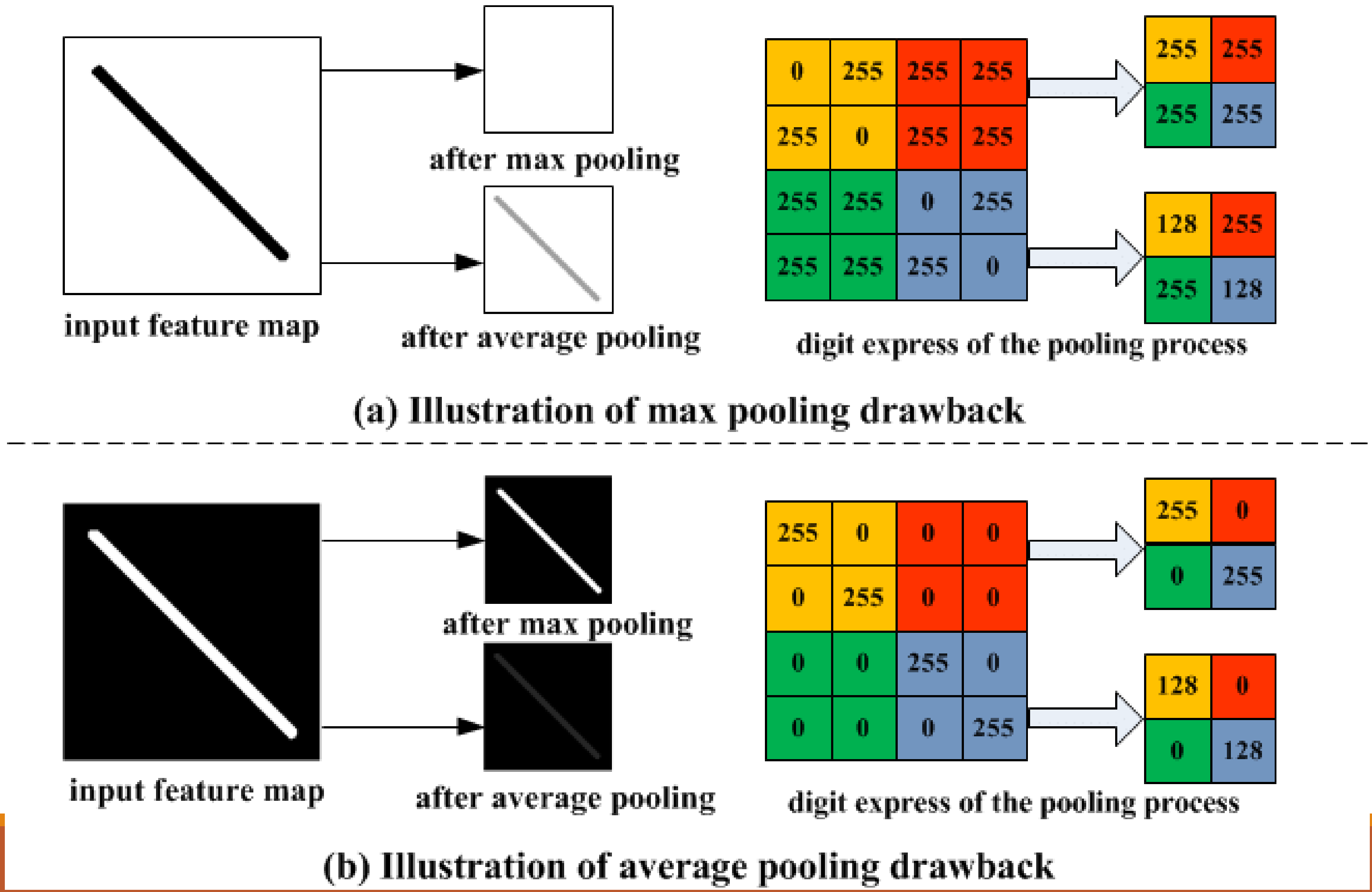
Reduce the dimensionality and the number of parameters and computation in the network.

- This shortens the training time and controls overfitting.

Pooling functions

- Max pooling
- Average pooling

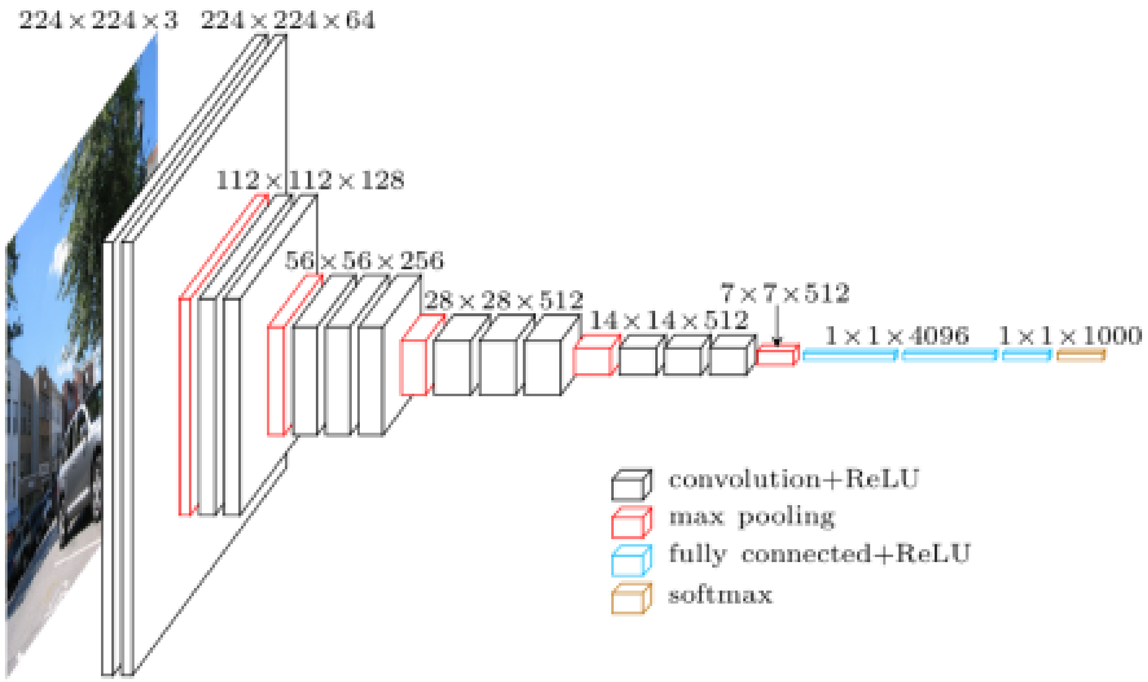
Convolution **Pooling**



(a) Illustration of max pooling drawback

(b) Illustration of average pooling drawback

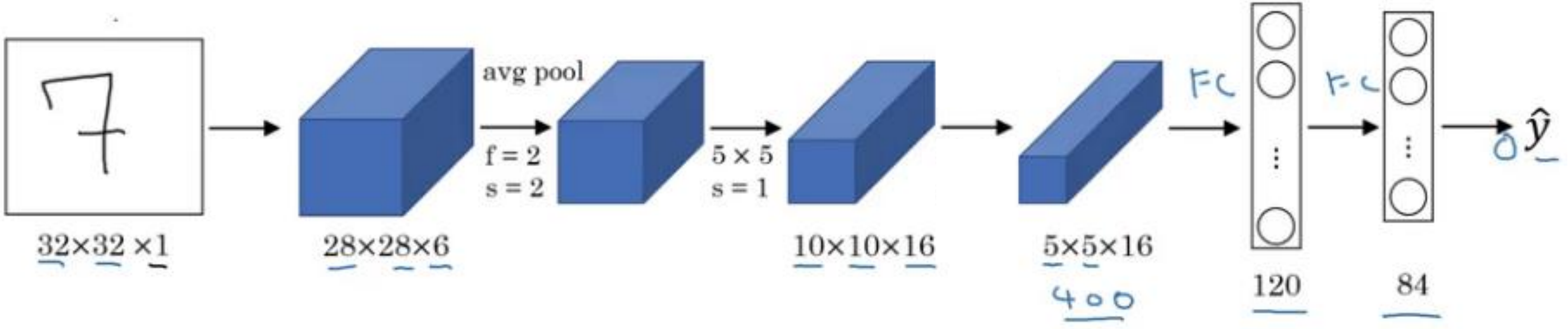
Let’s play with CNN structure (Lenet)



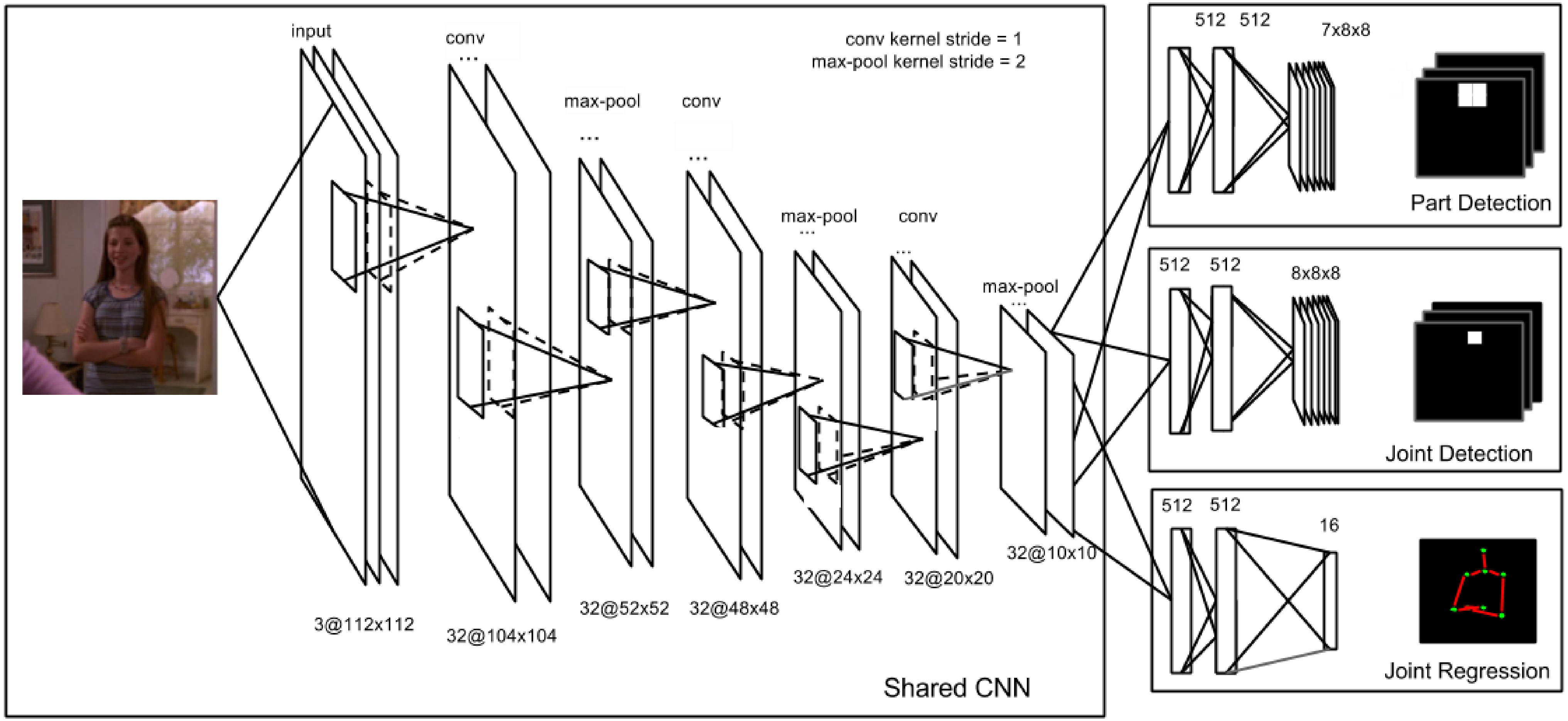

```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K   weights: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]  memory:  14*14*512=100K   weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
```

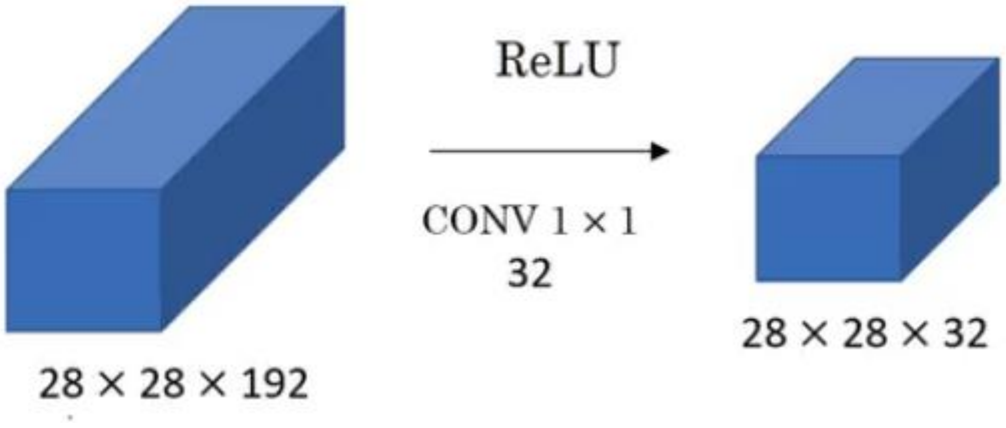
Let's play with CNN structure (Lenet)



Let’s play with CNN structure (PoseNet)



Let’s play with CNN structure (resnet)



The basic idea of using 1 X 1 convolution is to reduce the number of channels from the image. A couple of points to keep in mind:

We generally use a pooling layer to shrink the height and width of the image

To reduce the number of channels from an image, we convolve it using a 1 X 1 filter (hence reducing the computation cost as well)

DEEP NETWORK

MODEL CONFIGURATION

Model Configurations

Network Parameters

Layers

Nodes / Layers

Type of Activation for each nodes

- Sigmoid / Tanh / ReLU / Leakey ReLU / etc.

Optimizer Parameters

Optimizer

- -> Adam / SGD / Adadelta / etc.

Loss Function

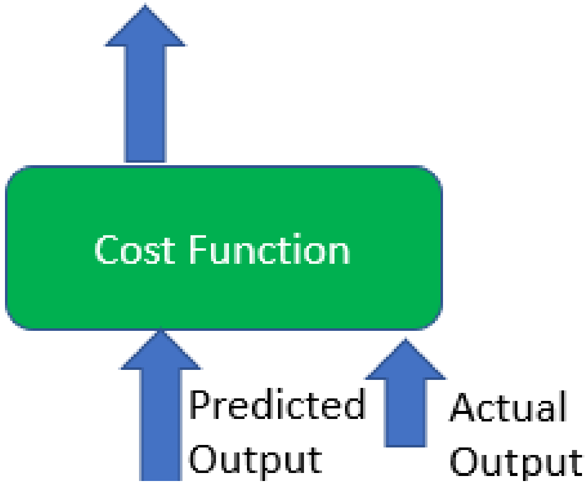
- -> L2 (square error) / RMSE (Root Mean Square Error)/ categorical_crossentropy

Optimizer parameters

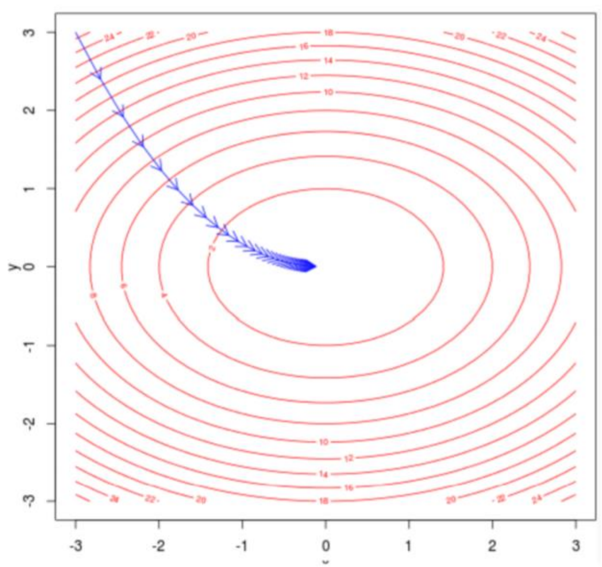
- -> Learning rate / Epoch / Batch / Iteration

Optimizer

Error = Actual Output-predicted output

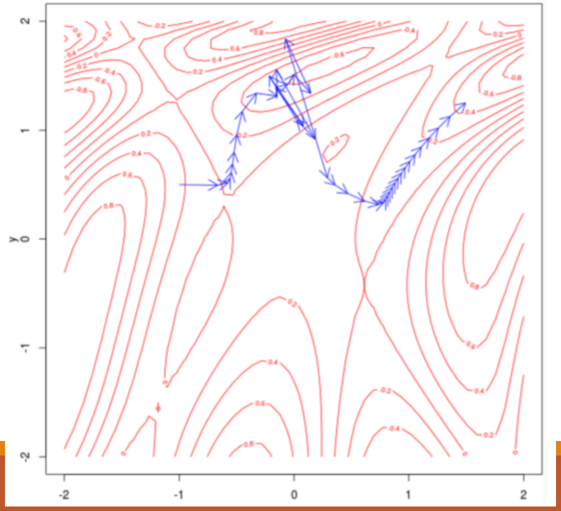


Minimum Error (Loss / Cost)



Theory

Real Applications



What should be the type of Optimizer?

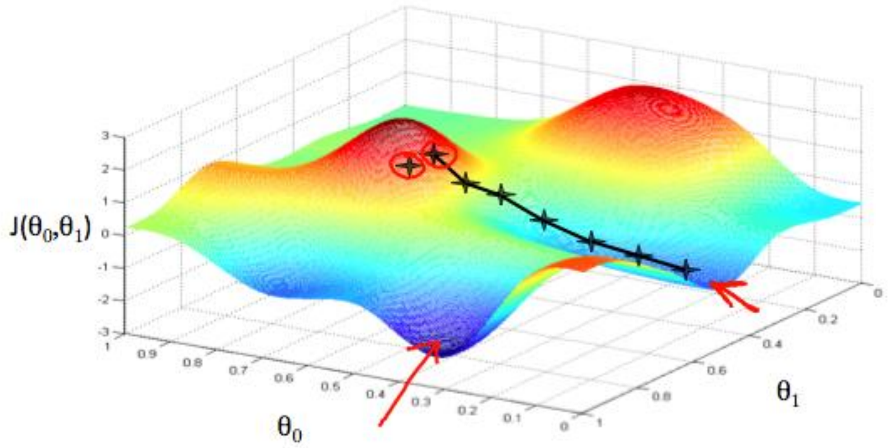
ADAM / STOCHASTIC GRADIENT DECENT / ADADELTA / ETC.

Why do we need to know activation derivative function?

Derivative form of activation function

- Used in optimization process of gradient descent search

Optimizer looks for **directions of next weights to move** for next search iteration



The gradient descent algorithm is:

repeat until convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

Stochastic Gradient Descent (SGD)

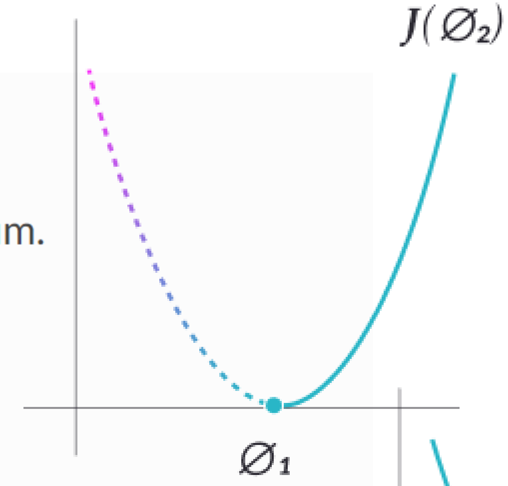
```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

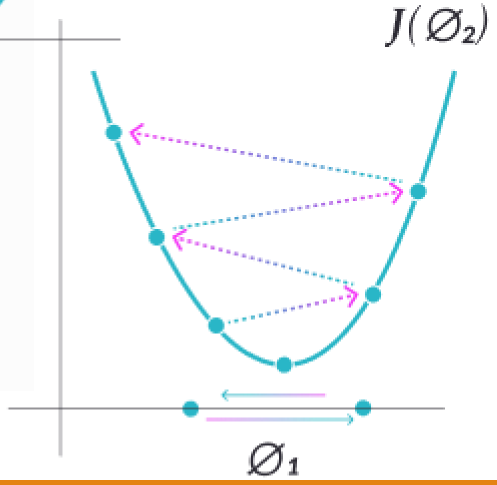
Stochastic gradient descent optimizer.

Includes support for momentum, learning rate decay, and Nesterov momentum.

Arguments

- **lr**: float >= 0. Learning rate.
- **momentum**: float >= 0. Parameter that accelerates SGD in the relevant direction and dampens oscillations.
- **decay**: float >= 0. Learning rate decay over each update.
- **nesterov**: boolean. Whether to apply Nesterov momentum.





SDG Optimizer Learning Rate (LR) Adjust

Constant LR

Time-based Decay ->

- k = hyperparameter (decay) / t = # epochs
- Momentum -> [0.5, 0.9] `lr = lr0/(1+kt)`

Step Decay -> lr0 / t

Image 2: SGD without momentum

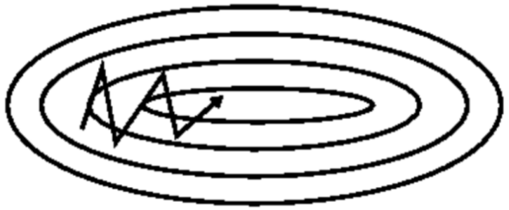


Image 3: SGD with momentum

```
learning_rate = 0.1
decay_rate = learning_rate / epochs
momentum = 0.8
```

Exponential Decay ->

```
lr = lr0 * e^(-kt)
```

```
def exp_decay(epoch):
   initial_lrate = 0.1
   k = 0.1
   lrate = initial_lrate * exp(-k*t)
   return lrate
```

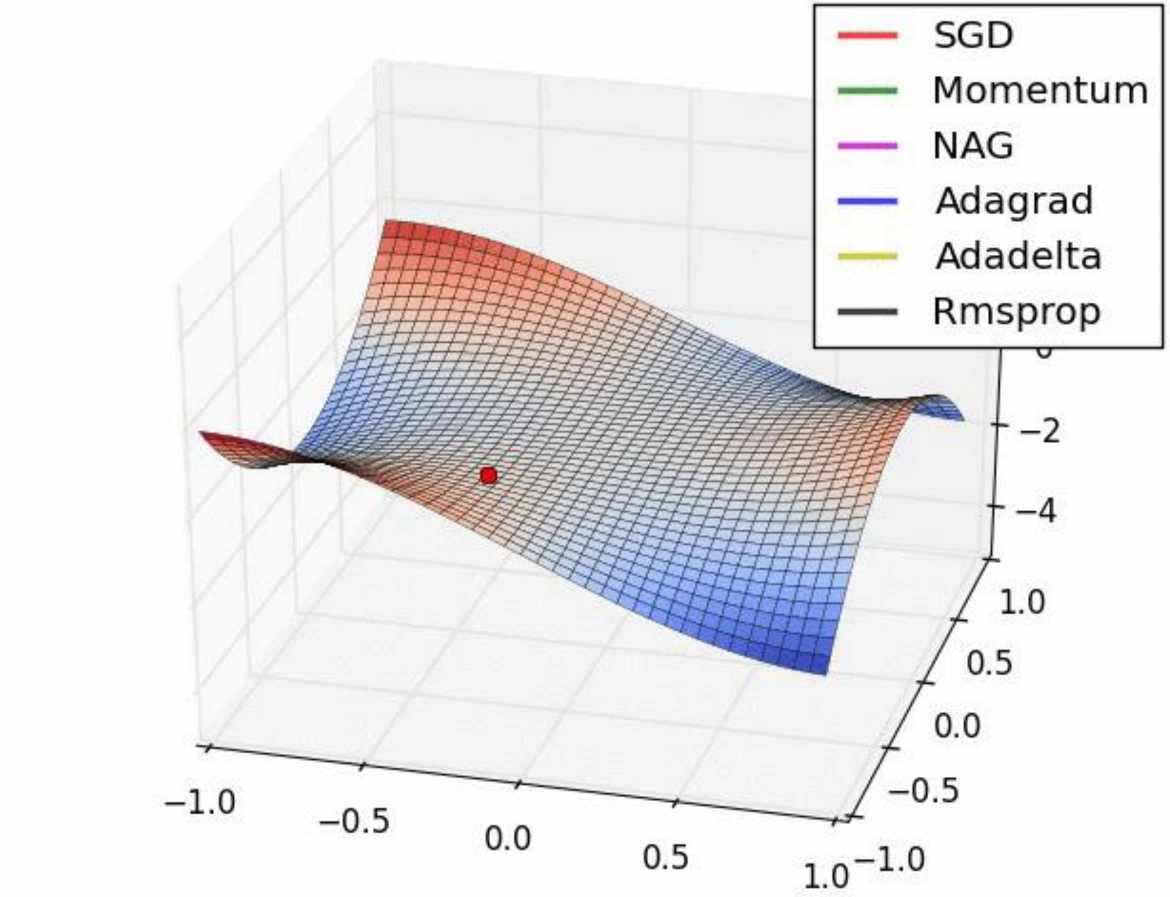
Optimizer with Adaptive Learning Rate

Adagrad: Adaptive Gradient

Adadelta

RMSprop

Adam



Optimizer Comparison

SGD Momentum

- Manual tuning
  - Fixed learning rate
  - Exponential decay of time step t
    - always Decreasing
    - Need initial LR

Adagrad

- Adaptive tuning
  - Adjusting with
    - Global LR
      - time step t
      - past gradients
    - always Decreasing
    - Need initial LR

Adadelta

- Adaptive tuning
  - Adjusting with
    - Individual LR
      - Time step t
      - Past gradient
    - No initial LR

Adam

- Adaptive tuning
  - Adjusting with
    - Individual LR
      - Time step t
      - Past gradient
      - Momentum
- Normally default optimizer

Accuracy Drop when increasing epoch

Adaptive Learning Rate Methods

- Adaptive gradient descent algorithms such as Adagrad, Adadelta, RMSprop,Adam, provide an alternative to classical SGD.

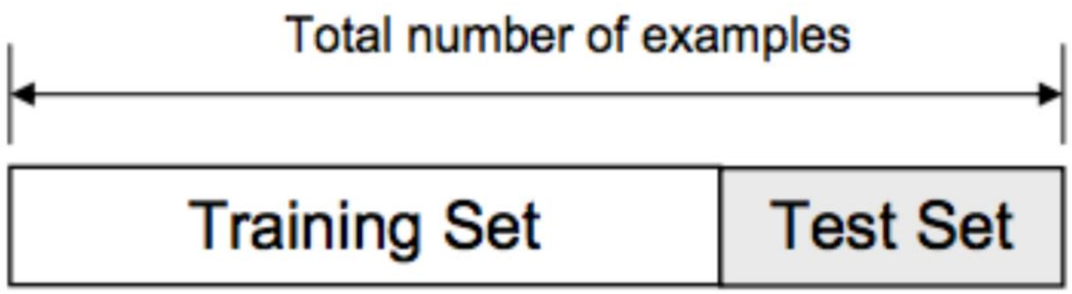
keras.optimizers.Adagrad(lr=0.01, epsilon=1e-08, decay=0.0)

keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=1e-08, decay=0.0)

keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)

keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)

Train/test splitting



Partition Dataset into

- Model selection: using XTrain
  - Usually 70% Training
    - Evaluate on various model learning parameters

X_{train} *for fitting model*
X_{test} *for prediction test*

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
```

- Model tolerance evaluation: XTest
  - Usually 30%

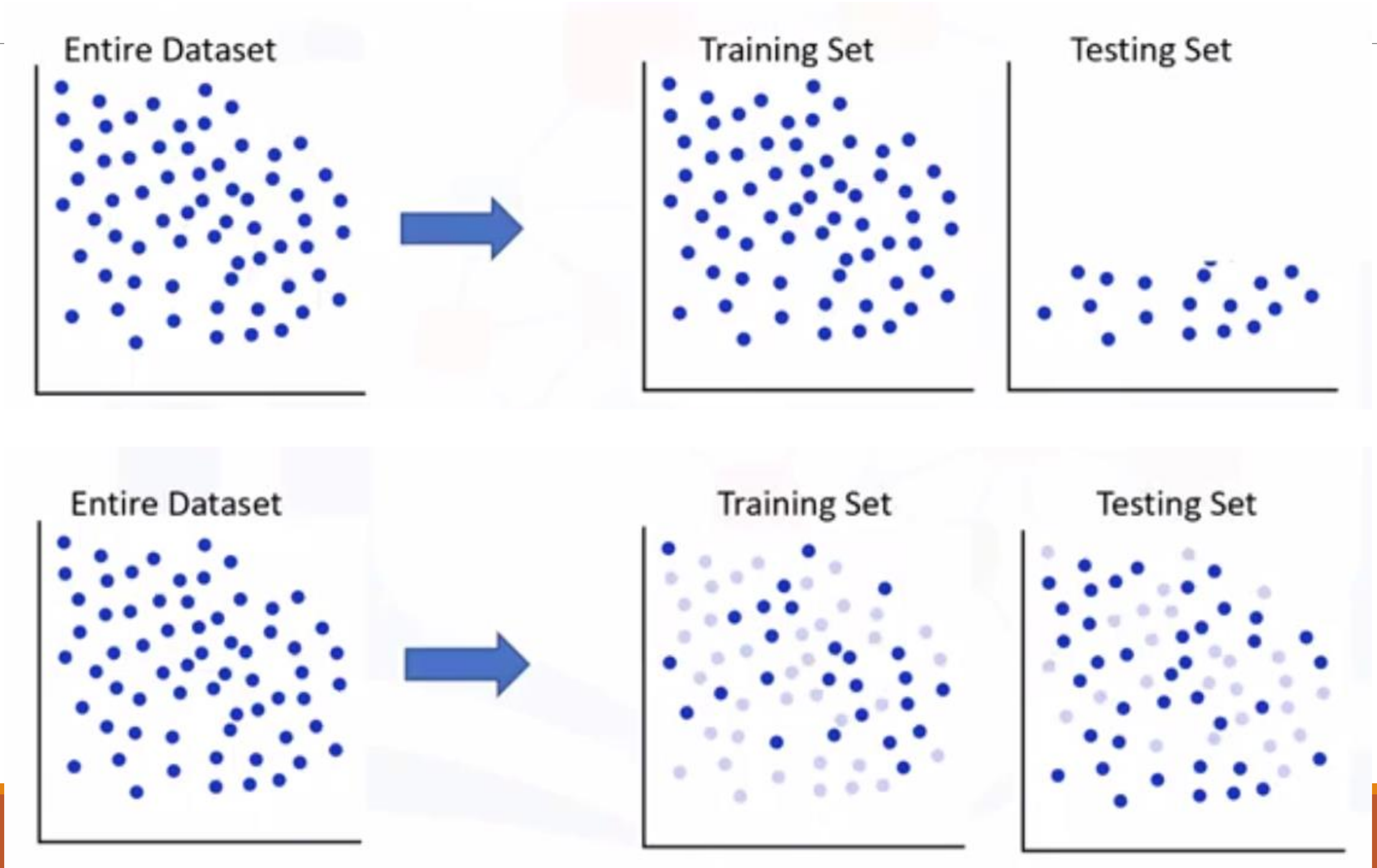
```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3)
```

Training error is also known as In-Sample-Error(ISE)
Testing error is also known as Out-Of-Sample-Error(OSE)

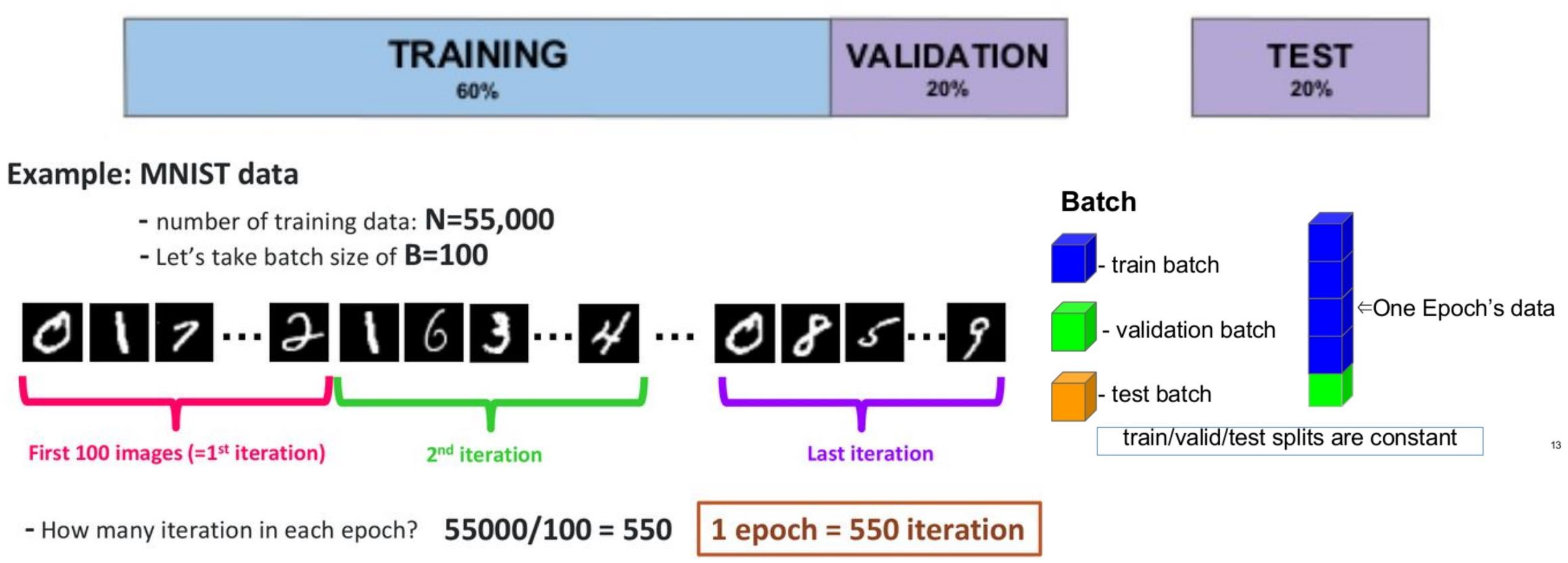
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=42)
```

Random_state: identify seed value to always fix random result

Equally distributed Tran/test using Stratified_split

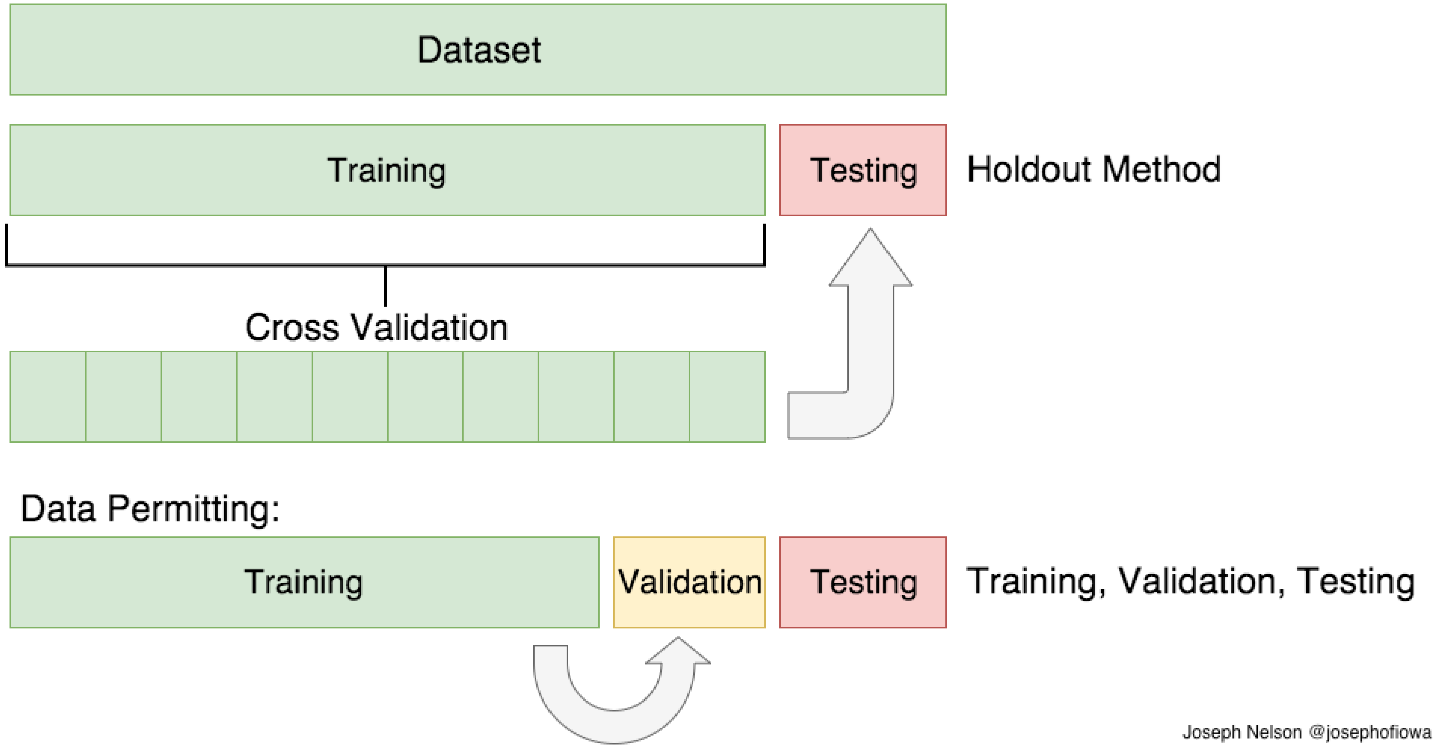


Epoch vs Iteration vs Batch size



Cross validation (CV)

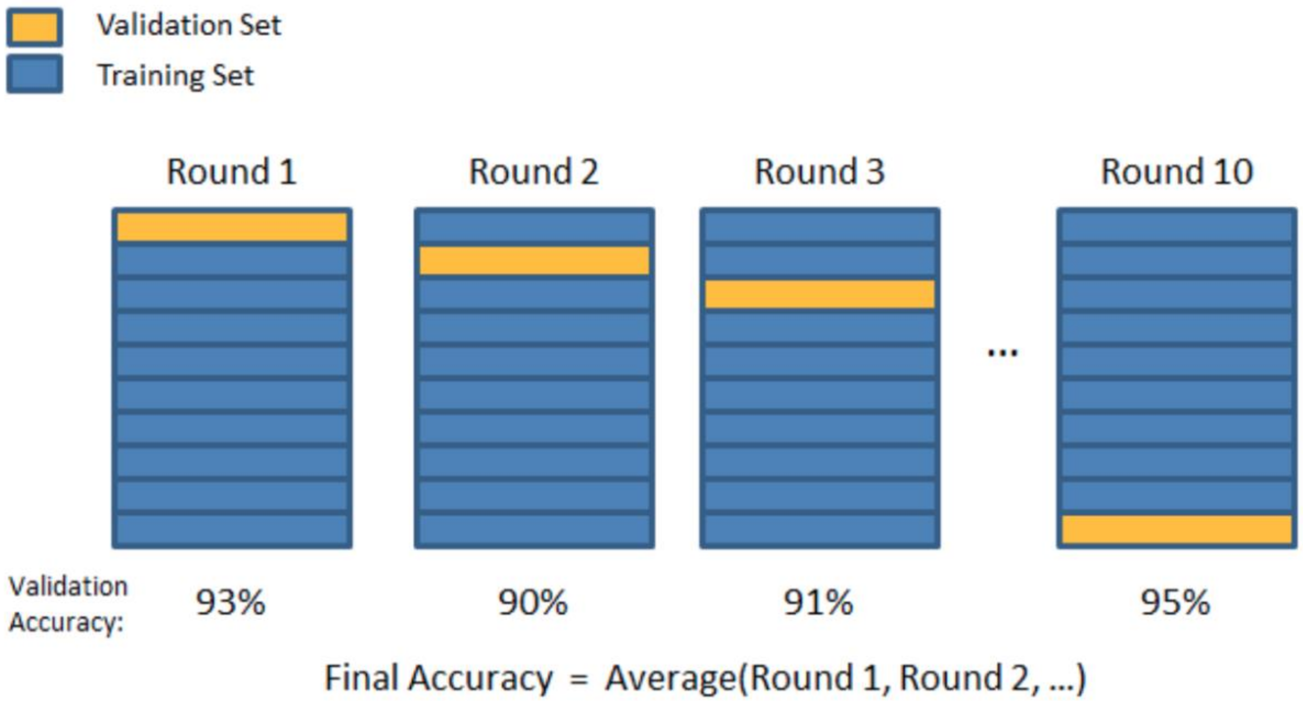
Partition Dataset into
- Cross Validation (CV)
  - K-Fold CV
  - Leave one out (LOOCV)
  - Shuffle Split CV



K-fold cv

K-Fold Cross Validation

- Partition Training into k subsets
  - For i=1:k iteration
    - Select ith subset as test data
    - k-1 subsets are used to train



K-fold cv

K-Fold Cross Validation

- Partition Training into k subsets
  - For i=1:k iteration
    - Select ith subset as test data
    - k-1 subsets are used to train

```
from sklearn import model_selection

seed = 7
kfold = model_selection.KFold(n_splits=3, Shuffle = True,
random_state=seed)

score = model_selection.cross_val_score(model, Xtrain, Ytrain,
cv=kfold)
```

Note: Shuffle = True เพื่อ random data before K-Fold partition

score = array of k-accuracy
score.mean() = Average Accuracy
score.std() = Std Accuracy

Classification performance evaluation

Classification performance metrics

- ❖ Choice of metrics influences
  - how the performance of machine learning algorithms
    - is measured and compared.

- ❖ Types of Metrics
  - Confusion Matrix
  - Precision
  - Recall
  - Accuracy
  - F1 Score
  - ROC curve / AUC curve
  - Log-Loss

Confusion matrix: Binary classification

Confusion Matrix	Predicted class	
Actual Class	Class = Yes	Class = No
Class = Yes	True Positive	False Negative
Class = No	False Positive	True Negative

True Positive (TP) -> Green

True Negative (TN) -> Green

False Negative (FN) -> Red (Result should say 'Yes' but return 'No'

False Positives (FP) -> Red (Results should say 'No' but return 'Yes'

- It measures statistics according to hypothesis test
  - Desired vs undesired classes
  - Ex. Hypothesis
    - *When a person is having cancer :*
      - ***Yes***
    - *When a person is NOT having cancer*
      - ***No***

Confusion matrix: Binary classification

Confusion Matrix	Predicted class		
Actual Class		Class = Yes	Class = No
	Class = Yes	True Positive	False Negative
	Class = No	False Positive	True Negative

- FN: Actual (Yes) ≠ Predicted (No)
  - *A person having cancer and*
  - *the model classifying his case as No-cancer*
- FP: Actual (No) ≠ Predicted (Yes)
  - *A person NOT having cancer and*
  - *the model classifying his case as cancer*

- TP: Actual = Predicted = *Yes*
  - *where a person is actually having cancer(1) and*
  - *the model classifying his case as cancer(1)*
- TN: Actual = Predicted = *No*
  - *where a person NOT having cancer and*
  - *the model classifying his case as Not cancer*

Confusion matrix: Binary classification

Confusion Matrix	Predicted class		
Actual Class		Class = Yes	Class = No
	Class = Yes	True Positive	False Negative
	Class = No	False Positive	True Negative

- What should be minimized?
  - *No discrete rules*
  - *Depend on the business need*

- Minimizing FN in cancer detection
  - Missing a cancer patient will be a huge mistake as no further examination will be done on them
- Minimizing FP in spam email detection
  - the Model classifies that important email that you are desperately waiting for, as Spam

How about Banking business with face recognition for loging into the bank account?

Confusion matrix : Binary classification

Confusion Matrix	Predicted class		
Actual Class		Class = Yes	Class = No
	Class = Yes	True Positive	False Negative
	Class = No	False Positive	True Negative

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Accuracy should **NEVER be used** as a measure when the target variable classes in the data are a **majority of one class**.

- Patient with cancer = 5
- Patient with no cancer = 95

Confusion matrix : Binary classification

Confusion Matrix	Predicted class		
Actual Class		Class = Yes	Class = No
	Class = Yes	True Positive	False Negative
	Class = No	False Positive	True Negative

Precision = TP/TP+FP

Recall = TP/TP+FN

- **When should we use Precision or Recall?**
  - **Precision**
    - looking mainly on **Correct Prediction Rate**
    - **Pay attention on FP**
  - **Recall**
    - looing on getting **all desired class samples**
    - **Pay attention on FN**

Confusion matrix : Binary classification

Confusion Matrix	Predicted class		
Actual Class		Class = Yes	Class = No
	Class = Yes	True Positive	False Negative
	Class = No	False Positive	True Negative

F1 Score = 2*(Recall * Precision) / (Recall + Precision)

F1 Score is the weighted average of Precision and Recall.

ROC curve can be used to select a threshold for a classifier which maximises the true positives, while minimising the false positives.

- **F1 Score**
  - **Combining Precision & Recall in one metric**
    - **In term of Harmonic Mean**
- **Log-Los**
  - **In term of entropy**

$$-(y\log(p) + (1-y)\log(1-p))$$

Confusion matrix : Binary classification

Confusion Matrix

	Predict (Fraud)	Predict (Not Fraud)
Actual (Fraud)	1	2
Actual (Not Fraud)	0	97

- **What should be pay attention?**
  - **In Credit card business**
    - **Accuracy**
    - **Precision**
    - **Recall**
    - **F1-score**

Confusion matrix : multi-class classification

Multi-class

	Predicted		
Actual		A	B	C
	A	TP_A	E_{AB}	E_{AC}
	B	E_{BA}	TP_B	E_{BC}
	C	E_{CA}	E_{CB}	TP_C

Predicted class

		P	N
Actual class	P	*TP*	*FN*
	N	*FP*	*TN*

	Predicted		
Actual		A	Not A
	A	TP_A	$E_{AB} + E_{AC}$
	Not A	$E_{BA} + E_{CA}$	TP_B + E_{BC} E_{CB} + TP_C

	Predicted		
Actual		B	Not B
	B	TP_B	$E_{BA} + E_{BC}$
	Not B	$E_{AB} + E_{CB}$	TP_A + E_{AC} E_{CA} + TP_C

	Predicted		
Actual		C	Not C
	C	TP_C	$E_{CA} + E_{CB}$
	Not C	$E_{AC} + E_{BC}$	TP_A + E_{AB} E_{BA} + TP_B

Confusion Matrix		Predicted Class 1	Class 2	Class 3	False Negative (FN)	Recall
Actual	Class 1	A	B	C	B + C	A/(A + B + C)
	Class 2	D	E	F	D + F	E/(D + E + F)
	Class 3	G	H	I	G + H	I/(G + H + I)
	False Positive (FP)	D + G	B+H	C+F	Overall Accuracy = A + E + I/(Sum of red and green squares)	
	Precision	A/(A + D + G)	E/(B + E + H)	I/(C + F + I)		

True positives True Negatives Misclassified cases False Positives alse Negatives.

	A	B	C	
A	2	2	0	4
B	1	2	0	3
C	0	0	3	3
	3	4	3	

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$