University of Idaho

CS 502
Directed Studies: Adversarial Machine Learning

Dr. Alex Vakanski

Lecture 2

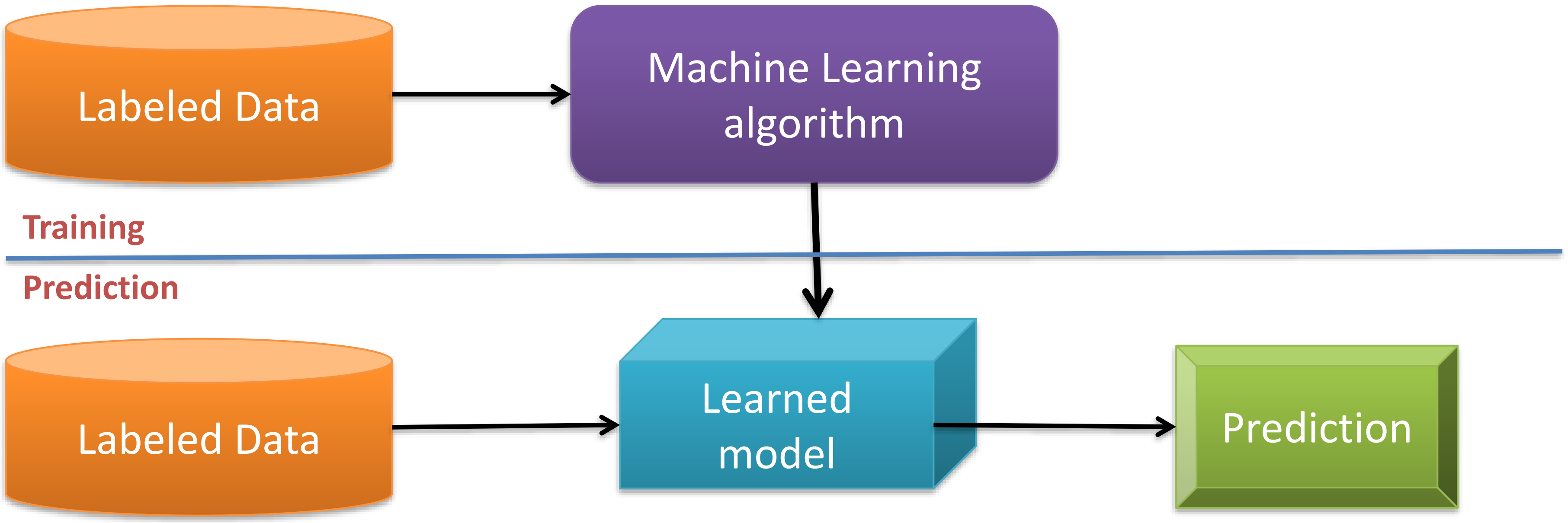
Deep Learning Overview

Lecture Outline

- Machine learning basics
  - Supervised and unsupervised learning
  - Linear and non-linear classification methods
- Introduction to deep learning
- Elements of neural networks (NNs)
  - Activation functions
- Training NNs
  - Gradient descent
  - Regularization methods
- NN architectures
  - Convolutional NNs
  - Recurrent NNs

Machine Learning Basics

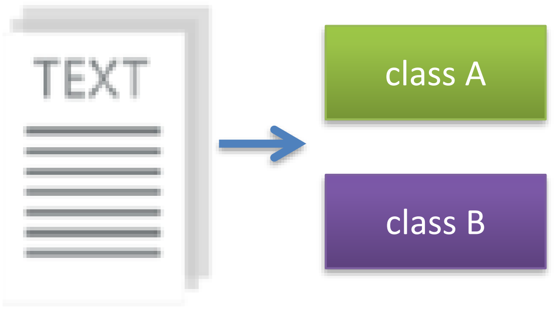
- ***Artificial Intelligence*** is a scientific field concerned with the development of algorithms that allow computers to learn without being explicitly programmed
- ***Machine Learning*** is a branch of Artificial Intelligence, which focuses on methods that learn from data and make predictions on unseen data



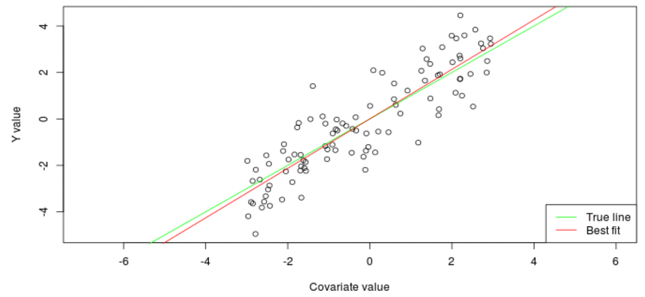
Picture from: Ismini Lourentzou – Introduction to Deep Learning

Machine Learning Types

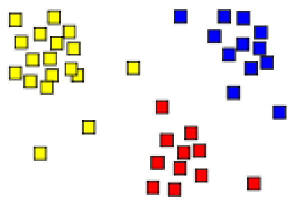
- ***Supervised***: learning with **labeled data**
  - – Example: email classification, image classification
  - – Example: regression for predicting real-valued outputs
- ***Unsupervised***: discover patterns in **unlabeled data**
  - – Example: cluster similar data points
- ***Reinforcement learning***: learn to act based on **feedback/reward**
  - – Example: learn to play Go



Classification



Regression


Clustering

Slide credit: Ismini Lourentzou – Introduction to Deep Learning

Supervised Learning

- *Supervised learning* categories and techniques
  - **Numerical classifier functions**
    - Linear classifier, perceptron, logistic regression, support vector machines (SVM), neural networks
  - **Parametric (probabilistic) functions**
    - Naïve Bayes, Gaussian discriminant analysis (GDA), hidden Markov models (HMM), probabilistic graphical models
  - **Non-parametric (instance-based) functions**
    - *k*-nearest neighbors, kernel regression, kernel density estimation, local regression
  - **Symbolic functions**
    - Decision trees, classification and regression trees (CART)
  - **Aggregation (ensemble) learning**
    - Bagging, boosting (Adaboost), random forest

Slide credit: Y-Fan Chang – An Overview of Machine Learning

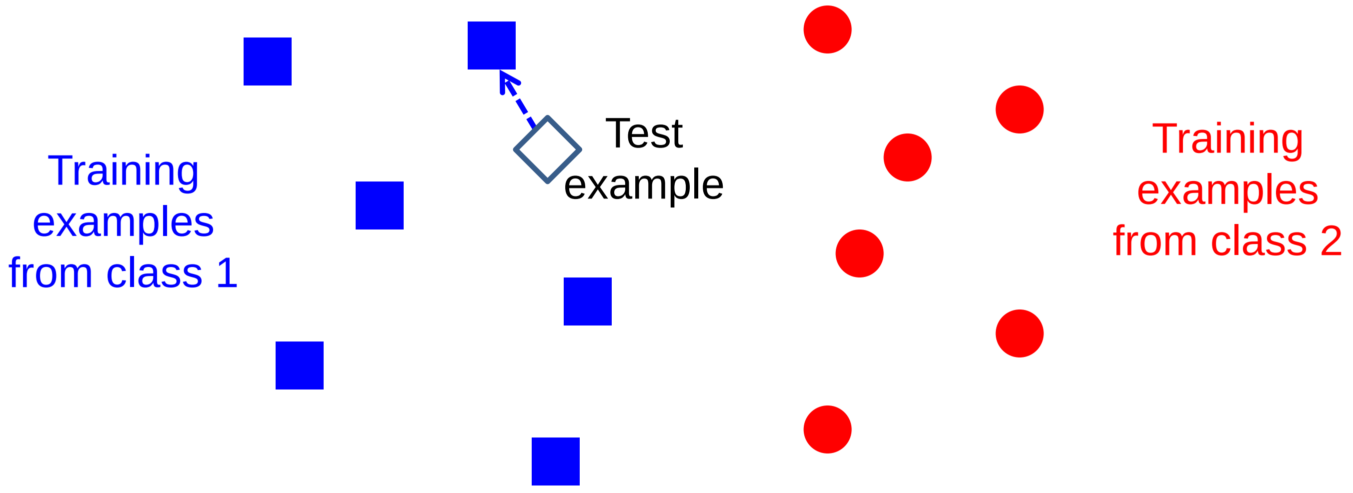
Unsupervised Learning

- *Unsupervised learning* categories and techniques
  - **Clustering**
    - *k*-means clustering
    - Mean-shift clustering
    - Spectral clustering
  - **Density estimation**
    - Gaussian mixture model (GMM)
    - Graphical models
  - **Dimensionality reduction**
    - Principal component analysis (PCA)
    - Factor analysis

Slide credit: Y-Fan Chang – An Overview of Machine Learning 7

Nearest Neighbor Classifier

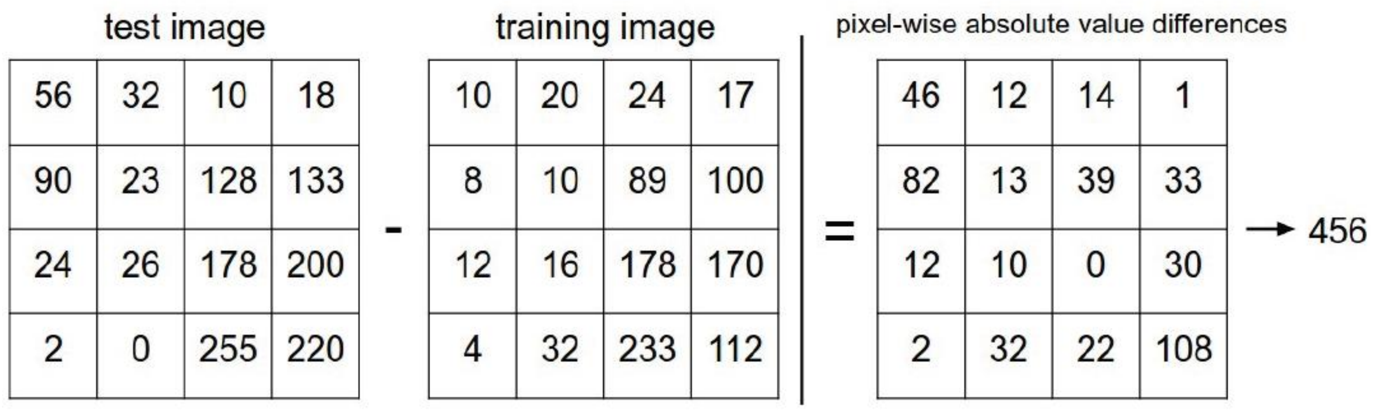
- *Nearest Neighbor* - for each test data point, assign the class label of the nearest training data point
  - Adopt a distance function to find the nearest neighbor
    - Calculate the distance to each data point in the training set, and assign the class of the nearest data point (minimum distance)
  - No parameter training required


Training examples from class 1
Test example
Training examples from class 2

Picture from: James Hays – Machine Learning Overview

Nearest Neighbor Classifier

- For image classification, the distance between all pixels is calculated (e.g., using ℓ_1 norm, or ℓ_2 norm)
  - Accuracy on CIFAR10: 38.6%
- Disadvantages:
  - The classifier must remember all training data and store it for future comparisons with the test data
  - Classifying a test image is expensive since it requires a comparison to all training images



test image

56	32	10	18
90	23	128	133
24	26	178	200
2	0	255	220

-

training image

10	20	24	17
8	10	89	100
12	16	178	170
4	32	233	112

=

pixel-wise absolute value differences

46	12	14	1
82	13	39	33
12	10	0	30
2	32	22	108

→ 456

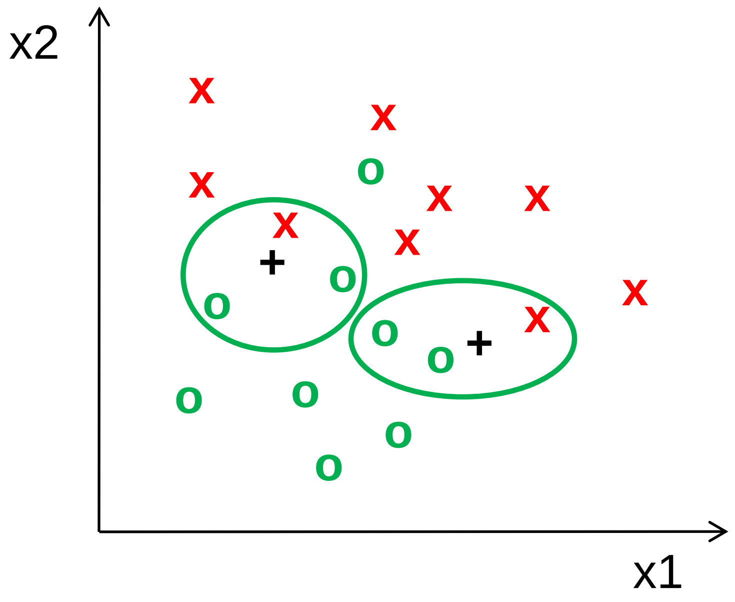
ℓ_1 norm
(Manhattan distance)

$$d_1(I_1, I_2) = \Sigma_p \left|I_1^p - I_2^p\right|$$

Picture from: https://cs231n.github.io/classification/

Nearest Neighbor Classifier

- ***k-Nearest Neighbors*** approach considers multiple neighboring data points to classify a test data point
  - E.g., 3-nearest neighbors
    - The test example in the figure is the + mark
    - The class of the test example is obtained by voting (of 3 closest points)



Picture from: James Hays – Machine Learning Overview

Linear Classifier

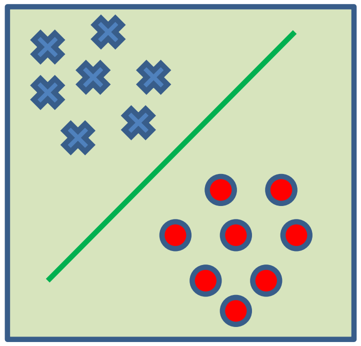
- *Linear classifier*
  - Find a linear function *f* of the inputs x_i that separates the classes

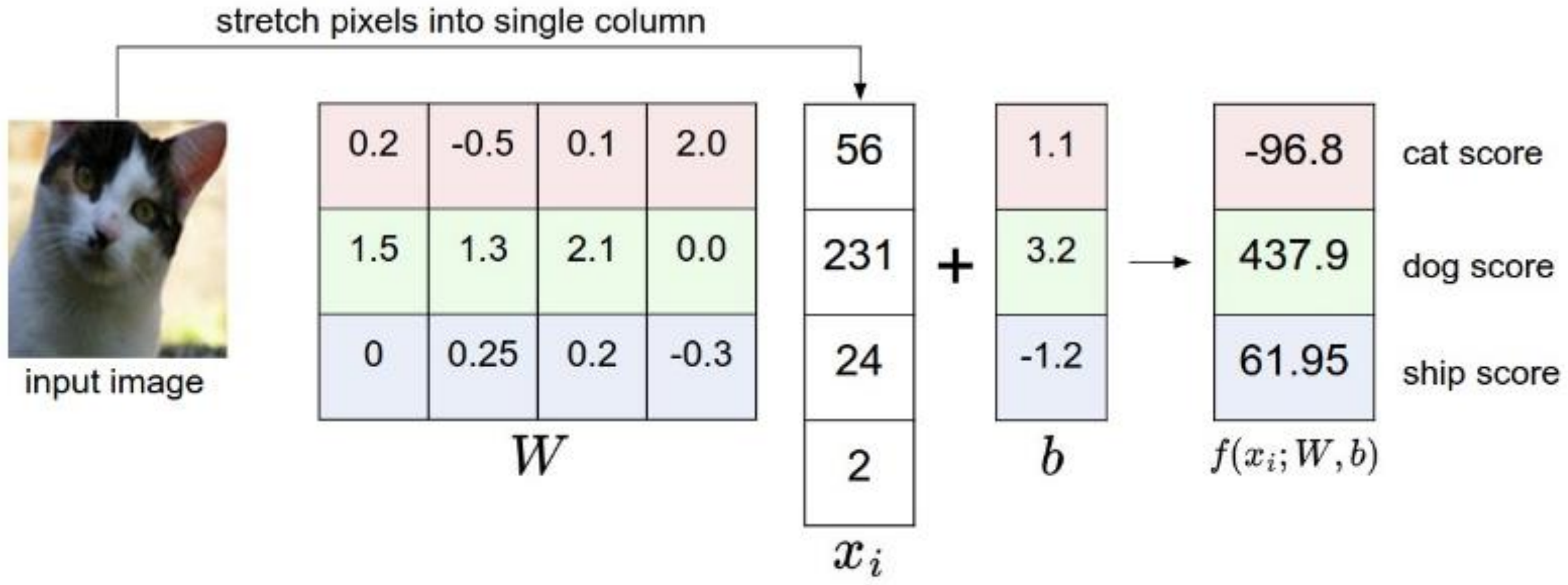
$$f(x_i, W, b) = Wx_i + b$$

  - Use pairs of inputs and labels to find the **weights matrix** *W* and the **bias vector** *b*
    - The weights and biases are the **parameters** of the function *f*
    - Parameter learning is solved by minimizing a loss function
  - **Perceptron** algorithm is often used to find optimal parameters
    - Updates the parameters until a minimal error is reached (similar to gradient descent)
  - Linear classifier is a simple approach, but it is a building block of advanced classification algorithms, such as SVM and neural networks

Linear Classifier

- The decision boundary is linear
  - A straight line in 2D, a flat plane in 3D, a hyperplane in 3D and higher dimensional space
- Example: classify an input image
  - The selected parameters in this example are not good, because the predicted cat score is low





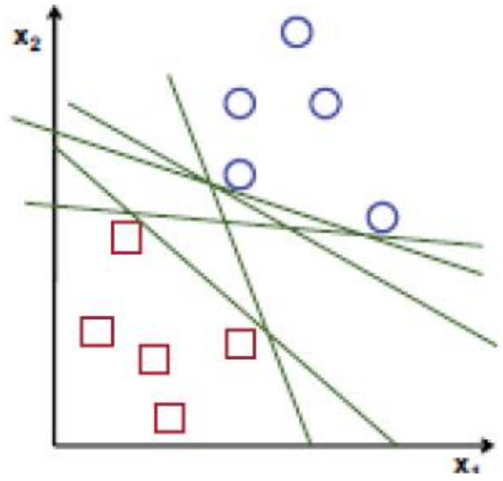
Picture from: https://cs231n.github.io/classification/

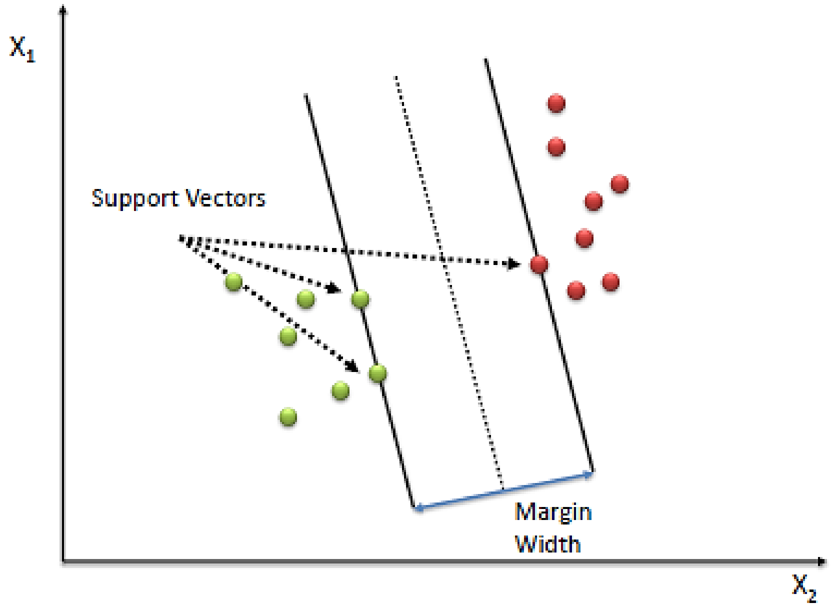
Support Vector Machines

- ***Support vector machines (SVM)***
  - How to find the best decision boundary?
    - All lines in the figure correctly separate the 2 classes
    - The line that is farthest from all training examples will have better generalization capabilities
  - SVM solves an optimization problem:
    - First, identify a decision boundary that correctly classifies all examples
    - Next, increase the geometric margin between the boundary and all examples
  - The data points that define the maximum margin width are called support vectors
  - Find *W* and *b* by solving:

$$\min \frac{1}{2}\|w\|^2$$

$$s.t.\ y_i(w \cdot x_i + b) \geq 1,\ \forall x_i$$





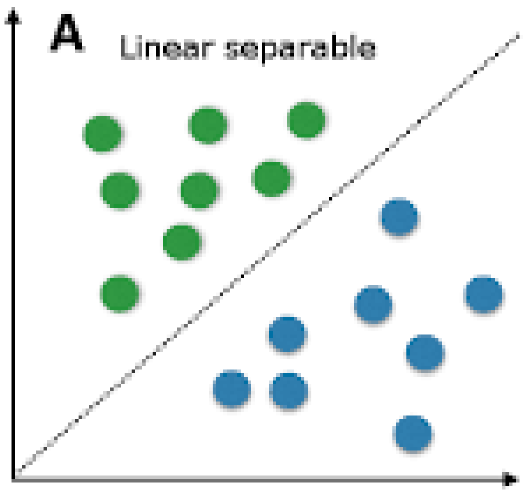
Linear vs Non-linear Techniques

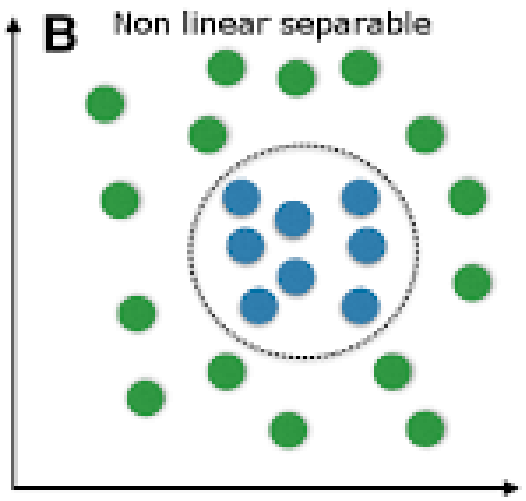
- Linear classification techniques
  - Linear classifier
  - Perceptron
  - Logistic regression
  - Linear SVM
  - Naïve Bayes
- Non-linear classification techniques
  - *k*-nearest neighbors
  - Non-linear SVM
  - Neural networks
  - Decision trees
  - Random forest

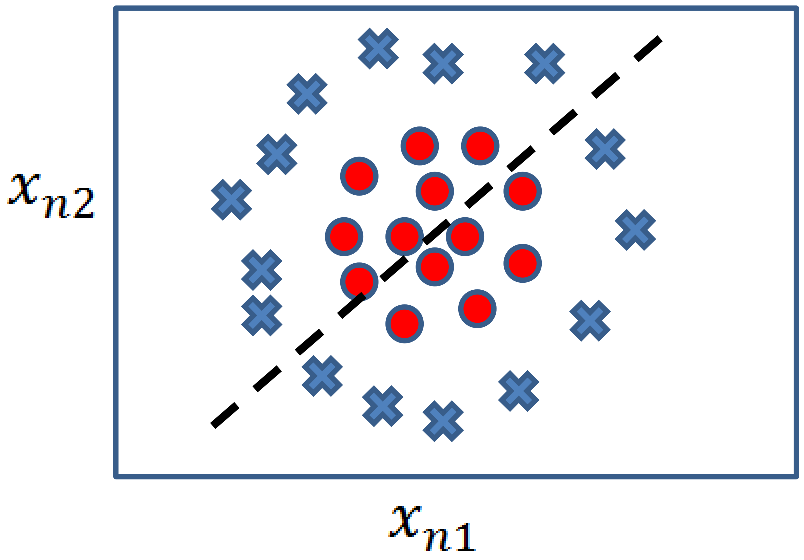
Linear vs Non-linear Techniques

- For some tasks, input data can be linearly separable, and linear classifiers can be suitably applied

- For other tasks, linear classifiers may have difficulties to produce adequate decision boundaries



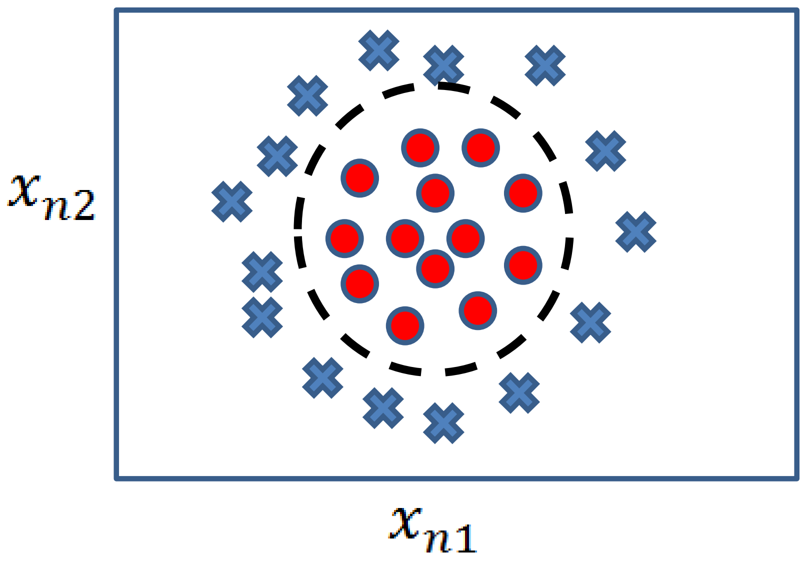




Picture from: Y-Fan Chang – An Overview of Machine Learning

Non-linear Techniques

- Non-linear classification
  - Features z_i are obtained as non-linear functions of the inputs x_i
  - It results in non-linear decision boundaries
  - Can deal with non-linearly separable data



Inputs: $x_i = [x_{n1} \quad x_{n2}]$

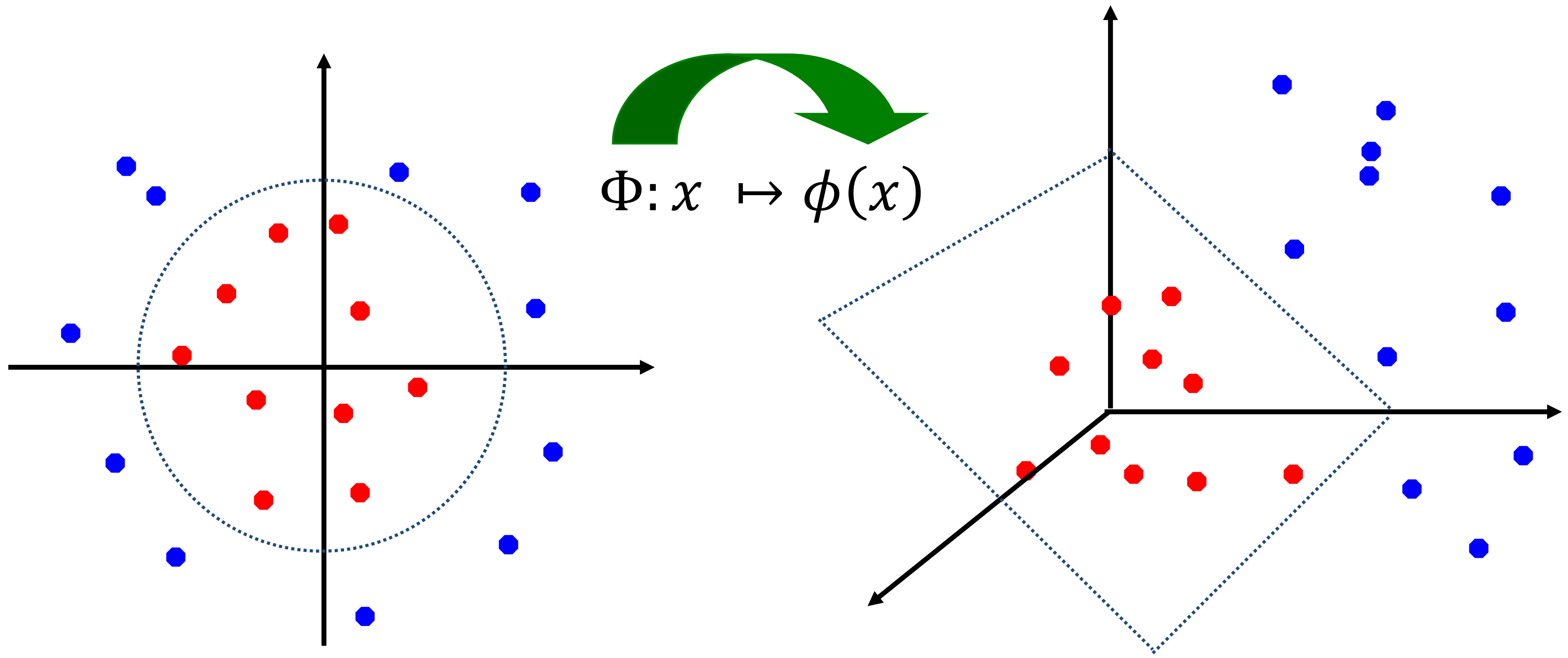
Features: $z_i = [x_{n1} \quad x_{n2} \quad x_{n1} \cdot x_{n2} \quad x_{n1}^2 \quad x_{n2}^2]$

Outputs: $f(x_i, W, b) = W z_i + b$

Picture from: Y-Fan Chang – An Overview of Machine Learning

Non-linear Support Vector Machines

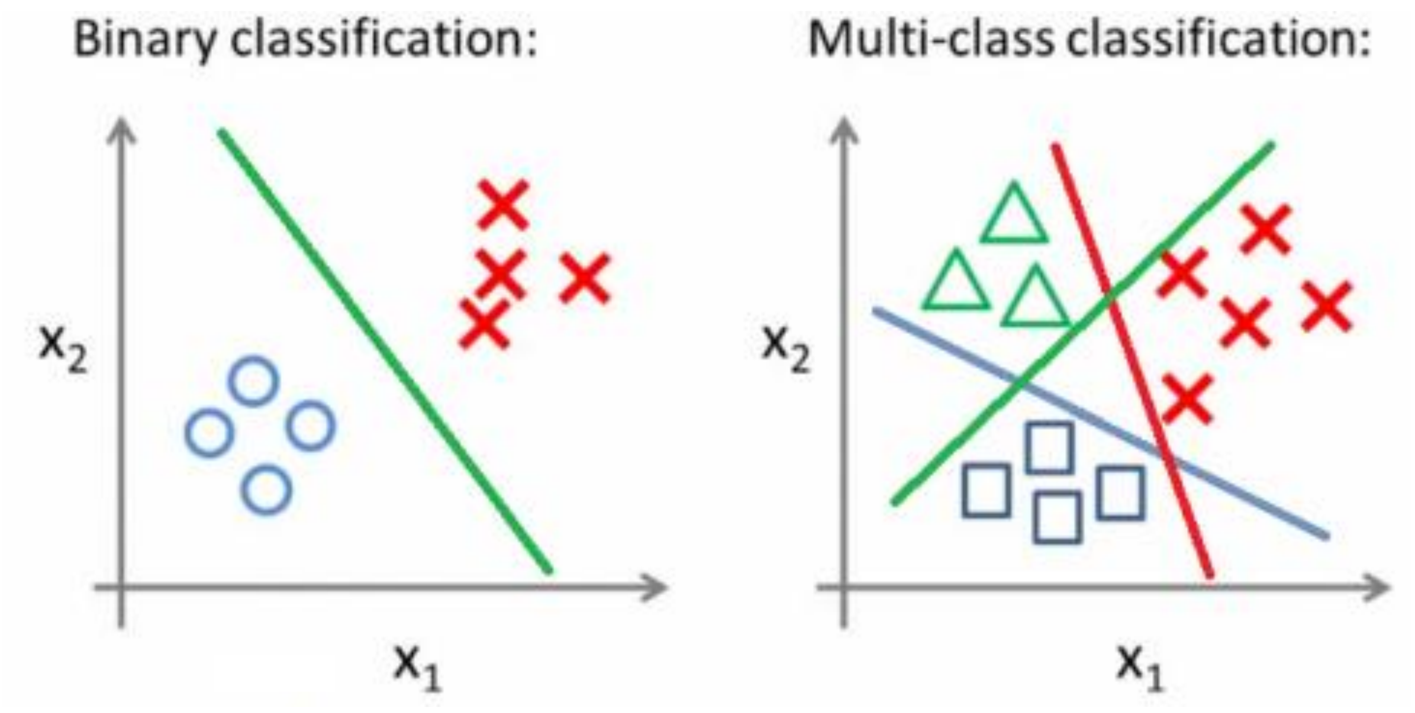
- *Non-linear SVM*
  - The original input space is mapped to a higher-dimensional feature space where the training set is linearly separable
  - Define a non-linear kernel function to calculate a non-linear decision boundary in the original feature space



Picture from: James Hays – Machine Learning Overview

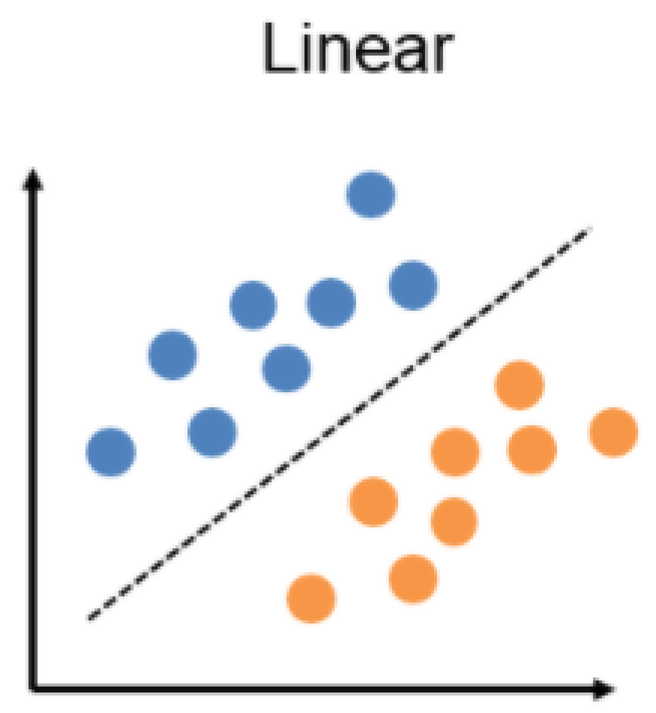
Binary vs Multi-class Classification

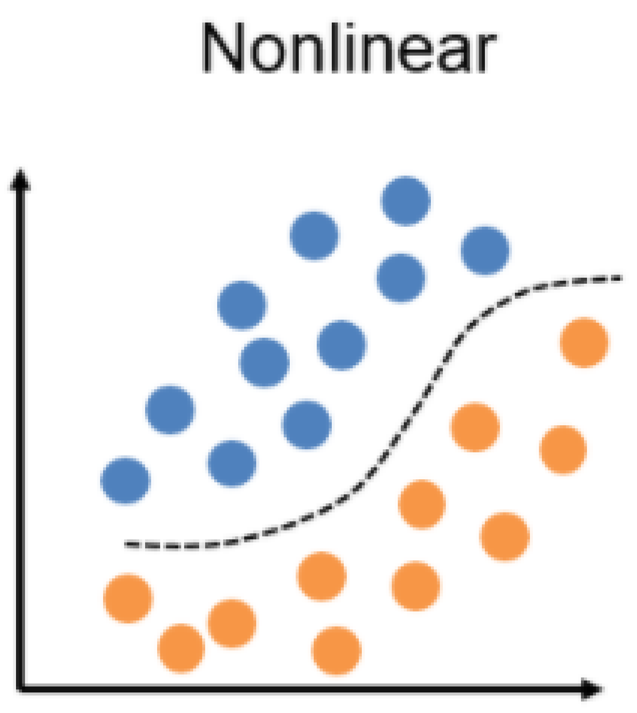
- A problem with only 2 classes is referred to as ***binary classification***
  - The output labels are 0 or 1
  - E.g., benign or malignant tumor, spam or no spam email
- A problem with 3 or more classes is referred to as ***multi-class classification***



Binary vs Multi-class Classification

- Both the binary and multi-class classification problems can be linearly or non-linearly separated
  - Linearly and non-linearly separated data for binary classification problem



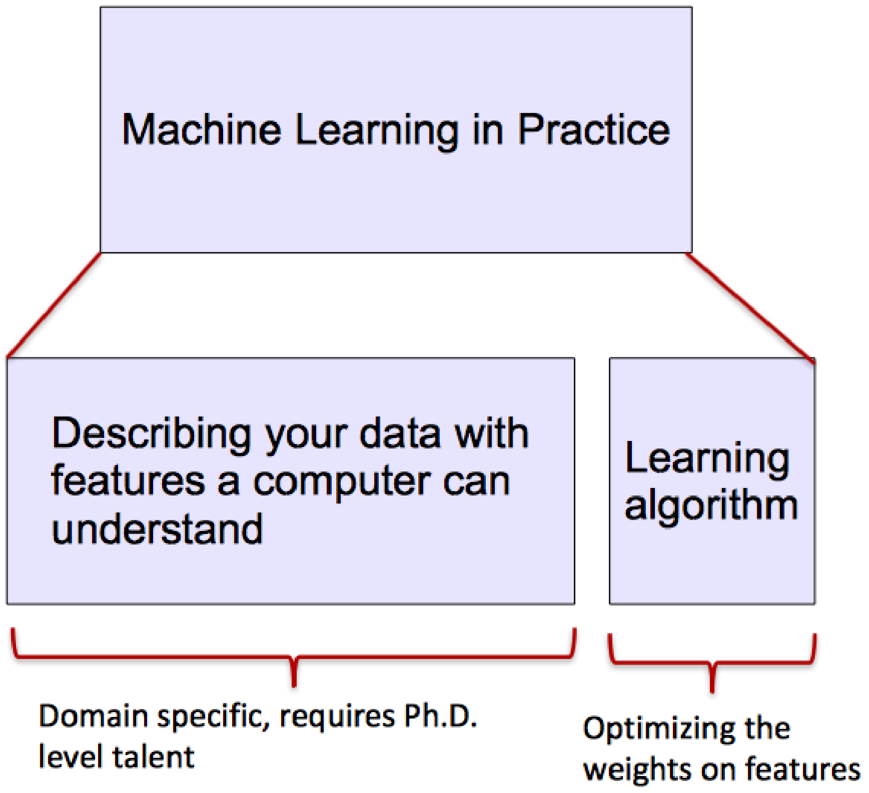


No-Free-Lunch Theorem

- Wolpert (2002) - The Supervised Learning No-Free-Lunch Theorems
- The derived classification models for supervised learning are simplifications of the reality
  - The simplifications are based on certain assumptions
  - The assumptions fail in some situations
    - E.g., due to inability to perfectly estimate parameters from limited data
- In summary, the theorem states:
  - **No single classifier works the best for all possible situations**
  - Since we need to make assumptions to generalize

ML vs. Deep Learning

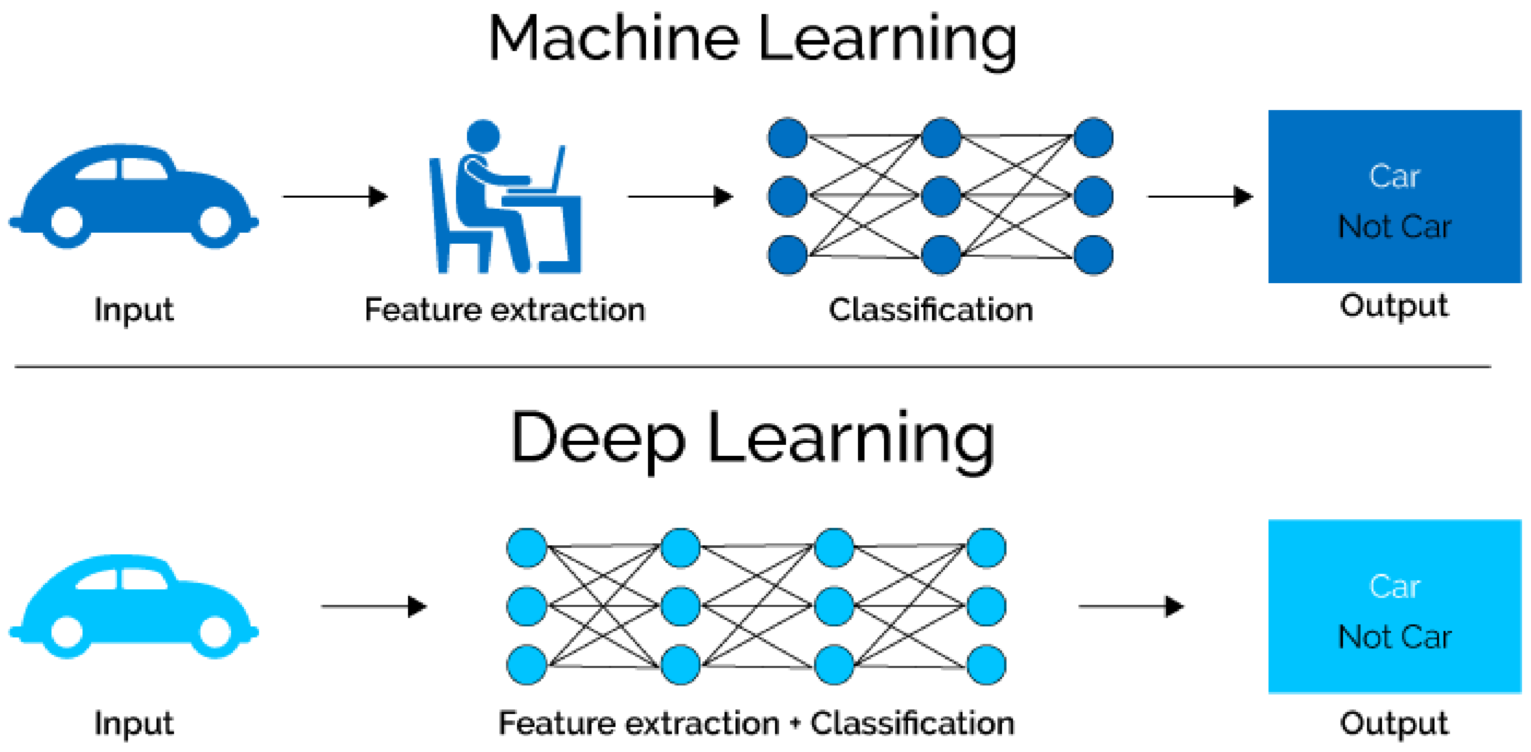
- Most machine learning methods rely on **human-designed feature representations**
  - ML becomes just optimizing weights to best make a final prediction


Machine Learning in Practice
Describing your data with features a computer can understand
Learning algorithm
Domain specific, requires Ph.D. level talent
Optimizing the weights on features

Picture from: Ismini Lourentzou – Introduction to Deep Learning

ML vs. Deep Learning

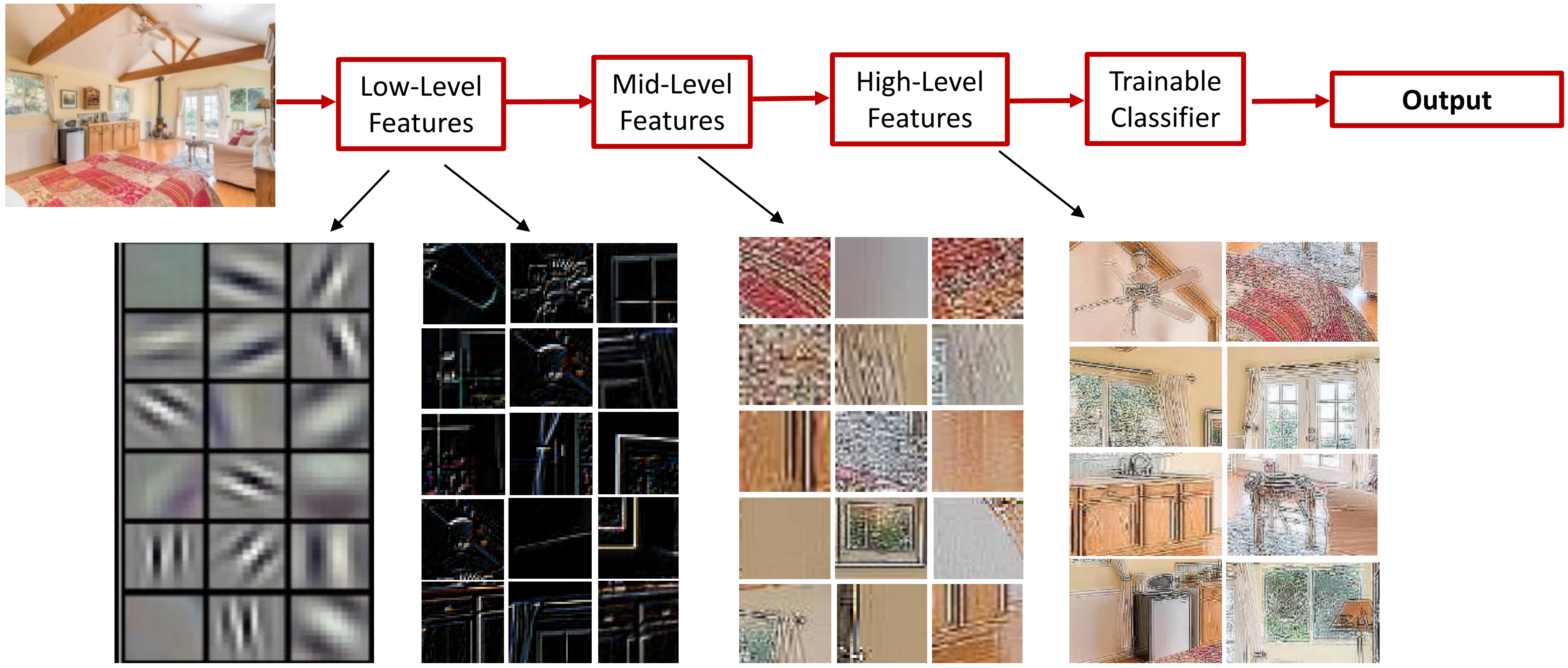
- *Deep learning* (DL) is a machine learning subfield that uses multiple layers for learning data representations
  - DL is exceptionally effective at learning patterns



Picture from: https://www.xenonstack.com/blog/static/public/uploads/media/machine-learning-vs-deep-learning.png

ML vs. Deep Learning

- DL applies a multi-layer process for learning rich hierarchical features (i.e., data representations)
  - Input image pixels → Edges → Textures → Parts → Objects



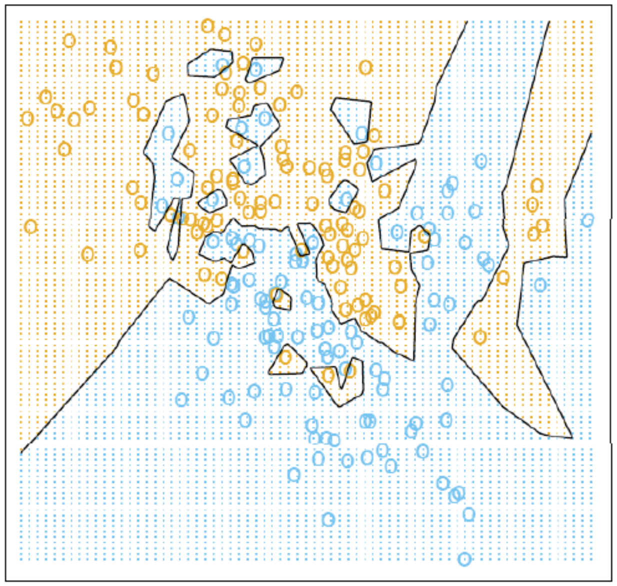
Slide credit: Param Vir Singh – Deep Learning

Why is DL Useful?

- DL provides a flexible, learnable framework for representing visual, text, linguistic information
  - Can learn in unsupervised and supervised manner
- DL represents an effective end-to-end system learning
- Requires large amounts of training data
- Since about 2010, DL has outperformed other ML techniques
  - First in vision and speech, then NLP, and other applications

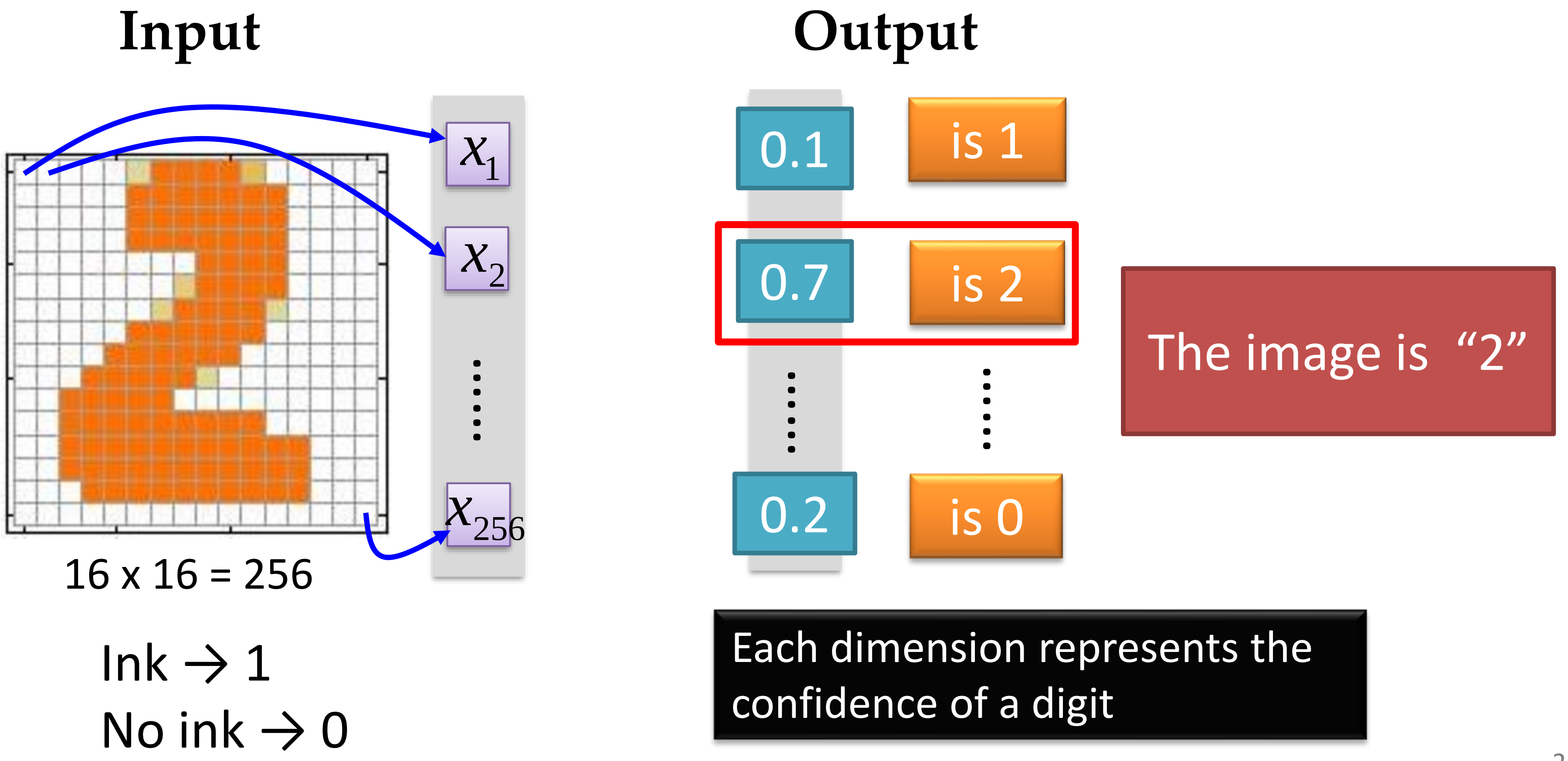
Representational Power

- NNs with at least one hidden layer are *universal approximators*
  - Given any continuous function $h(x)$ and some $\epsilon > 0$, there exists a NN with one hidden layer (and with a reasonable choice of non-linearity) described with the function $f(x)$, such that $\forall x, |h(x) - f(x)| < \epsilon$
  - I.e., NN can approximate any arbitrary complex continuous function
- NNs use nonlinear mapping of the inputs to the outputs $f(x)$ to compute complex decision boundaries
- But then, why use deeper NNs?
  - The fact that deep NNs work better is an empirical observation
  - Mathematically, deep NNs have the same representational power as a one-layer NN



Introduction to Neural Networks

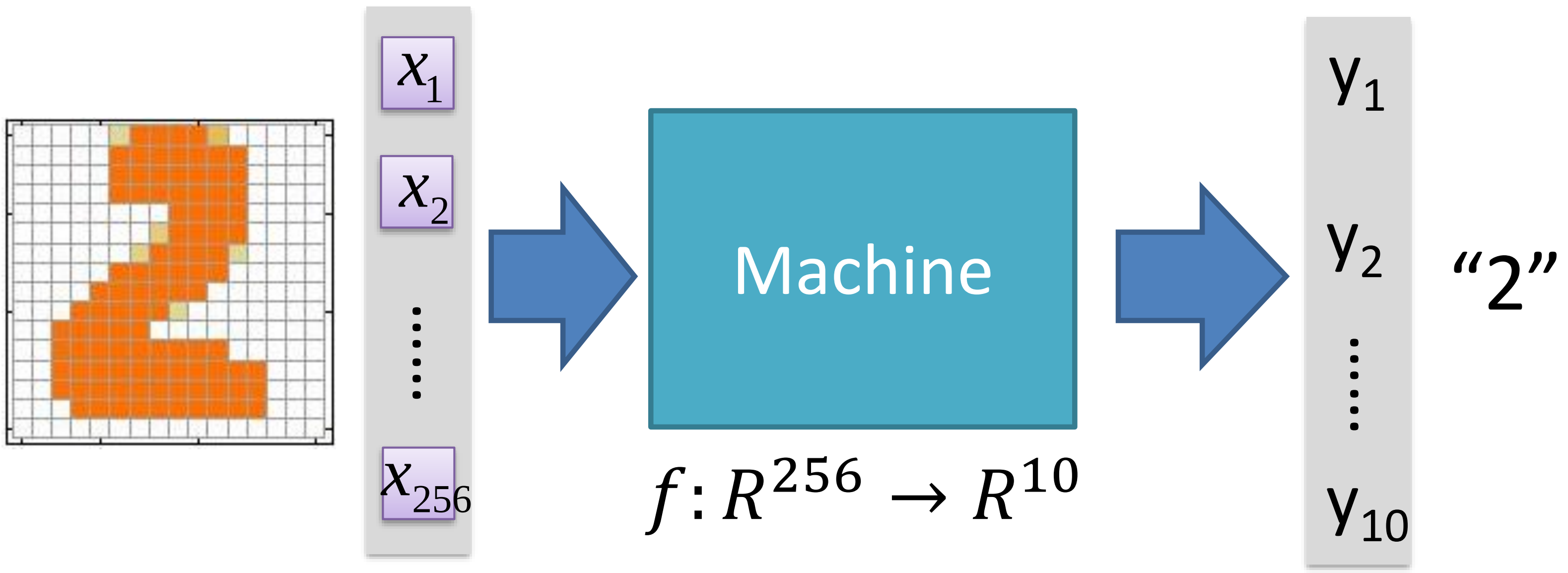
- Handwritten digit recognition (MNIST dataset)
  - The intensity of each pixel is considered an input element
  - The output is the class of the digit



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Introduction to Neural Networks

- Handwritten digit recognition



$f: R^{256} \rightarrow R^{10}$

The function f is represented by a neural network

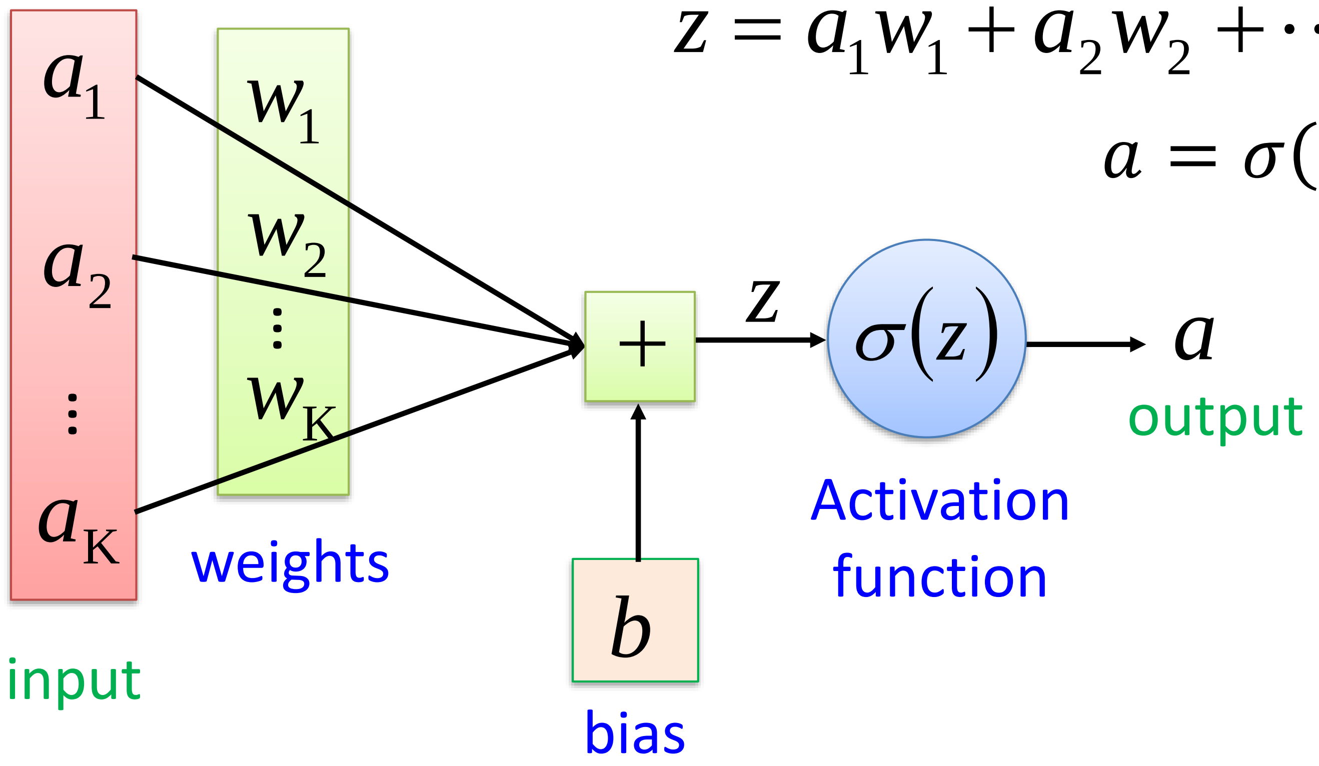
Slide credit: Hung-yi Lee – Deep Learning Tutorial

Elements of Neural Networks

- NNs consist of hidden layers with neurons (i.e., computational units)
- A single neuron maps a set of inputs into an output number, or $f: R^K \to R$

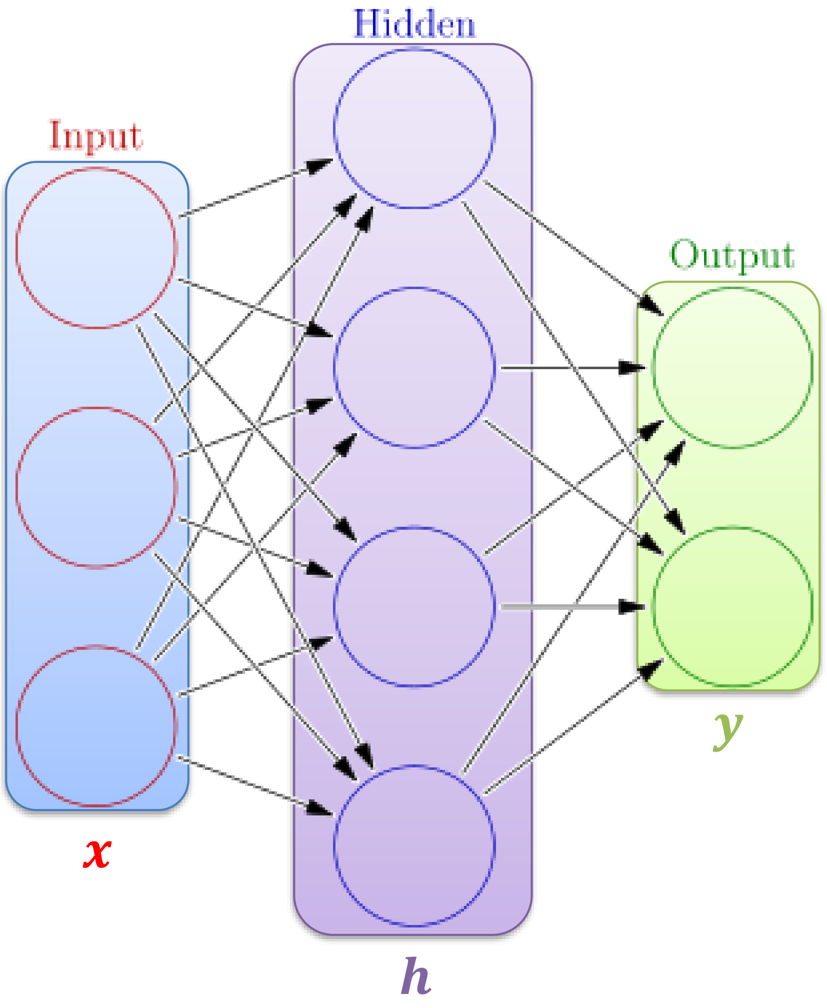
$$z = a_1 w_1 + a_2 w_2 + \cdots + a_K w_K + b$$

$$a = \sigma(z)$$



Elements of Neural Networks

- A NN with one hidden layer and one output layer



Weights   Biases

$$\textit{hidden layer } h = \sigma(W_1 x + b_1)$$

$$\textit{output layer } y = \sigma(W_2 h + b_2)$$

Activation functions

4 + 2 = 6 neurons (not counting inputs)

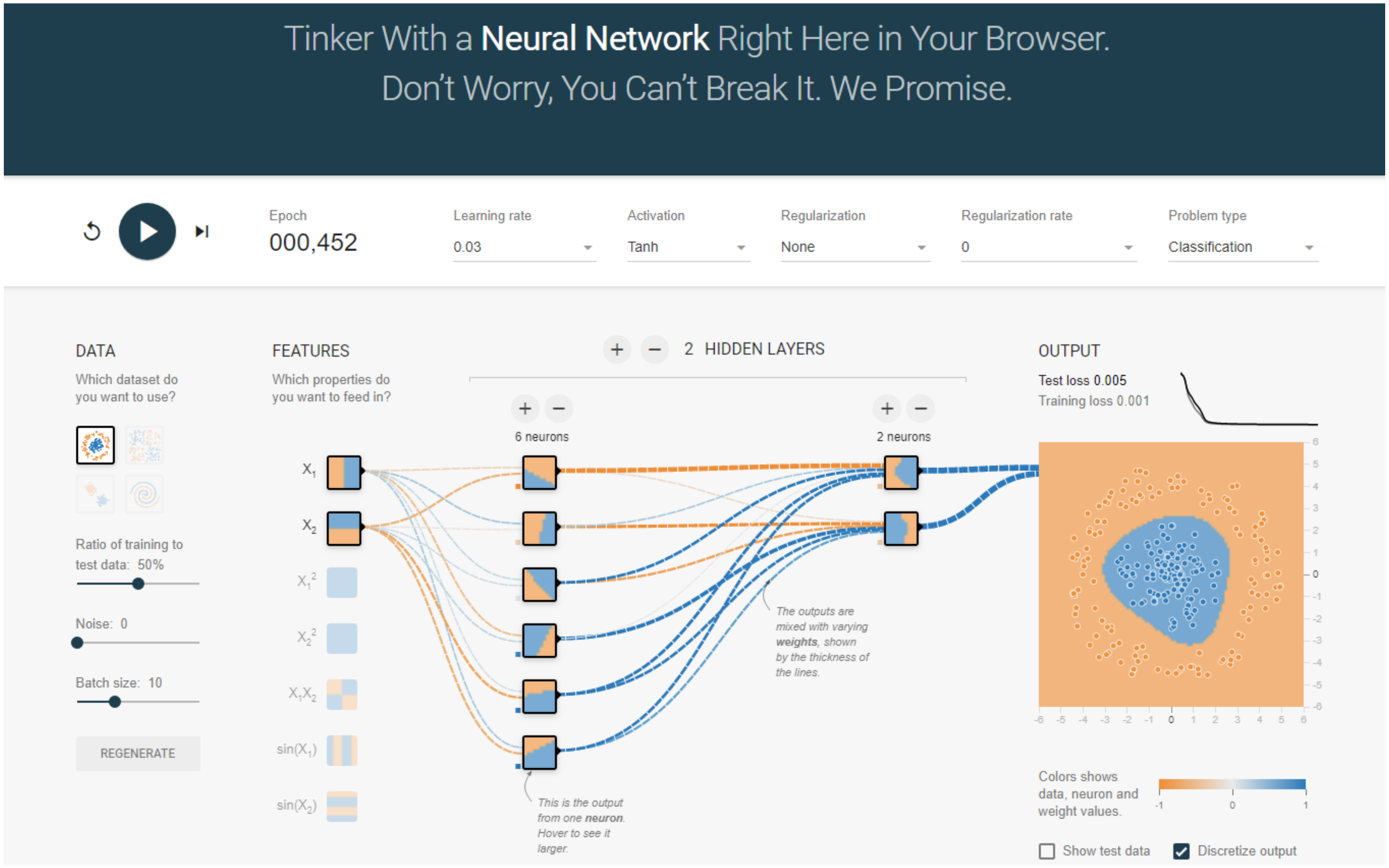
[3 × 4] + [4 × 2] = 20 weights

4 + 2 = 6 biases

26 learnable parameters

Elements of Neural Networks

- A neural network playground link


Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.

Epoch 000,452
Learning rate 0.03
Activation Tanh
Regularization None
Regularization rate 0
Problem type Classification

DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10

FEATURES
Which properties do you want to feed in?

2 HIDDEN LAYERS
6 neurons
2 neurons

*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

*This is the output from one **neuron**. Hover to see it larger.*

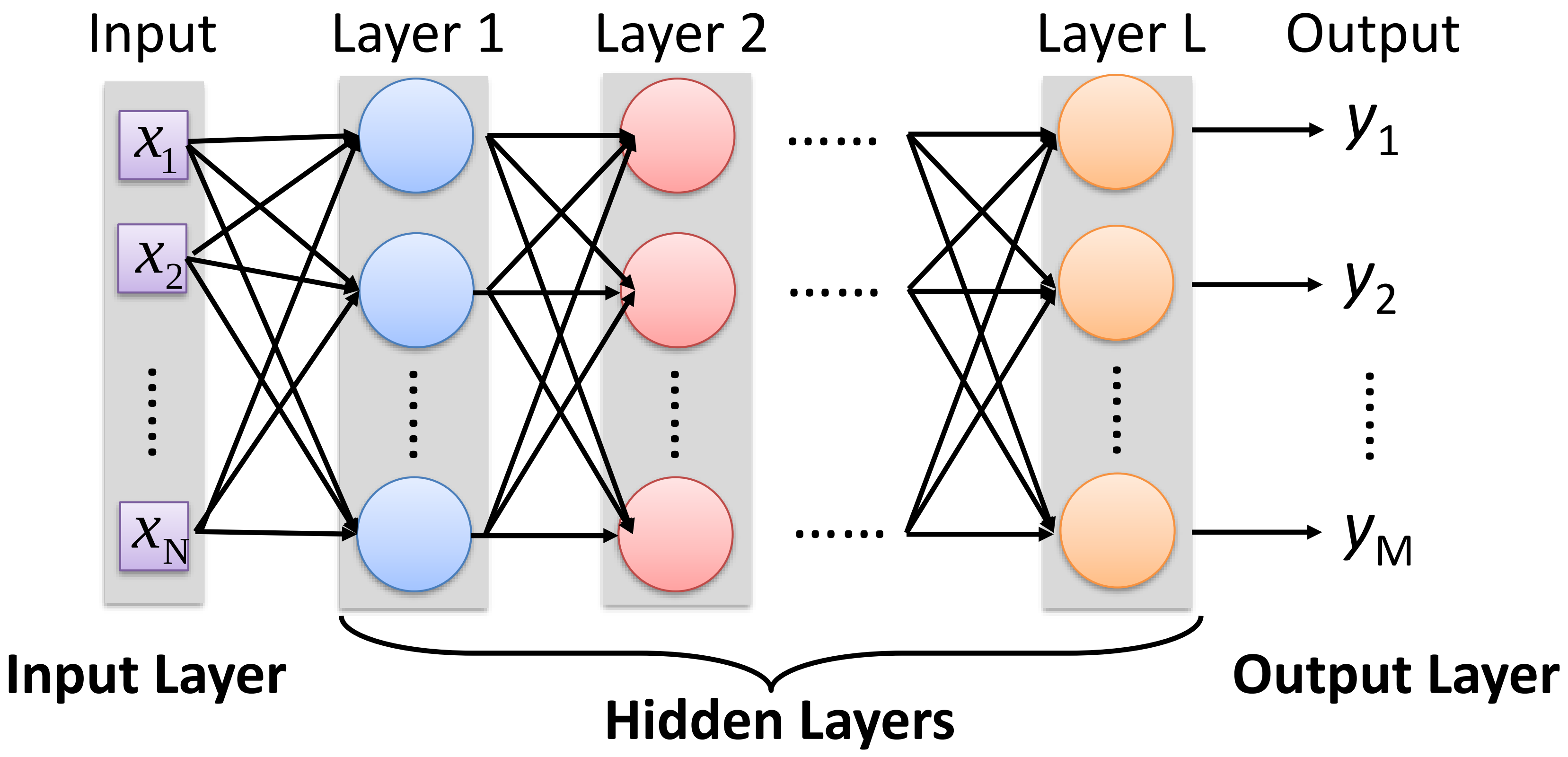
OUTPUT
Test loss 0.005
Training loss 0.001

Colors shows data, neuron and weight values.

Show test data
Discretize output

Elements of Neural Networks

- Deep NNs have many hidden layers
  - Fully-connected (dense) layers (a.k.a. Multi-Layer Perceptron or MLP)
  - Each neuron is connected to all neurons in the succeeding layer

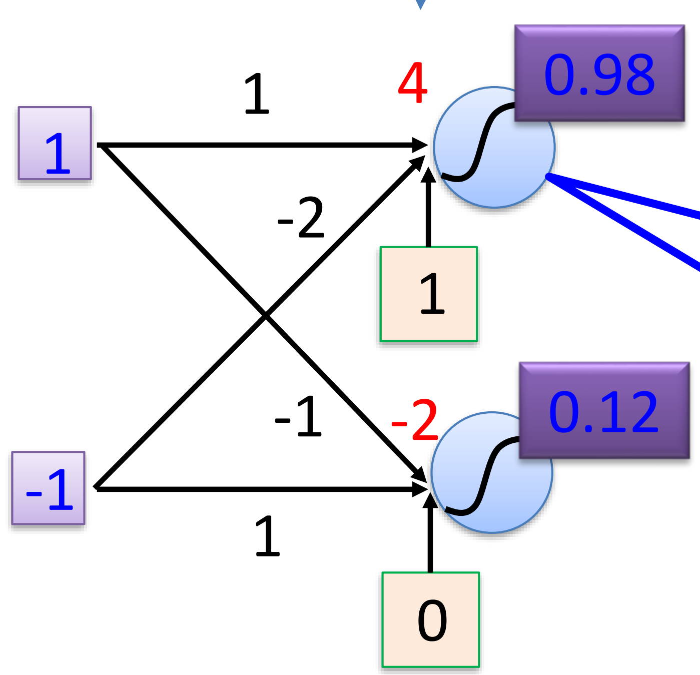


Slide credit: Hung-yi Lee – Deep Learning Tutorial

Elements of Neural Networks

- A simple network, toy example

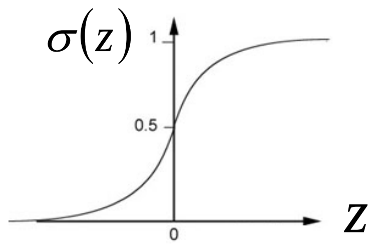
$\boxed{(1 \cdot 1) + (-1) \cdot (-2) + 1 = 4}$



$\boxed{1 \cdot\ (-1) + (-1) \cdot 1 + 0 = -2}$

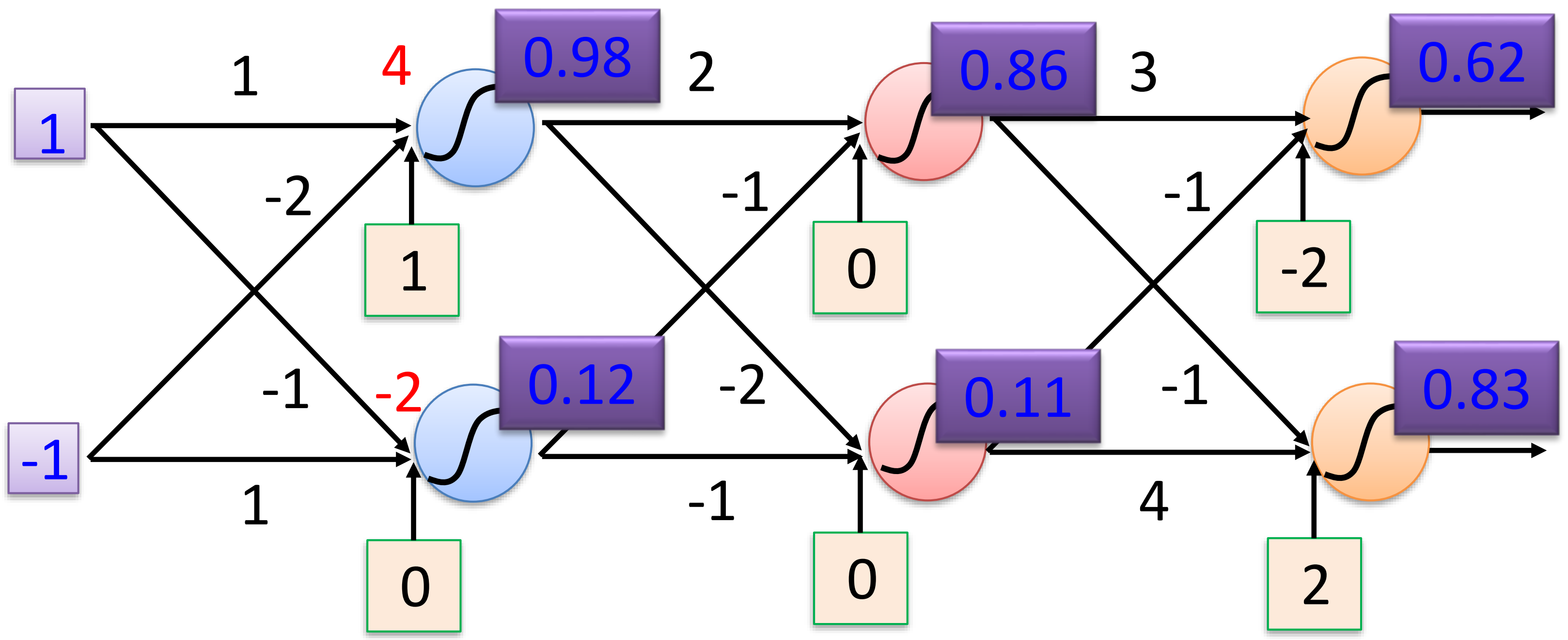
Sigmoid Function

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



Elements of Neural Networks

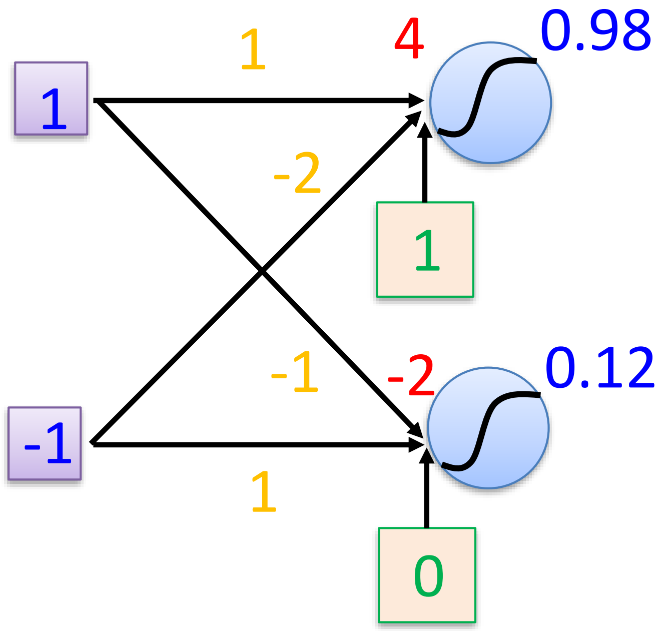
- A simple network, toy example (cont'd)
  - For an input vector $[1 \quad -1]^T$, the output is $[0.62 \quad 0.83]^T$



$$f: R^2 \to R^2 \qquad f\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix}\right) = \begin{bmatrix} 0.62 \\ 0.83 \end{bmatrix}$$

Matrix Operation

- Matrix operations are helpful when working with multidimensional inputs and outputs

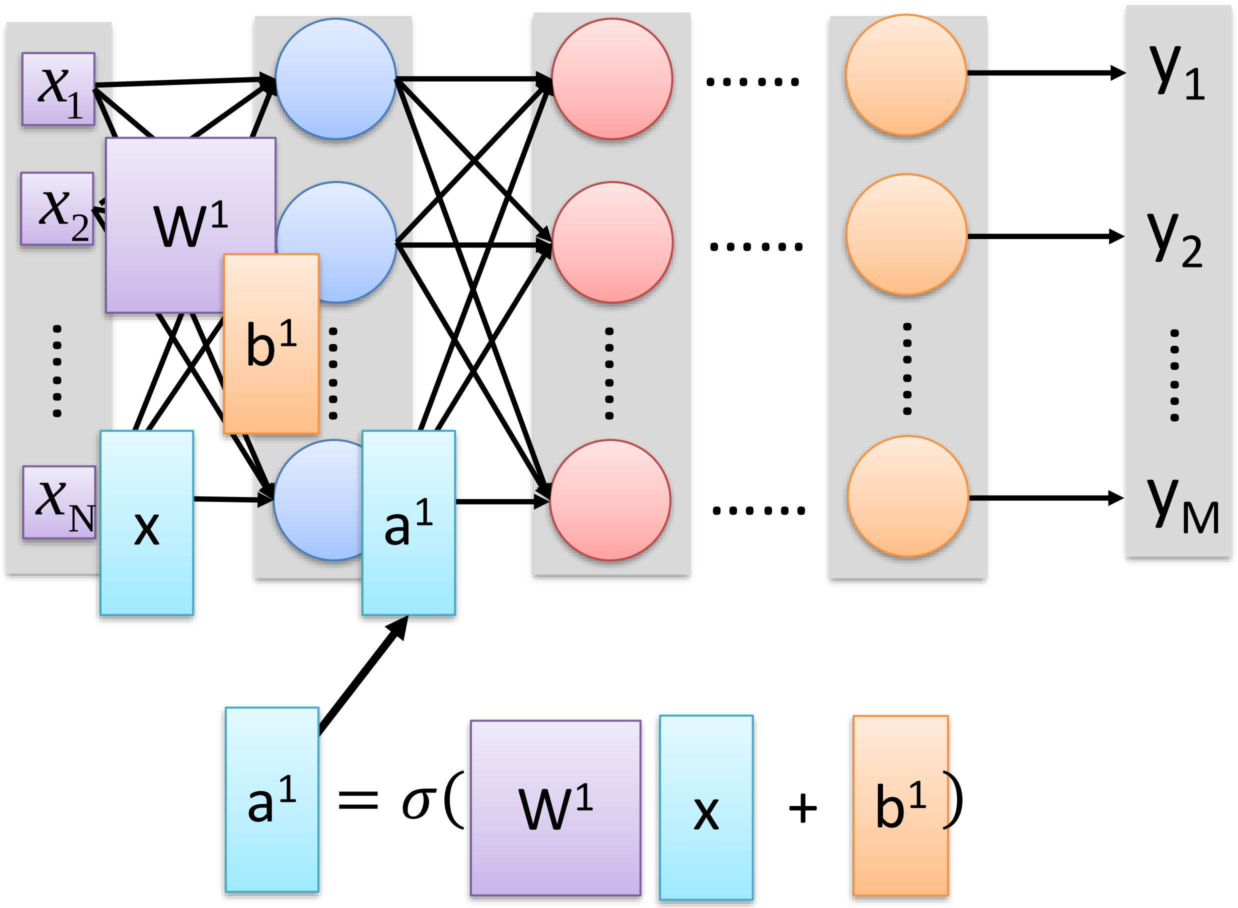


$$\sigma(\underbrace{\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{\begin{bmatrix} 4 \\ -2 \end{bmatrix}}) = \begin{bmatrix} 0.98 \\ 0.12 \end{bmatrix}$$

Slide credit: Hung-yi Lee – Deep Learning Tutorial

Matrix Operation

- Multilayer NN, matrix calculations for the first layer
  - Input vector x, weights matrix W^1, bias vector b^1, output vector a^1

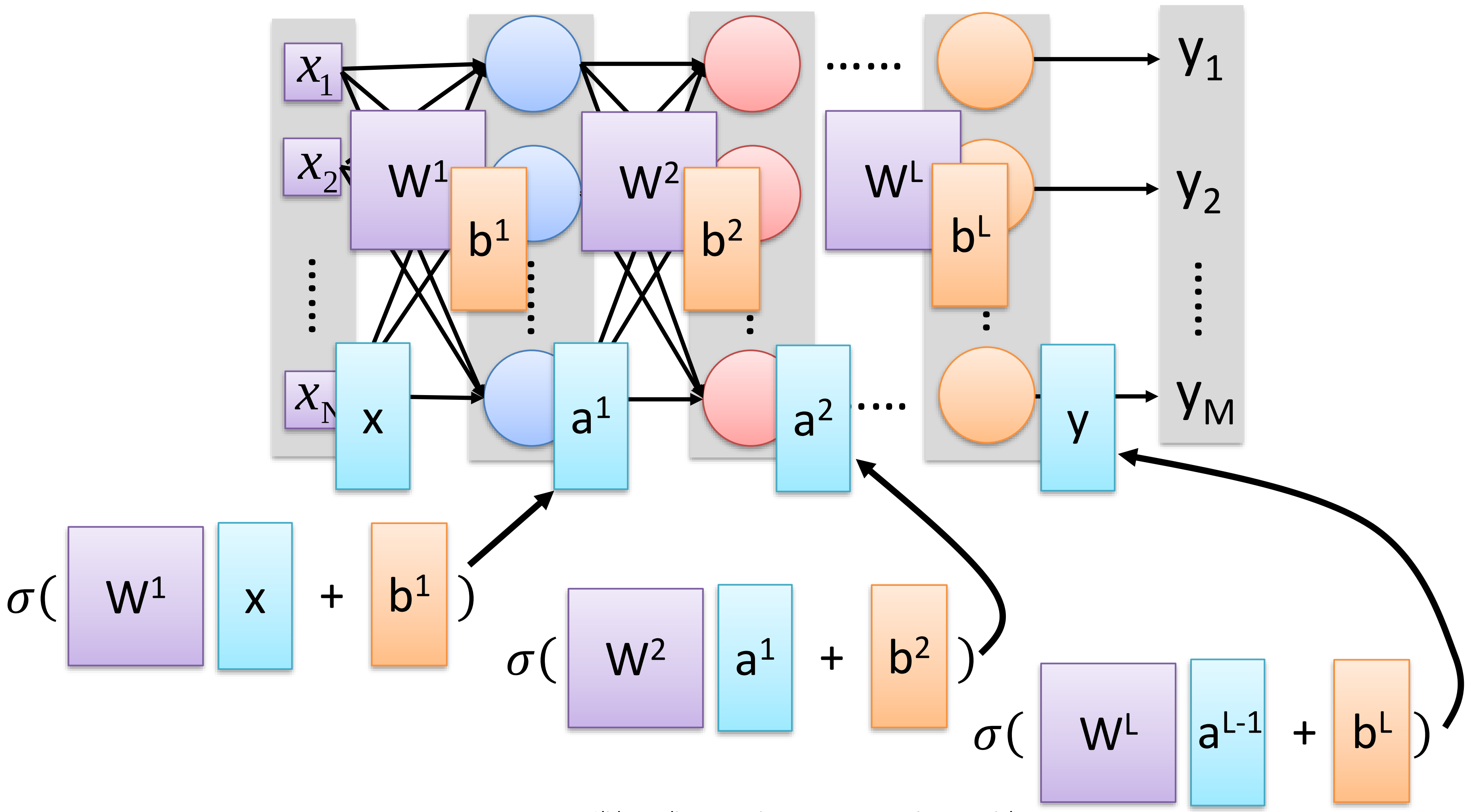


$$a^1 = \sigma(W^1 x + b^1)$$

Slide credit: Hung-yi Lee – Deep Learning Tutorial

Matrix Operation

- Multilayer NN, matrix calculations for all layers

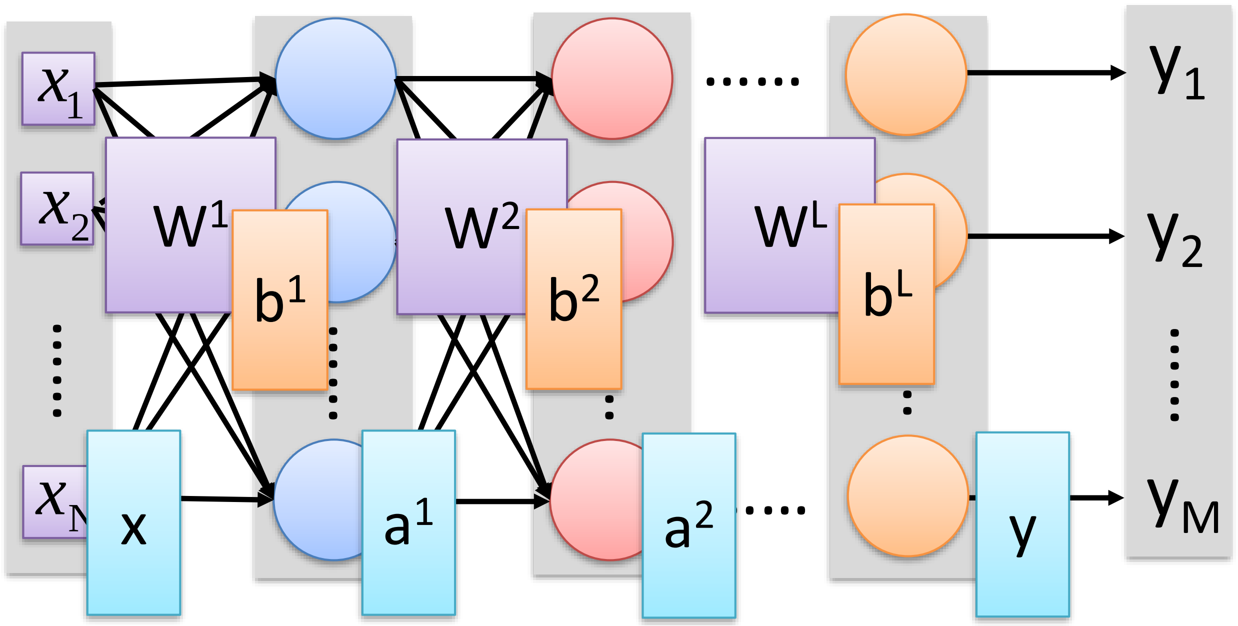


x_1
x_2
x_N
x
W^1
b^1
a^1
W^2
b^2
a^2
W^L
b^L
y
y_1
y_2
y_M
$\sigma(W^1 \; x + b^1)$
$\sigma(W^2 \; a^1 + b^2)$
$\sigma(W^L \; a^{L-1} + b^L)$

Slide credit: Hung-yi Lee – Deep Learning Tutorial

Matrix Operation

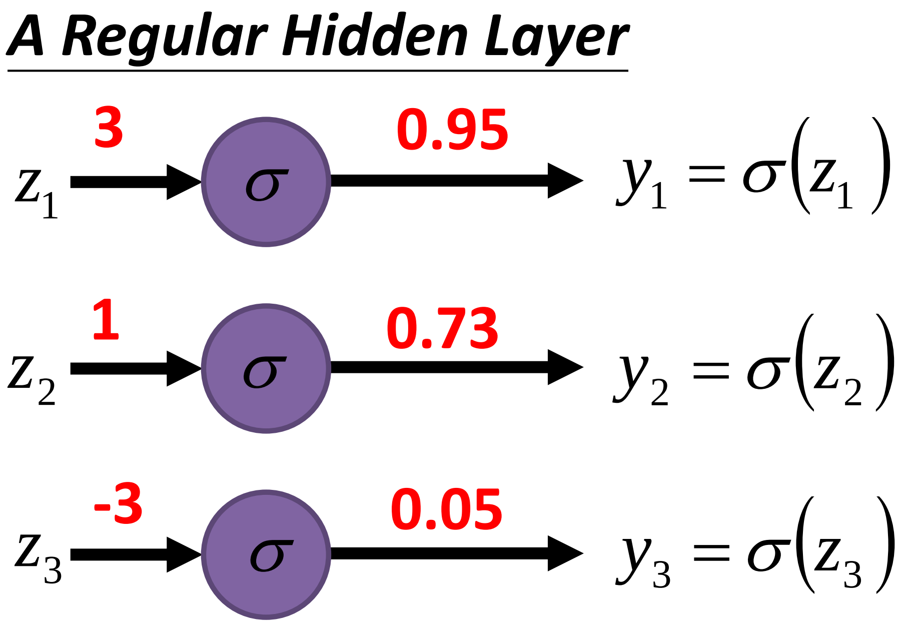
- Multilayer NN, function f maps inputs x to outputs y, i.e., $y = f(x)$



$$\boxed{y} = f\left(\boxed{x}\right) = \sigma\left(\boxed{W^L} \cdots \sigma\left(\boxed{W^2}\,\sigma\left(\boxed{W^1}\,\boxed{x} + \boxed{b^1}\right) + \boxed{b^2}\right) \cdots + \boxed{b^L}\right)$$

Softmax Layer

- In multi-class classification tasks, the output layer is typically a ***softmax layer***
  - If a regular hidden layer with sigmoid activations is used instead, the output of the NN may not be easy to interpret
    - Sigmoid activations can still be used for binary classification



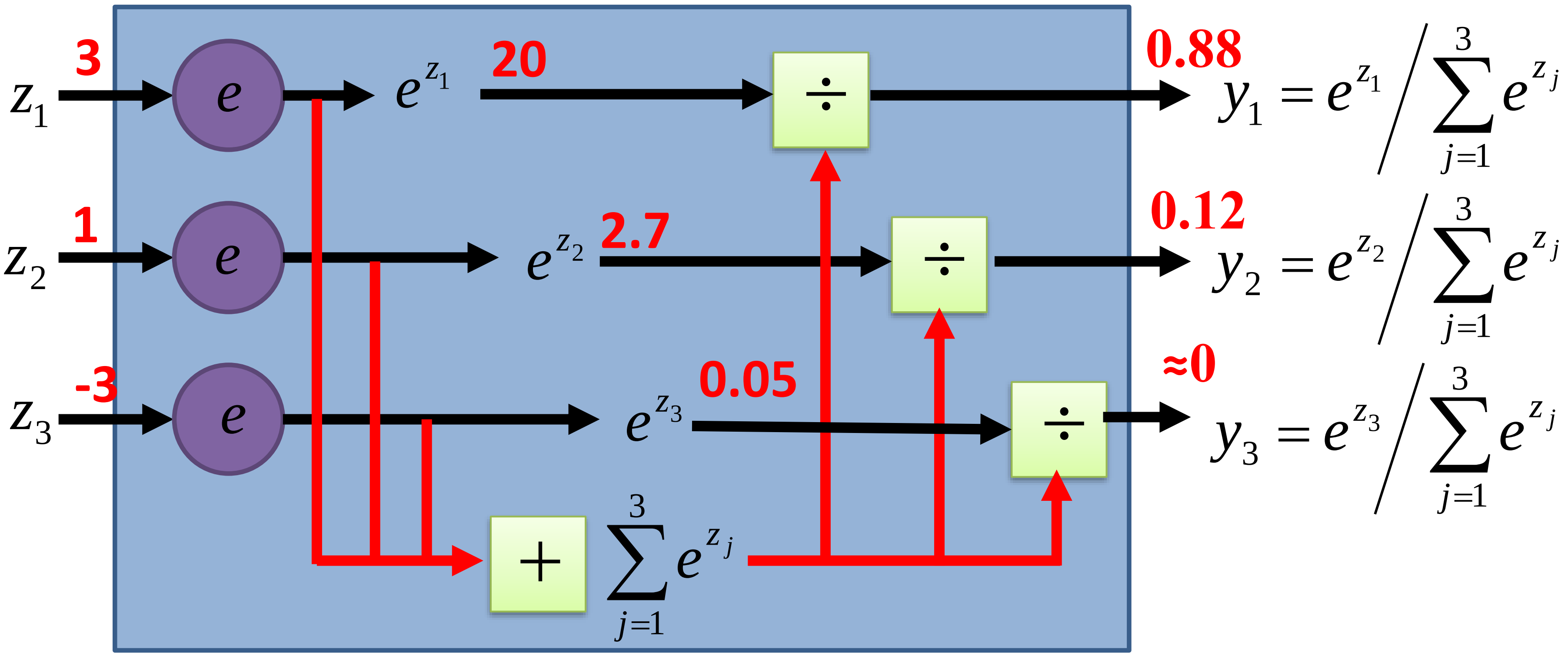
Softmax Layer

- The softmax layer applies softmax activations to output a probability value in the range [0, 1]
  - The values z inputted to the softmax layer are referred to as *logits*

Probability:

- $0 < y_i < 1$
- $\sum_i y_i = 1$

A Softmax Layer

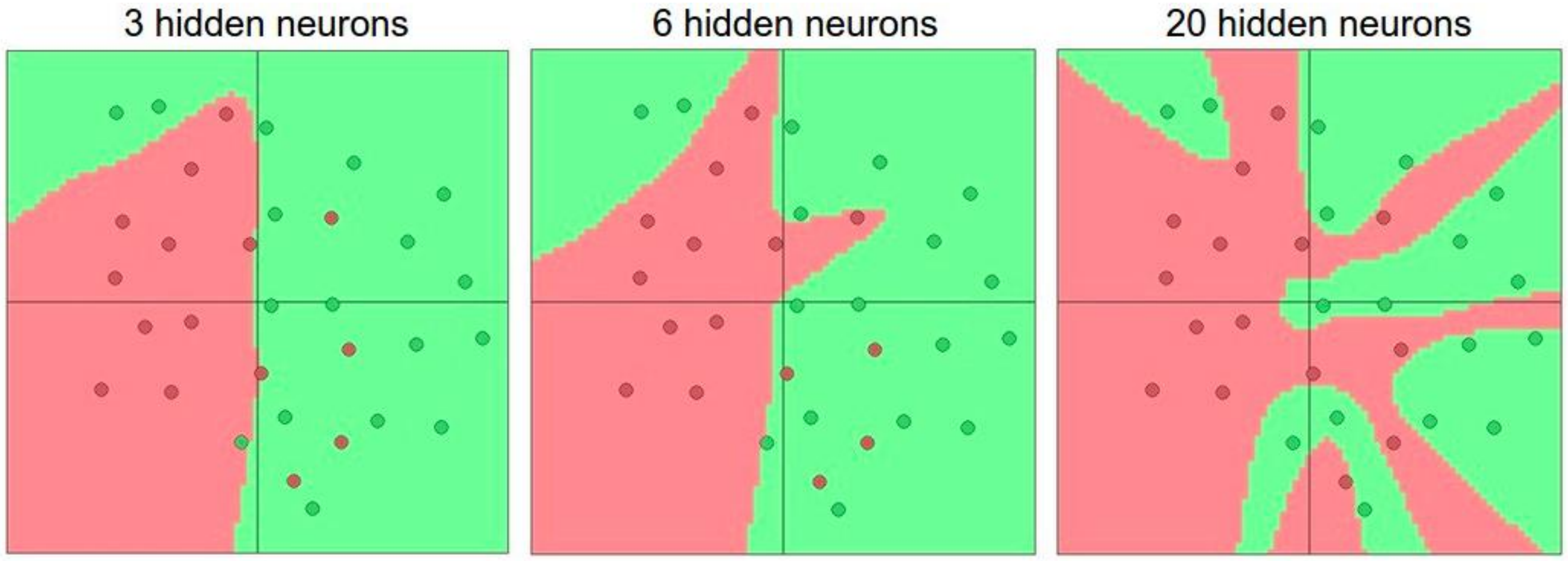


z_1
3
e
e^{z_1}
20
$\div$
0.88
$y_1 = e^{z_1} \Big/ \sum_{j=1}^{3} e^{z_j}$
z_2
1
e
e^{z_2}
2.7
$\div$
0.12
$y_2 = e^{z_2} \Big/ \sum_{j=1}^{3} e^{z_j}$
z_3
-3
e
e^{z_3}
0.05
$\div$
≈0
$y_3 = e^{z_3} \Big/ \sum_{j=1}^{3} e^{z_j}$
$+$
$\sum_{j=1}^{3} e^{z_j}$

Slide credit: Hung-yi Lee – Deep Learning Tutorial

Activation Functions

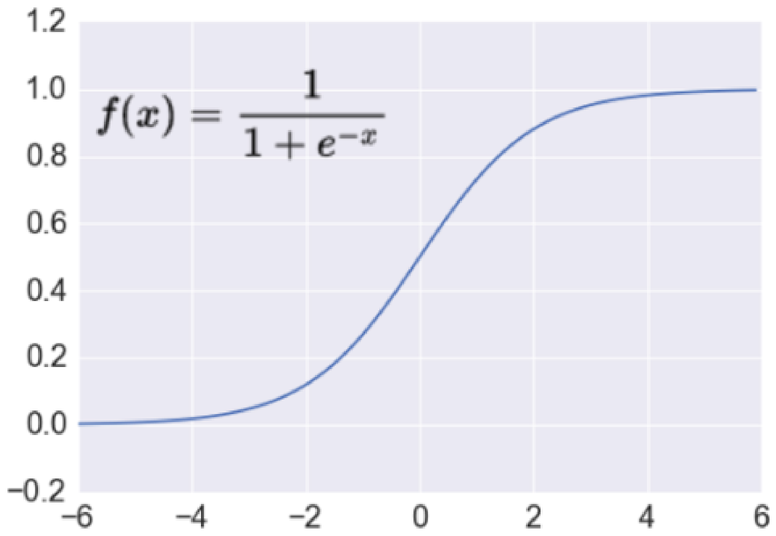
- Non-linear activations are needed to learn complex (non-linear) data representations
  - Otherwise, NNs would be just a linear function (such as $\mathrm{W}_1\mathrm{W}_2 x = Wx$)
  - NNs with large number of layers (and neurons) can approximate more complex functions
    - Figure: more neurons improve representation (but, may overfit)



Picture from: http://cs231n.github.io/assets/nn1/layer_sizes.jpeg

Activation: Sigmoid

- *Sigmoid function* σ: takes a real-valued number and "squashes" it into the range between 0 and 1
  - The output can be interpreted as the firing rate of a biological neuron
    - Not firing = 0; Fully firing = 1
  - When the neuron's activation are 0 or 1, sigmoid neurons saturate
    - Gradients at these regions are almost zero (almost no signal will flow)
  - Sigmoid activations are less common in modern NNs

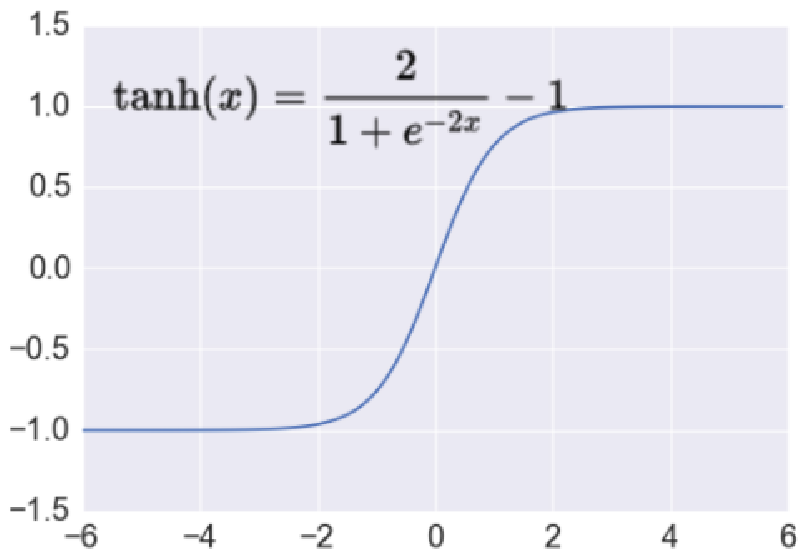


$\mathbb{R}^n \rightarrow [0,1]$

Slide credit: Ismini Lourentzou – Introduction to Deep Learning

Activation: Tanh

- *Tanh function*: takes a real-valued number and "squashes" it into range between -1 and 1
  - Like sigmoid, tanh neurons saturate
  - Unlike sigmoid, the output is zero-centered
    - It is therefore preferred than sigmoid
  - Tanh is a scaled sigmoid: tanh(x)=2σ(2x)−1



$\mathbb{R}^n \to [-1,1]$

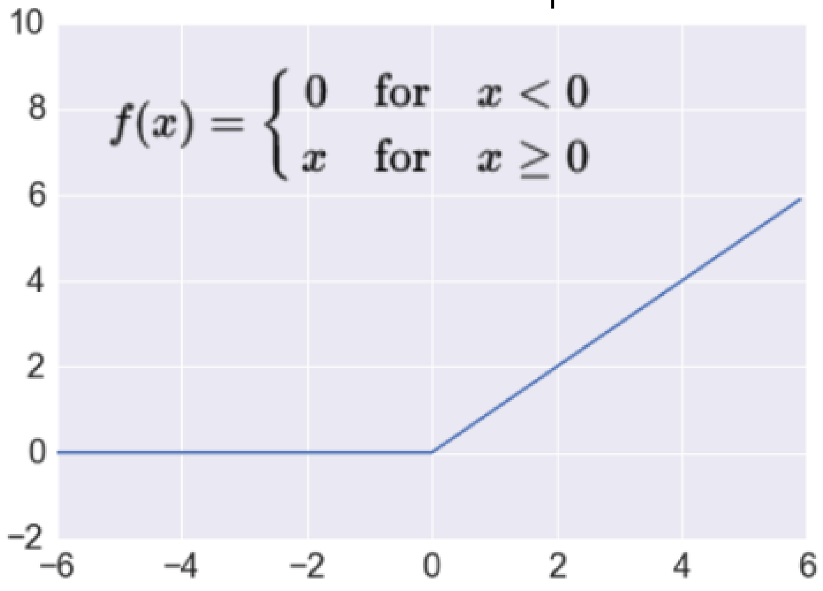
Slide credit: Ismini Lourentzou – Introduction to Deep Learning

Activation: ReLU

- ***ReLU*** (Rectified Linear Unit): takes a real-valued number and thresholds it at zero

$$f(x) = \max(0, x)$$

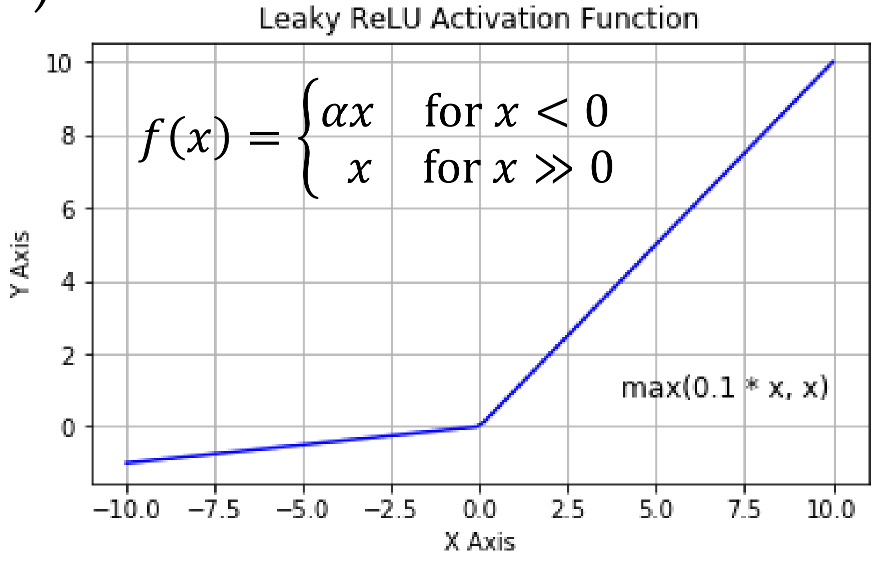
$$\mathbb{R}^n \to \mathbb{R}^n_+$$



  - Most modern deep NNs use ReLU activations
  - ReLU is fast to compute
    - Compared to sigmoid, tanh
    - Simply threshold a matrix at zero
  - Accelerates the convergence of gradient descent
    - Due to linear, non-saturating form
  - Prevents the gradient vanishing problem

Activation: Leaky ReLU

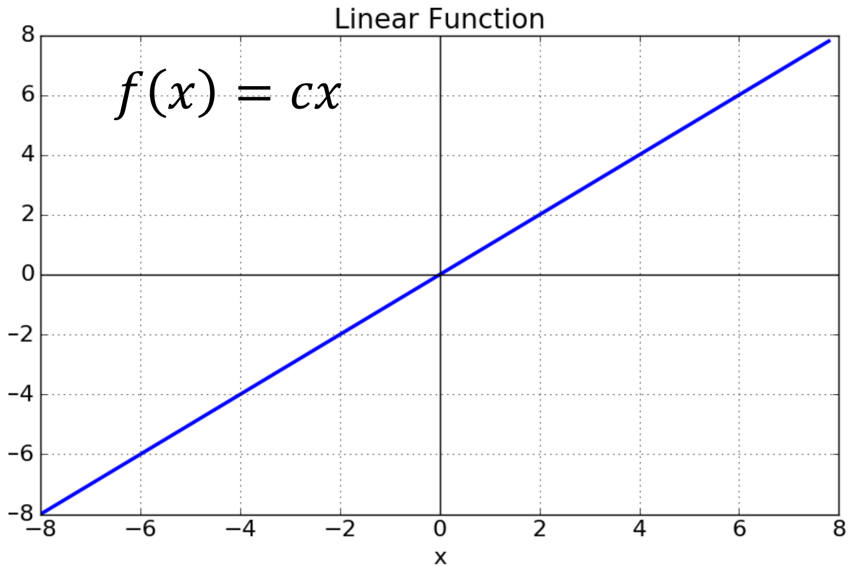
- The problem of ReLU activations: they can "die"
  - ReLU could cause weights to update in a way that the gradients can become zero and the neuron will not activate again on any data
  - E.g., when a large learning rate is used
- ***Leaky ReLU*** activation function is a variant of ReLU
  - Instead of the function being 0 when $x < 0$, a leaky ReLU has a small negative slope (e.g., $\alpha = 0.01$, or similar)
  - This resolves the dying ReLU problem
  - Most current works still use ReLU
    - With a proper setting of the learning rate, the problem of dying ReLU can be avoided



Activation: Linear Function

- *Linear function* means that the output signal is proportional to the input signal to the neuron
  - If the value of the constant c is 1, it is also called **identity activation function**
  - This activation type is used in regression problems
    - E.g., the last layer can have linear activation function, in order to output a real number (and not a class membership)

$\mathbb{R}^n \rightarrow \mathbb{R}^n$

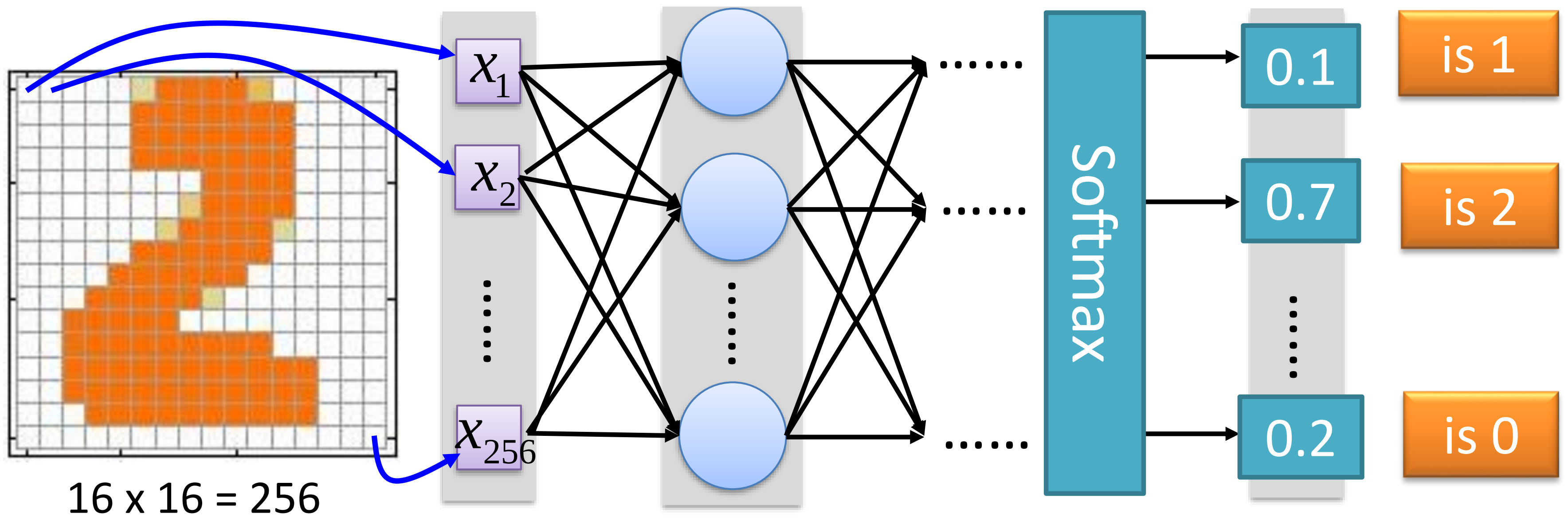


Training NNs

- The network ***parameters*** θ include the weight matrices and bias vectors from all layers

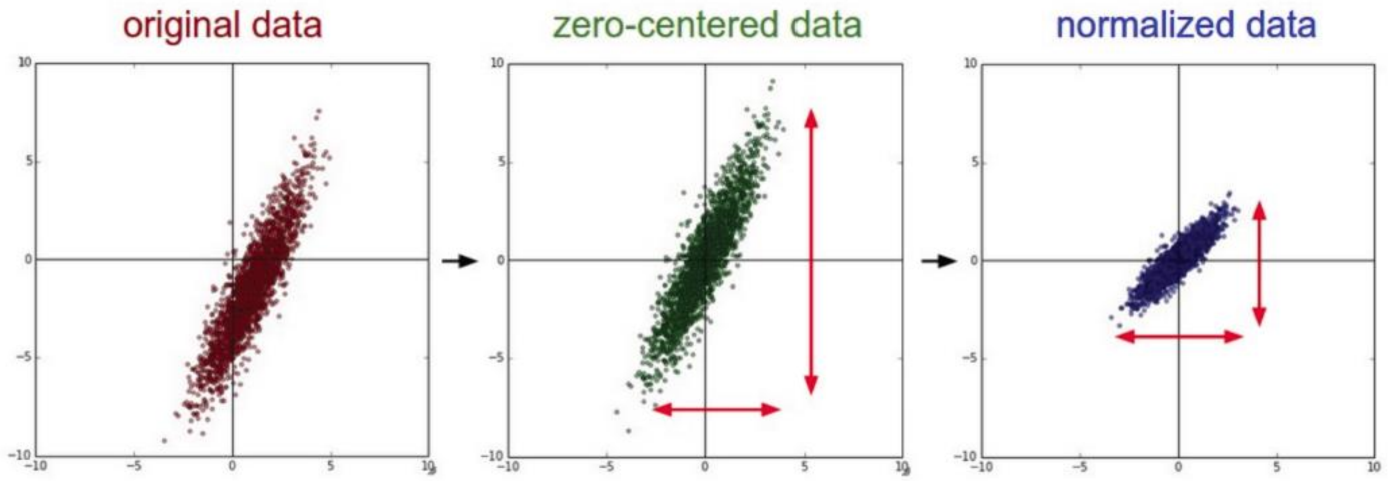
$$\theta = \{W^1, b^1, W^2, b^2, \cdots W^L, b^L\}$$

- Training a model to learn a set of parameters θ that are optimal (according to a criterion) is one of the greatest challenges in ML



Training NNs

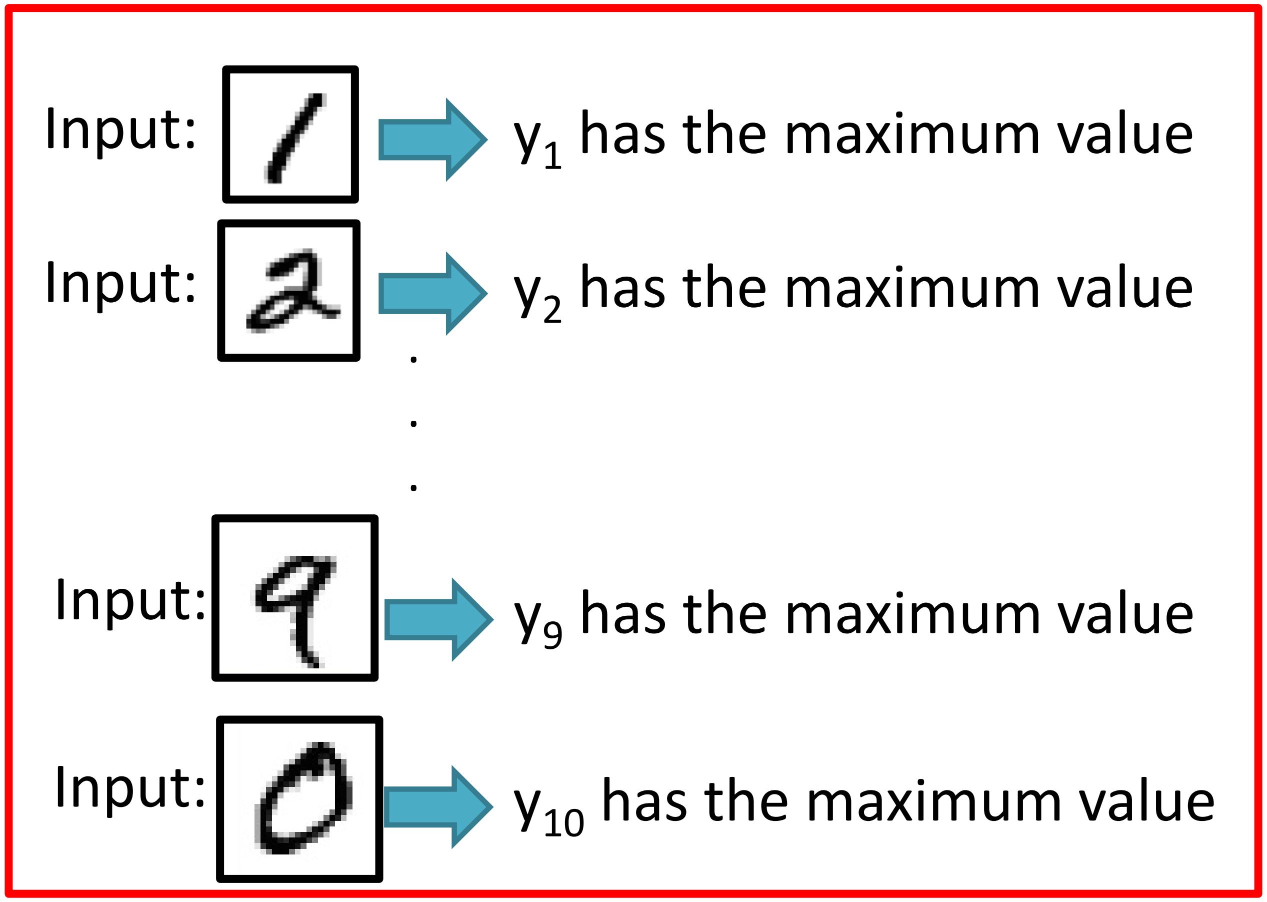
- *Data preprocessing* - helps convergence during training
  - Mean subtraction
    - Zero-centered data
      - Subtract the mean for each individual data dimension (feature)
  - Normalization
    - Divide each image by its standard deviation
      - To obtain standard deviation of 1 for each data dimension (feature)
    - Or, scale the data within the range [-1, 1]



Picture from: https://cs231n.github.io/neural-networks-2/

Training NNs

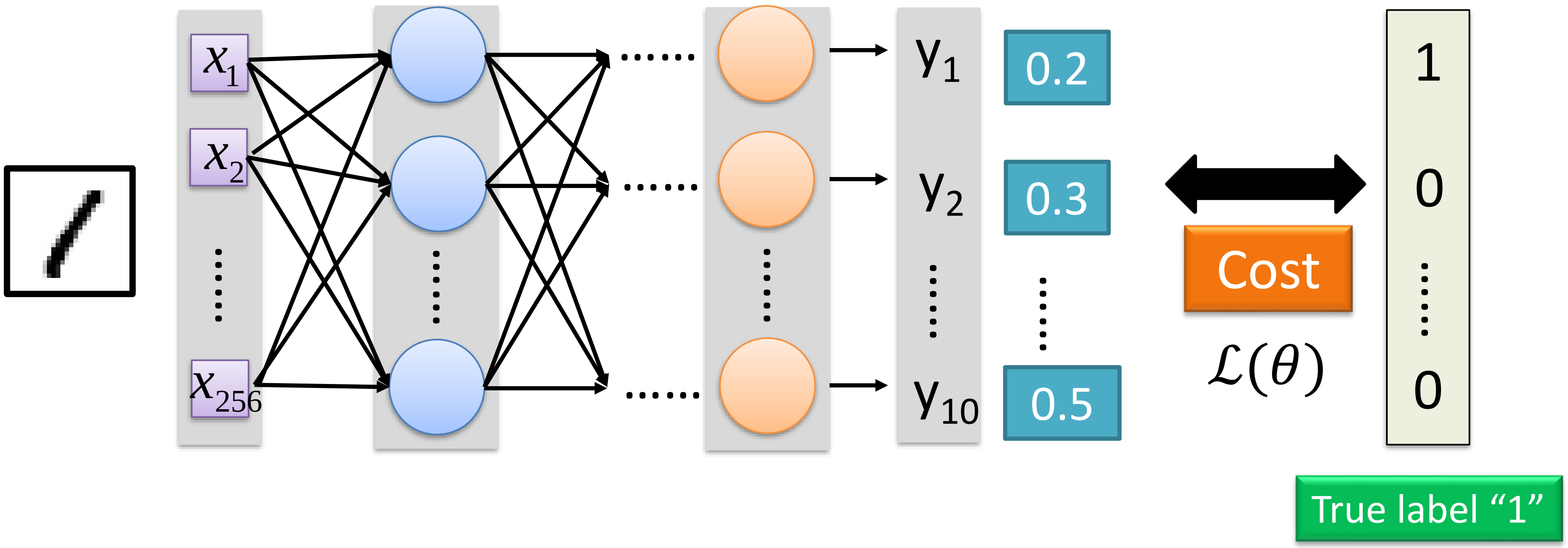
- To train a NN, set the parameters θ such that for a training subset of images, the corresponding elements in the predicted output have maximum values


Input:
y_1 has the maximum value
Input:
y_2 has the maximum value
Input:
y_9 has the maximum value
Input:
y_{10} has the maximum value

Slide credit: Hung-yi Lee – Deep Learning Tutorial

Training NNs

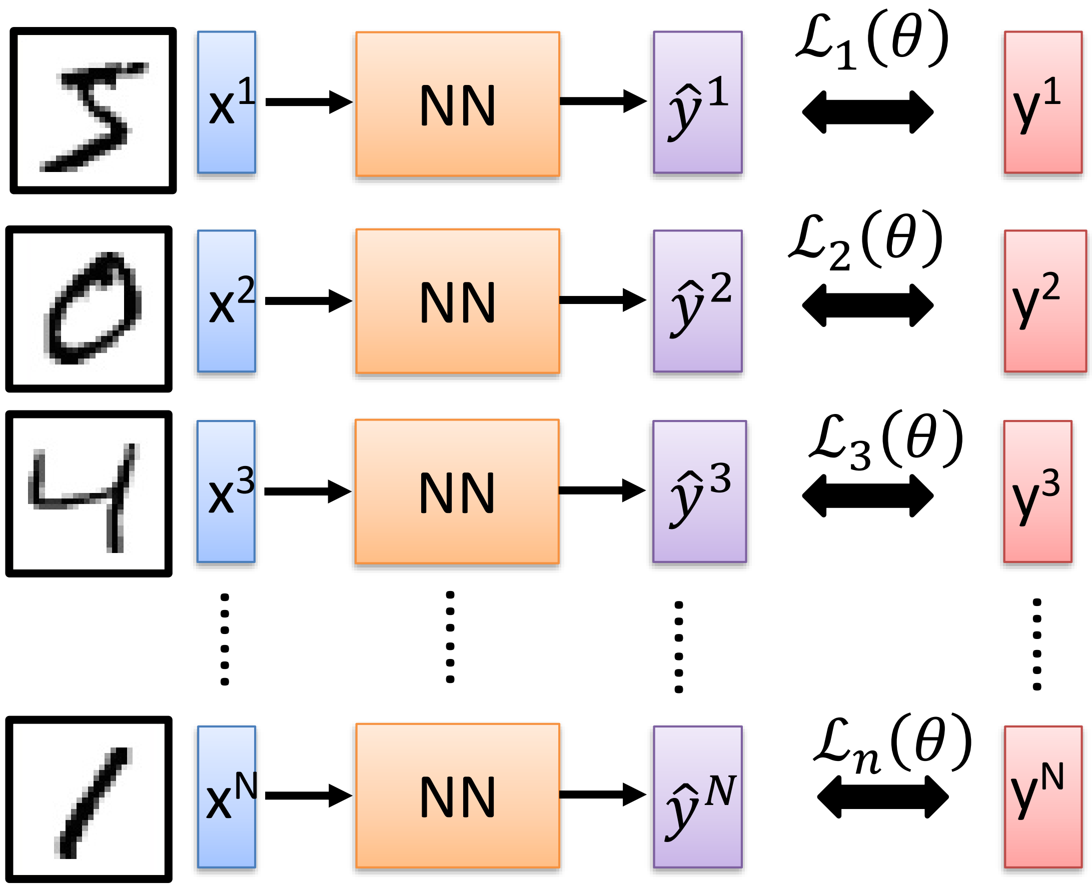
- Define an objective function/cost function/***loss function*** $\mathcal{L}(\theta)$ that calculates the difference between the model prediction and the true label
  - E.g., $\mathcal{L}(\theta)$ can be mean-squared error, cross-entropy, etc.



Slide credit: Hung-yi Lee – Deep Learning Tutorial

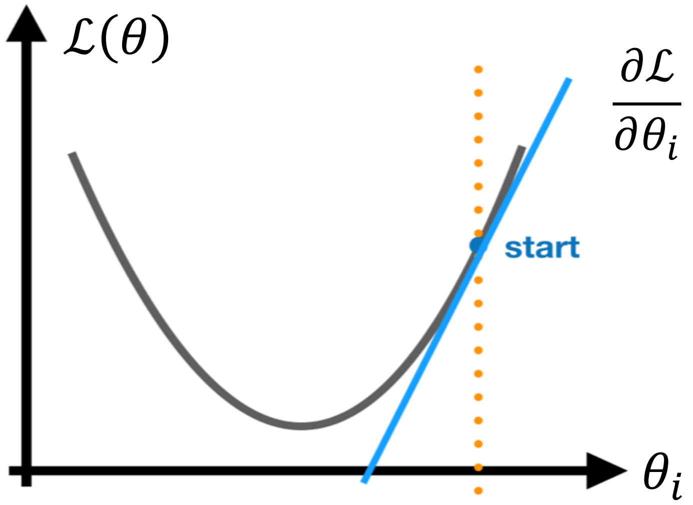
Training NNs

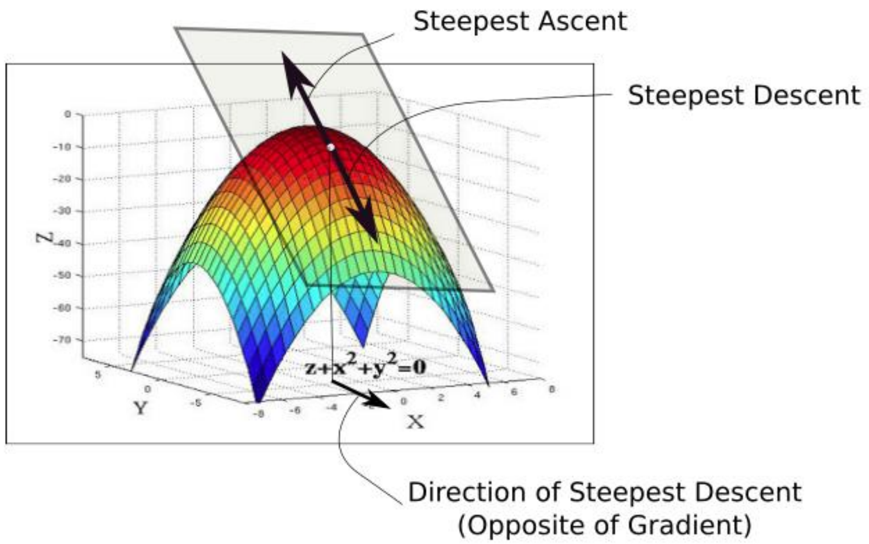
- For n training images, calculate the total loss: $\mathcal{L}(\theta) = \sum_{n=1}^{N} \mathcal{L}_n(\theta)$
- Find the optimal NN parameters θ^* that minimize the total loss $\mathcal{L}(\theta)$



Training NNs

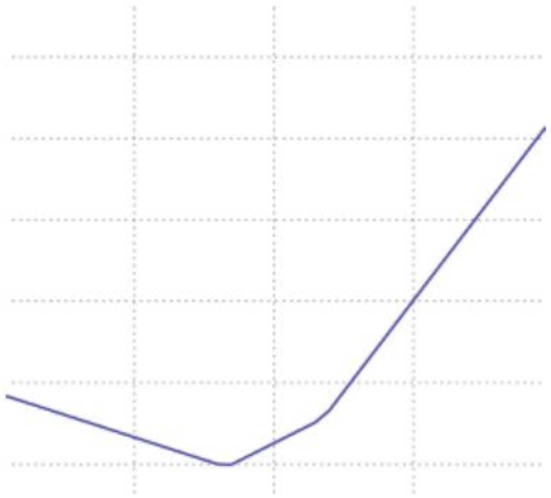
- Optimizing the loss function $\mathcal{L}(\theta)$
  - Almost all DL models these days are trained with a variant of the *gradient descent* (GD) algorithm
  - GD applies iterative refinement of the network parameters θ
  - GD uses the opposite direction of the gradient of the loss with respect to the NN parameters (i.e.,$\nabla\mathcal{L}(\theta) = [\partial\mathcal{L}/\partial\theta_i]$) for updating θ
    - The gradient of the loss function $\nabla\mathcal{L}(\theta)$ gives the direction of fastest increase of the loss function $\mathcal{L}(\theta)$ when the parameters θ are changed

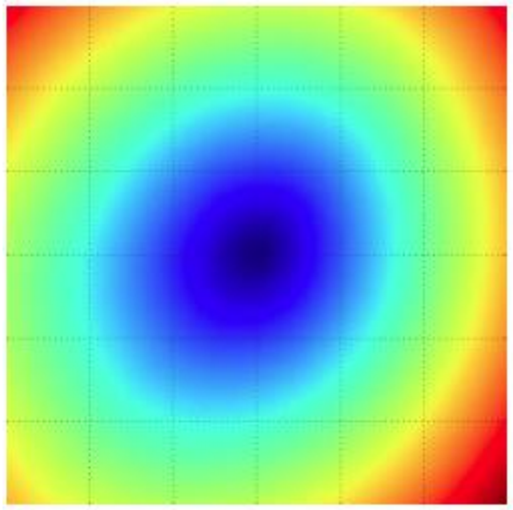




Training NNs

- The loss functions for most DL tasks are defined over very high-dimensional spaces
  - E.g., ResNet50 NN has about 23 million parameters
  - This makes the loss function impossible to visualize
- We can still gain intuitions by studying 1-dimensional and 2-dimensional examples of loss functions

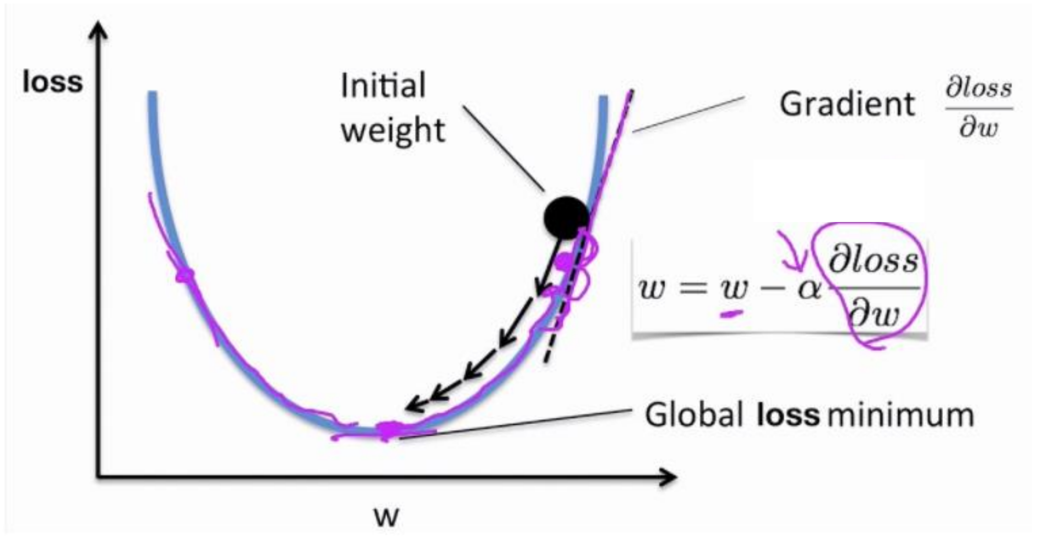

1D loss (the minimum point is obvious)


2D loss (blue = low loss, red = high loss)

Picture from: https://cs231n.github.io/optimization-1/

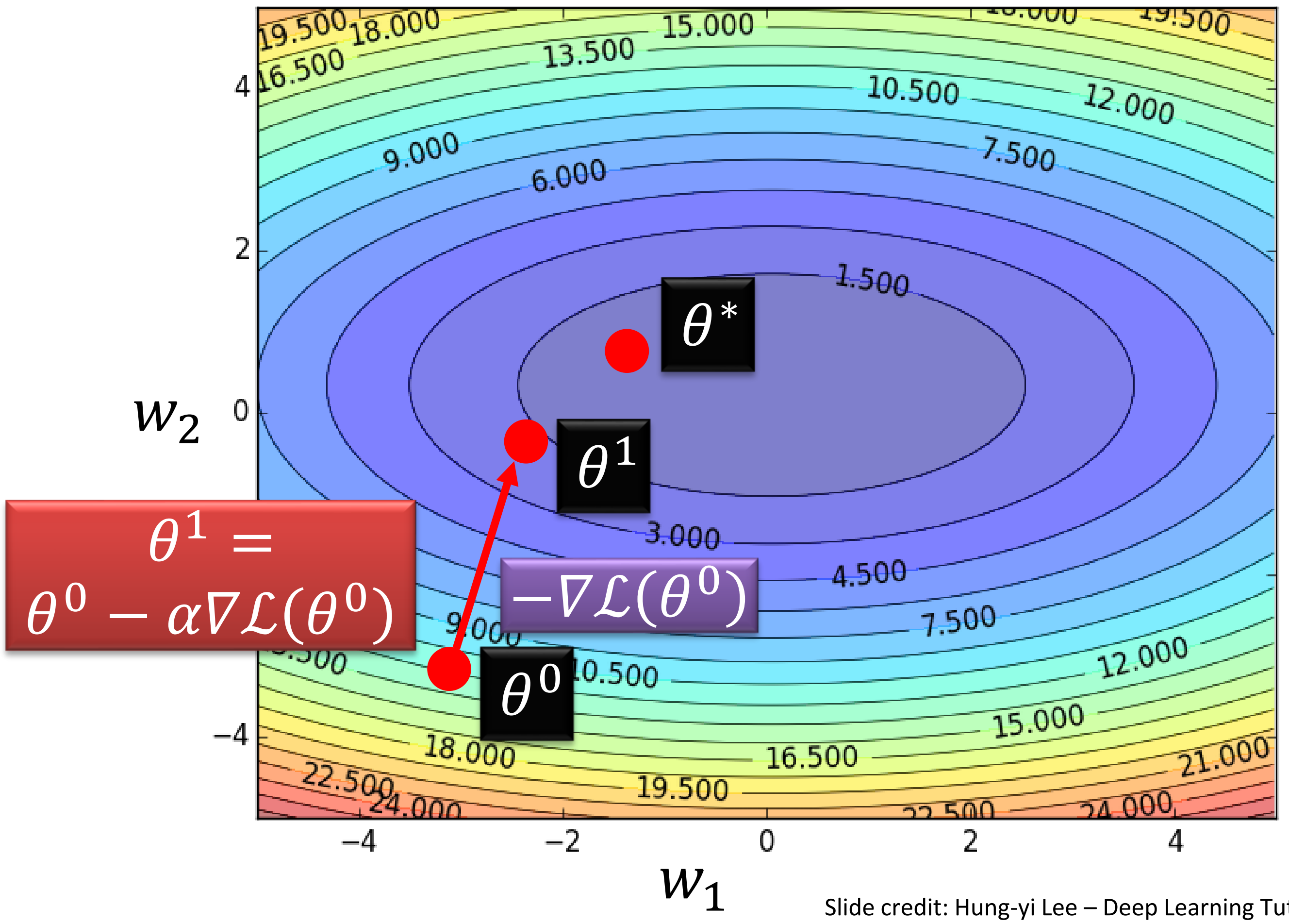
Gradient Descent Algorithm

- Steps in the gradient descent algorithm:
  1. Randomly initialize the model parameters, θ^0
     - In the figure, the parameters are denoted w
  2. Compute the gradient of the loss function at θ^0: $\nabla\mathcal{L}(\theta^0)$
  3. Update the parameters as: $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$
     - Where α is the learning rate
  4. Go to step 2 and repeat (until a terminating criterion is reached)


loss
Initial weight
Gradient $\frac{\partial loss}{\partial w}$
$w = w - \alpha \frac{\partial loss}{\partial w}$
Global loss minimum
w

Gradient Descent Algorithm

- Example: a NN with only 2 parameters w_1 and w_2, i.e., $\theta = \{w_1, w_2\}$
  - Different colors are the values of the loss (minimum loss θ^* is ≈ 1.3)



1. Randomly pick a starting point θ^0

2. Compute the gradient at θ^0, $\nabla\mathcal{L}(\theta^0)$

3. Times the learning rate η, and update θ, $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$

4. Go to step 2, repeat

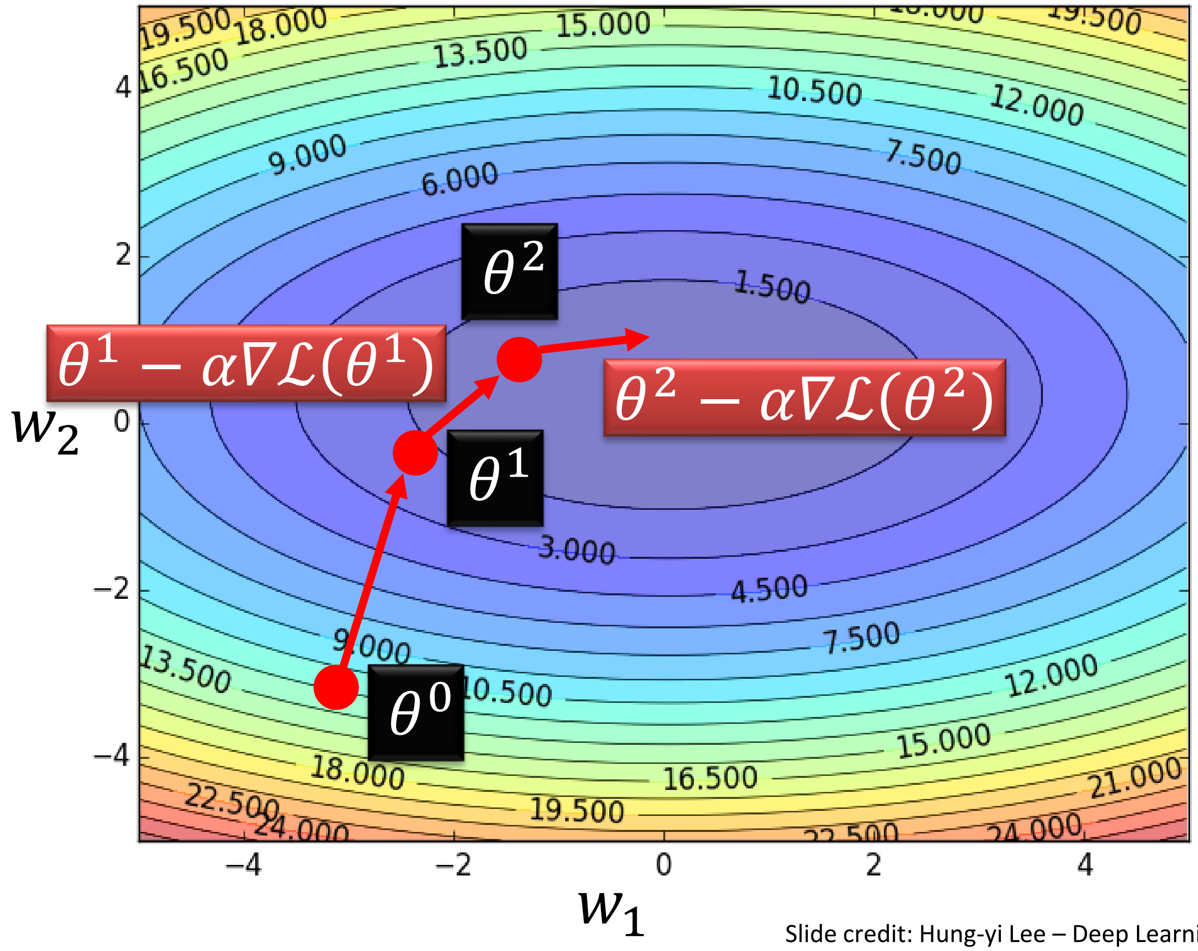
$$\nabla\mathcal{L}(\theta^0) = \begin{bmatrix} \partial\mathcal{L}(\theta^0)/\partial w_1 \\ \partial\mathcal{L}(\theta^0)/\partial w_2 \end{bmatrix}$$

Slide credit: Hung-yi Lee – Deep Learning Tutorial

Gradient Descent Algorithm

- Example (contd.)

Eventually, we would reach a minimum .....



1. Randomly pick a starting point θ^0

2. Compute the gradient at θ^0,$\nabla\mathcal{L}(\theta^0)$

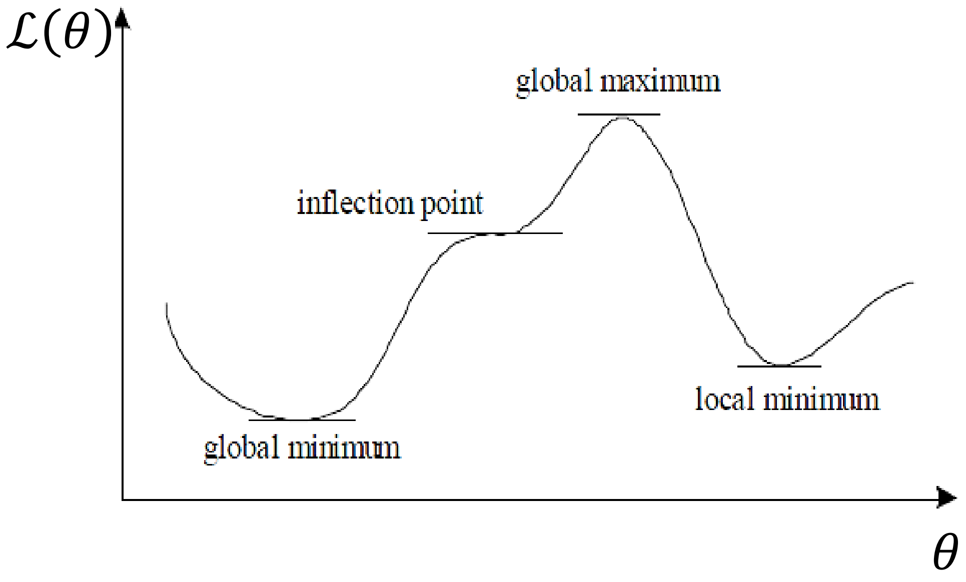
3. Times the learning rate η, and update θ,
$\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$

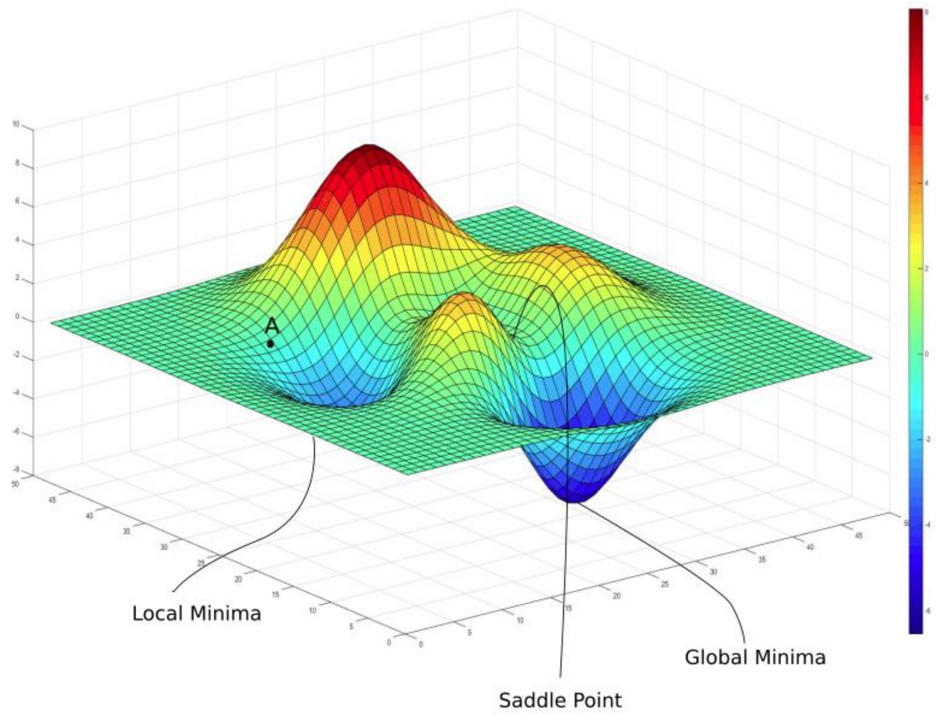
4. Go to step 2, repeat

Slide credit: Hung-yi Lee – Deep Learning Tutorial

Gradient Descent Algorithm

- Gradient descent algorithm stops when a local minimum of the loss surface is reached
  - GD does not guarantee reaching a global minimum
  - However, empirical evidence suggests that GD works well for NNs

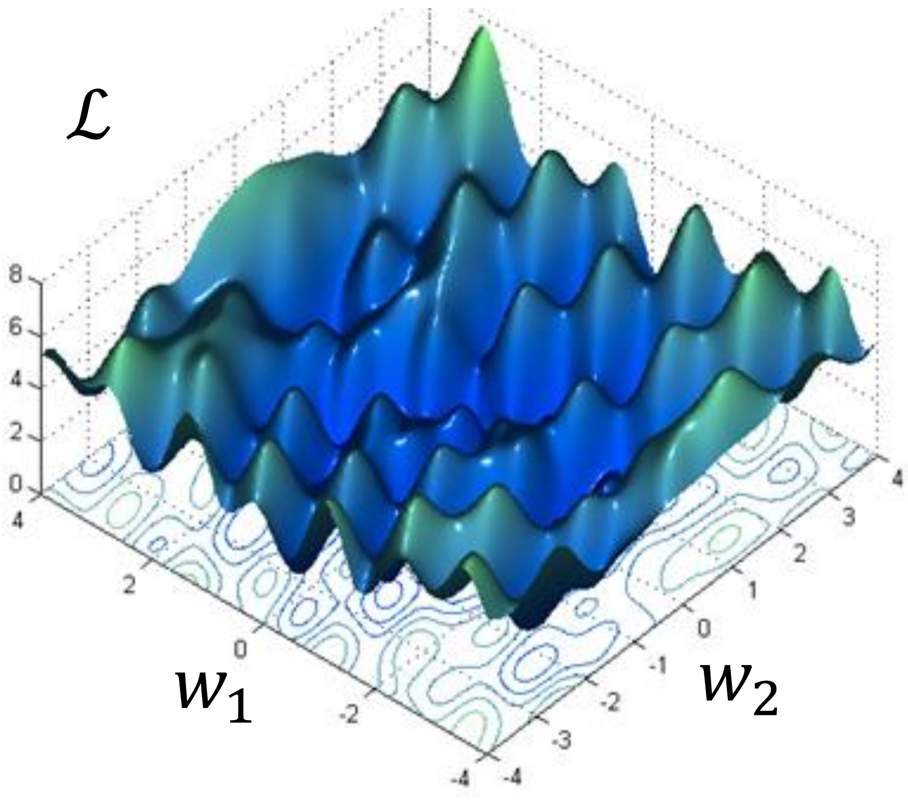




Picture from: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

Gradient Descent Algorithm

- For most tasks, the loss surface $\mathcal{L}(\theta)$ is highly complex (and non-convex)
- Random initialization in NNs results in different initial parameters θ^0
  - Gradient descent may reach different minima at every run
  - Therefore, NN will produce different predicted outputs
- Currently, we don't have an algorithm that guarantees reaching a global minimum for an arbitrary loss function



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Backpropagation

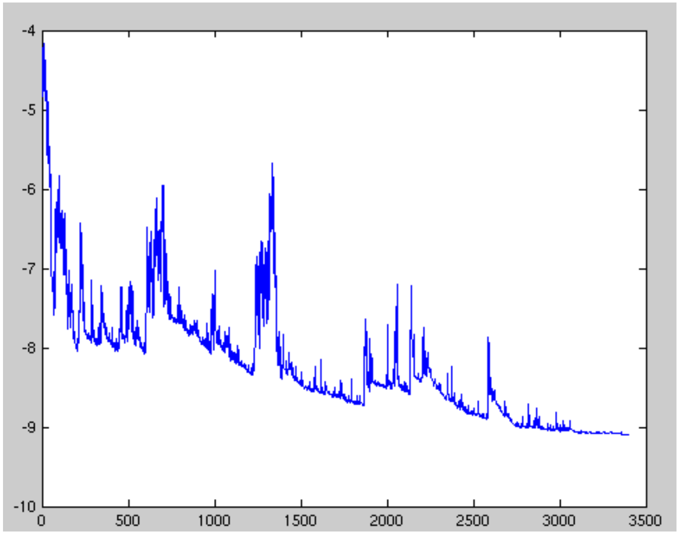
- How to calculate the gradients of the loss function in NNs?
- There are two ways:
  1. Numerical gradient: slow, approximate, but easy way
  2. Analytic gradient: requires calculus, fast, but more error-prone way
- In practice the analytic gradient is used
  – Analytical differentiation for gradient computation is available in almost all deep learning libraries

Mini-batch Gradient Descent

- It is wasteful to compute the loss over the entire set to perform a single parameter update for large datasets
  - E.g., ImageNet has 14M images
  - GD (a.k.a. vanilla GD) is replaced with mini-batch GD
- *Mini-batch gradient descent*
  - Approach:
    - Compute the loss $\mathcal{L}(\theta)$ on a batch of images, update the parameters θ, and repeat until all images are used
    - At the next epoch, shuffle the training data, and repeat above process
  - Mini-batch GD results in much faster training
  - Typical batch size: 32 to 256 images
  - It works because the examples in the training data are correlated
    - I.e., the gradient from a mini-batch is a good approximation of the gradient of the entire training set

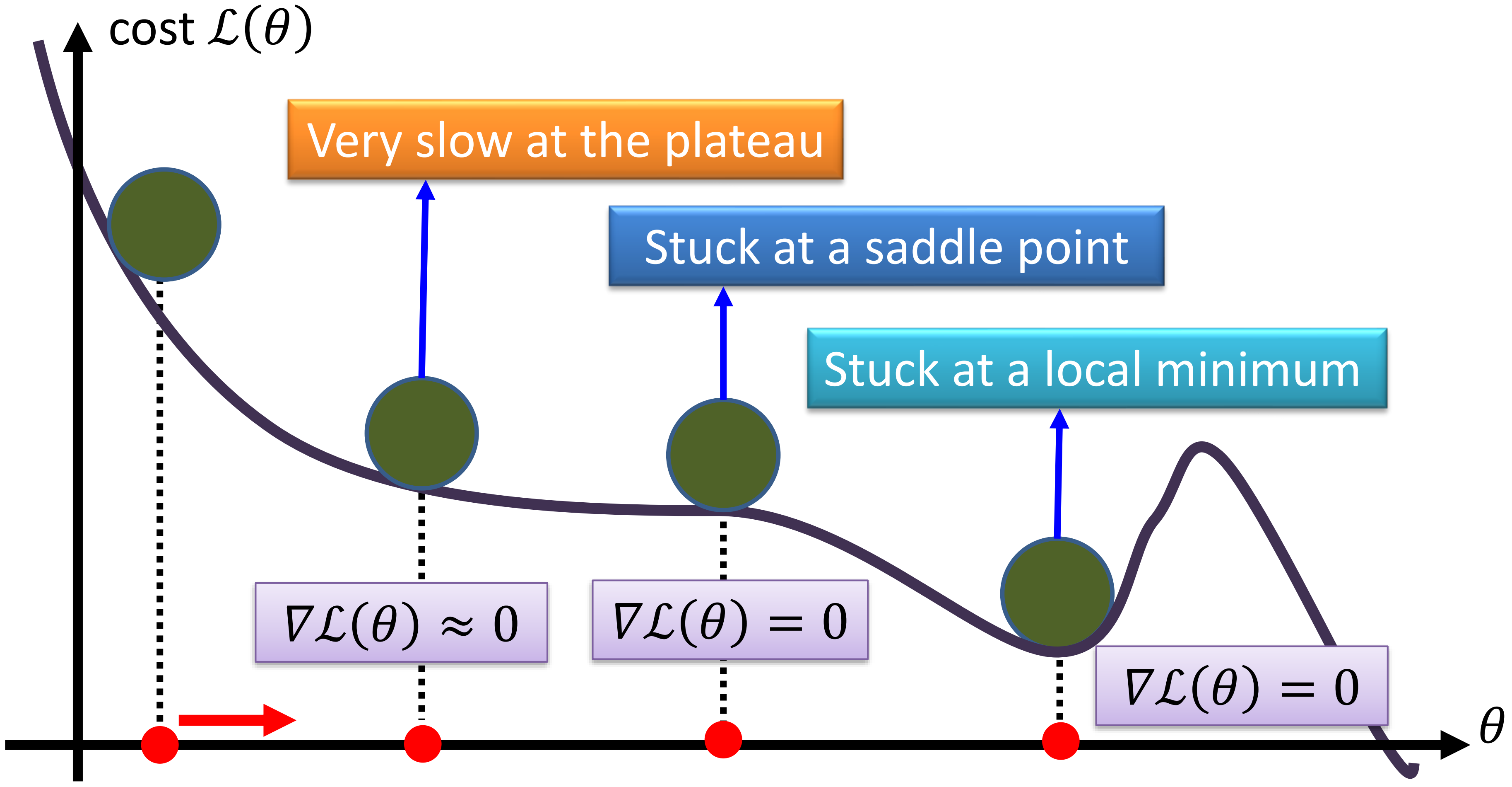
Stochastic Gradient Descent

- *Stochastic gradient descent*
  - SGD uses mini-batches that consist of a single input example
    - E.g., one image mini-batch
  - Although this method is very fast, it may cause significant fluctuations in the loss function
    - Therefore, it is less commonly used, and mini-batch GD is preferred
  - In most DL libraries, SGD is typically a mini-batch SGD (with an option to add momentum)



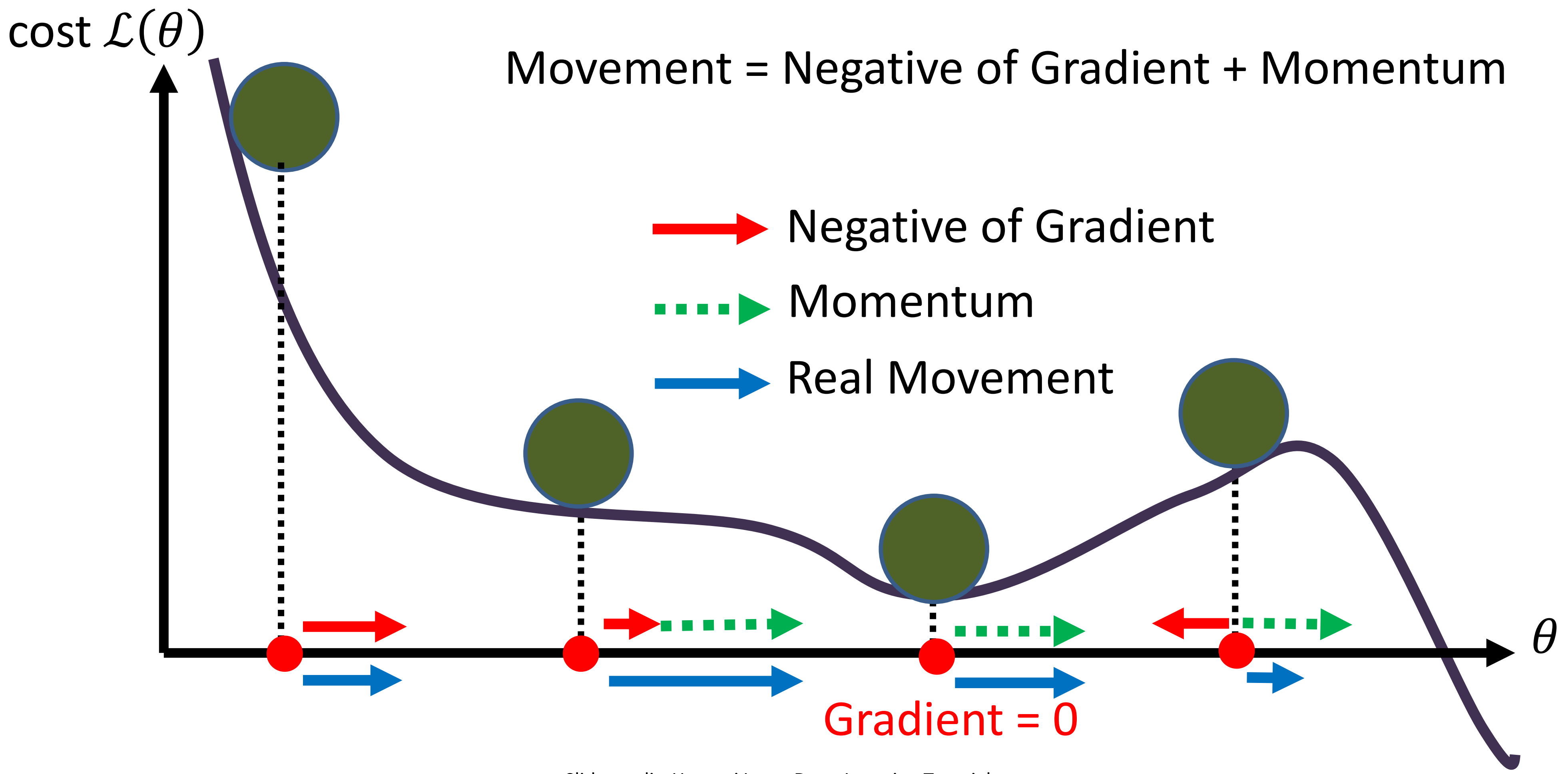
Problems with Gradient Descent

- Besides the local minima problem, the GD algorithm can be very slow at plateaus, and it can get stuck at saddle points


cost $\mathcal{L}(\theta)$
Very slow at the plateau
Stuck at a saddle point
Stuck at a local minimum
$\nabla\mathcal{L}(\theta) \approx 0$
$\nabla\mathcal{L}(\theta) = 0$
$\nabla\mathcal{L}(\theta) = 0$
θ

Gradient Descent with Momentum

- *Gradient descent with momentum* uses the momentum of the gradient for parameter optimization


cost $\mathcal{L}(\theta)$
Movement = Negative of Gradient + Momentum
Negative of Gradient
Momentum
Real Movement
θ
Gradient = 0

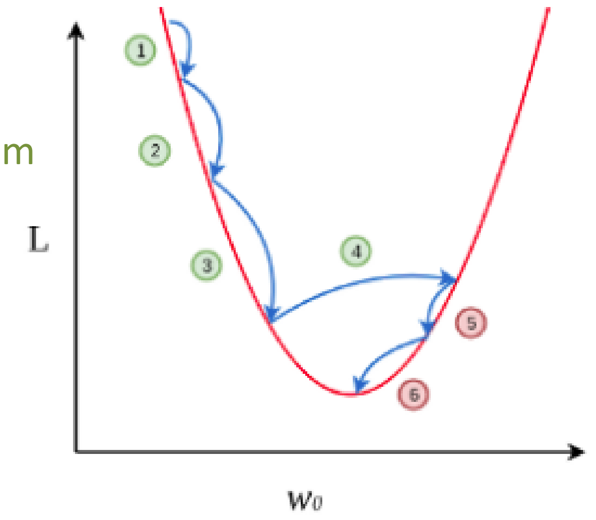
Gradient Descent with Momentum

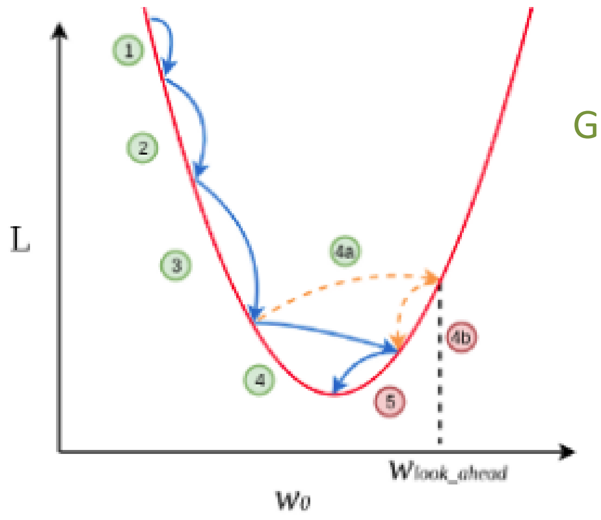
- Parameters update in GD with momentum : $\theta^{new} = \theta^{old} - V^{new}$
  - Where: $V^{new} = \beta V^{old} + \alpha \nabla \mathcal{L}(\theta^{old})$
- Compare to vanilla GD: $\theta^{new} = \theta^{old} - \alpha \nabla \mathcal{L}(\theta^{old})$
- The term V^{new} is called momentum
  - This term accumulates the gradients from the past several steps
  - It is similar to a momentum of a heavy ball rolling down the hill
- The parameter β referred to as a coefficient of momentum
  - A typical value of the parameter β is 0.9
- This method updates the parameters θ in the direction of the weighted average of the past gradients

Nesterov Accelerated Momentum

- ***Gradient descent with Nesterov accelerated momentum***
  - Parameters update: $\theta^{new} = \theta^{old} - V^{new}$
    - Where: $V^{new} = \beta V^{old} + \alpha \nabla \mathcal{L}(\theta^{old} - \beta V^{old})$
  - The term $\theta^{old} - \beta V^{old}$ allows us to predict the position of the parameters in the next step (i.e., $\theta^{next} \approx \theta^{old} - \beta V^{old}$)
  - The gradient is calculated with respect to the approximate future position of the parameters in the next step, θ^{next}

GD with momentum





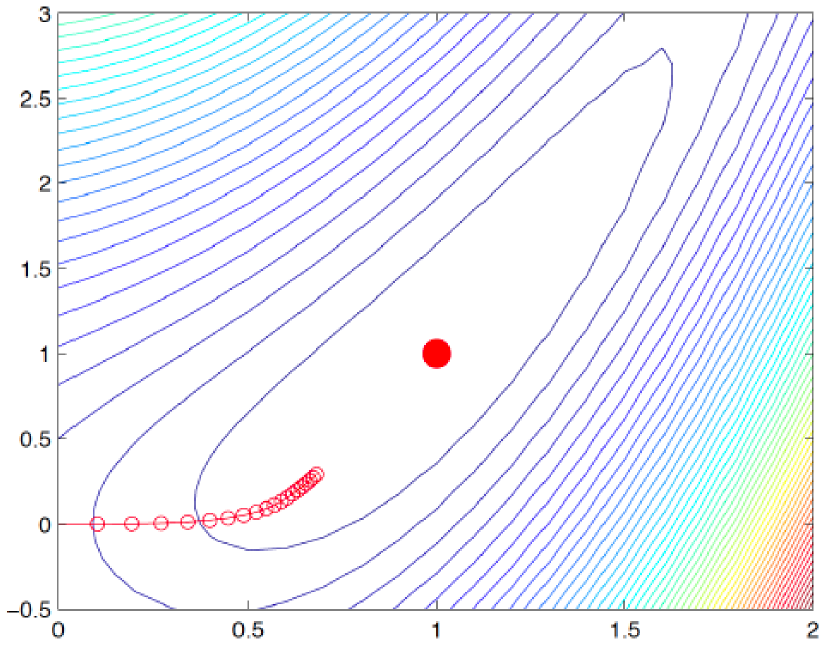
GD with Nesterov momentum

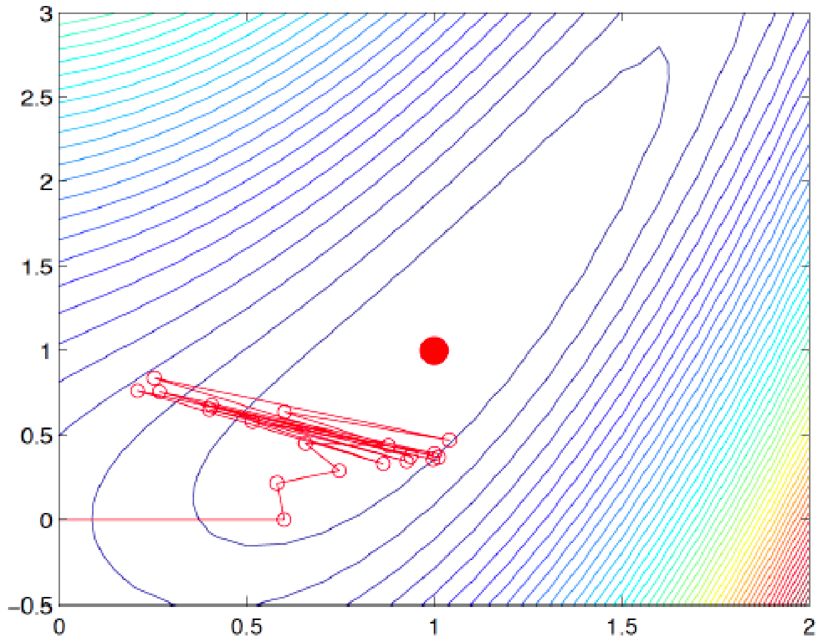
Picture from: https://towardsdatascience.com/learning-parameters-part-2-a190bef2d12

Learning Rate

- *Learning rate*
  - The gradient tells us the direction in which the loss has the steepest rate of increase, but it does not tell us how far along the opposite direction we should step
  - Choosing the learning rate (also called the step size) is one of the most important hyper-parameter settings for NN training

LR too small

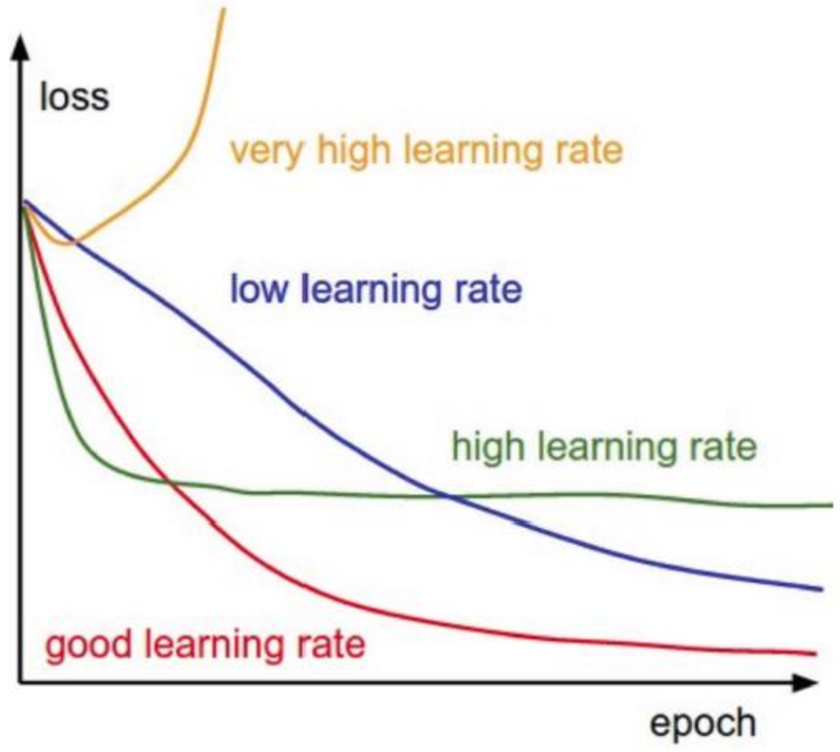




LR too large

Learning Rate

- Training loss for different learning rates
  - High learning rate: the loss increases or plateaus too quickly
  - Low learning rate: the loss decreases too slowly (takes many epochs to reach a solution)



Picture from: https://cs231n.github.io/neural-networks-3/

Annealing the Learning Rate

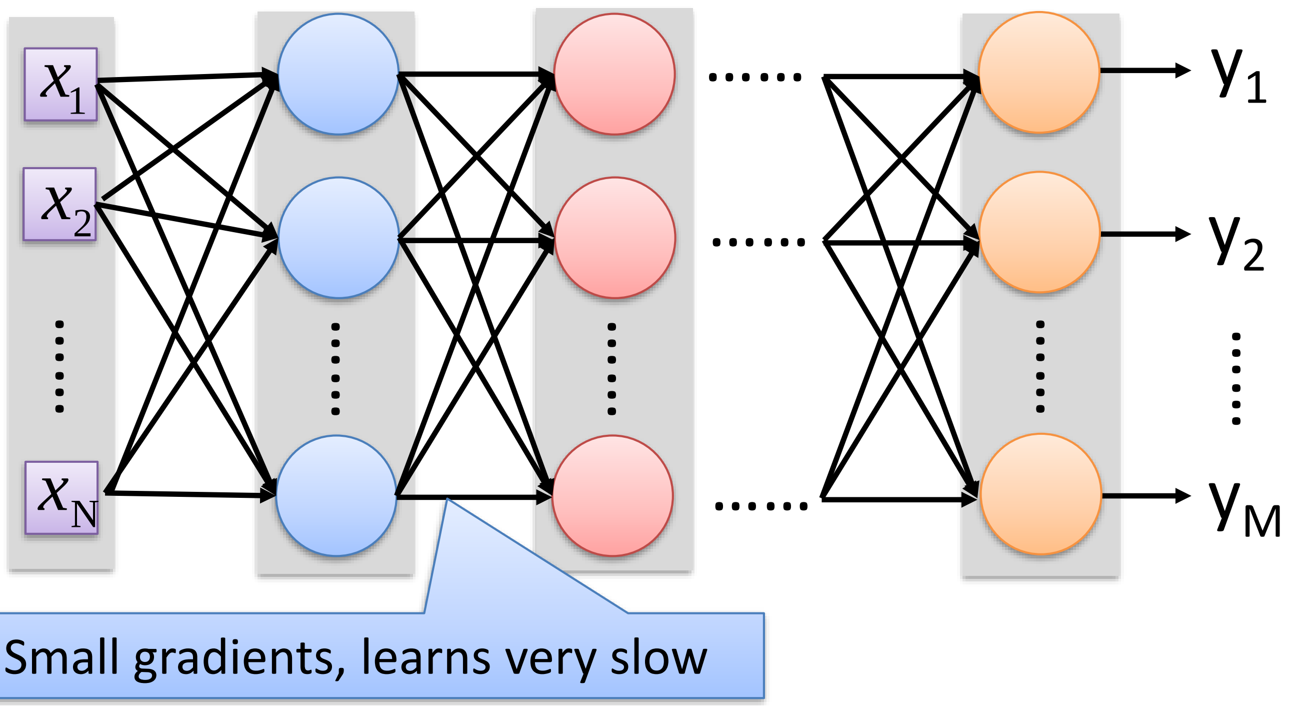
- Reduce the learning rate over time (learning rate decay)
  - Approach 1
    - Reduce the learning rate by some factor every few epochs
      - Typical values: reduce the learning rate by a half every 5 epochs, or by 10 every 20 epochs
      - Exponential decay reduces the learning rate exponentially over time
      - These numbers depend heavily on the type of problem and the model
  - Approach 2
    - Reduce the learning rate by a constant (e.g., by half) whenever the validation loss stops improving
      - In TensorFlow: tf.keras.callbacks.ReduceLROnPleateau()
        - » Monitor: validation loss
        - » Factor: 0.1 (i.e., divide by 10)
        - » Patience: 10 (how many epochs to wait before applying it)
        - » Minimum learning rate: 1e-6 (when to stop)

Adam

- *Adaptive Moment Estimation (Adam)*
  - Adam computes adaptive learning rates for each dimension of θ
    - Similar to GD with momentum, Adam computes a weighted average of past gradients, i.e., $V^{new} = \beta_1 V^{old} + (1-\beta_1)\nabla\mathcal{L}\left(\theta^{old}\right)$
    - Adam also computes a weighted average of past squared gradients, i.e., $U^{new} = \beta_2 U^{old} + (1-\beta_2)\left(\nabla\mathcal{L}\left(\theta^{old}\right)\right)^2$
  - The parameters update is: $\theta^{new} = \theta^{old} - \frac{\alpha}{\sqrt{\hat{V}^{new}}+\epsilon}\hat{U}^{new}$
    - Where: $\hat{V}^{new} = \frac{V^{new}}{1-\beta_1}$ and $\hat{U}^{new} = \frac{U^{new}}{1-\beta_2}$
    - The proposed default values are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$
- Other commonly used optimization methods include:
  - Adagrad, Adadelta, RMSprop, Nadam, etc.
  - Most papers nowadays used Adam and SGD with momentum

Vanishing Gradient Problem

- In some cases, during training, the gradients can become either very small (vanishing gradients) of very large (exploding gradients)
  - They result in very small or very large update of the parameters
  - Solutions: ReLU activations, regularization, LSTM units in RNNs



Loss Functions

- Classification tasks

Training examples	$\mathbb{R}^n \times \{class_1, \ldots, class_m\}$ (one-hot encoding)
Output Layer	Softmax Activation [maps $\mathbb{R}^m$ to a probability distribution] $P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$
Loss function	*Cross-entropy* $\mathcal{L}(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\left[y_k^{(i)} \log \hat{y}_k^{(i)} + \left(1 - y_k^{(i)}\right) \log\left(1 - \hat{y}_k^{(i)}\right)\right]$

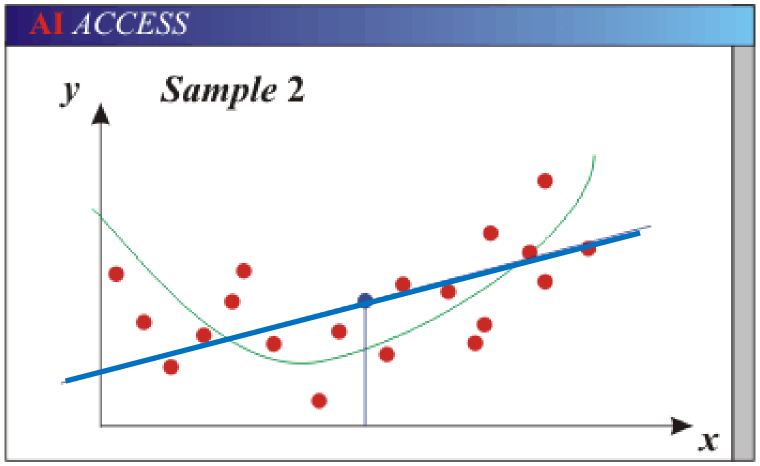
Loss Functions

- Regression tasks

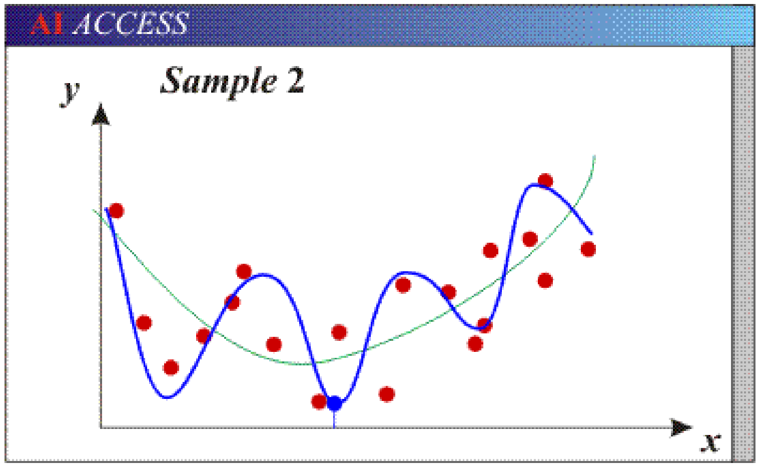
Training examples	$\mathbb{R}^n \times \mathbb{R}^m$
Output Layer	Linear (Identity) or Sigmoid Activation
Loss function	***Mean Squared Error*** $\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$
	Mean Absolute Error $\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left\|y^{(i)} - \hat{y}^{(i)}\right\|$

Generalization

- ***Underfitting***
  - The model is too “simple” to represent all the relevant class characteristics
  - Model with too few parameters
  - High error on the training set and high error on the testing set
- ***Overfitting***
  - The model is too “complex” and fits irrelevant characteristics (noise) in the data
  - Model with too many parameters
  - Low error on the training error and high error on the testing set

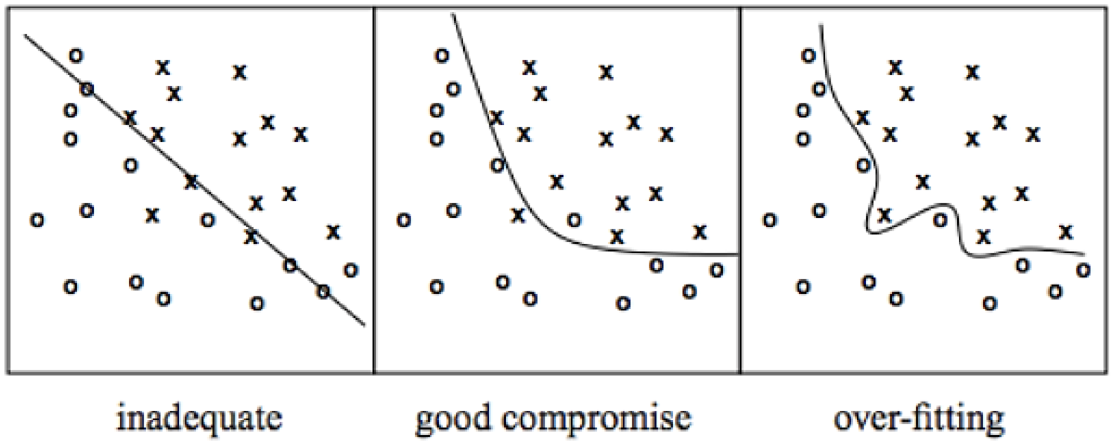


Blue line – decision boundary by the model
Green line – optimal decision boundary

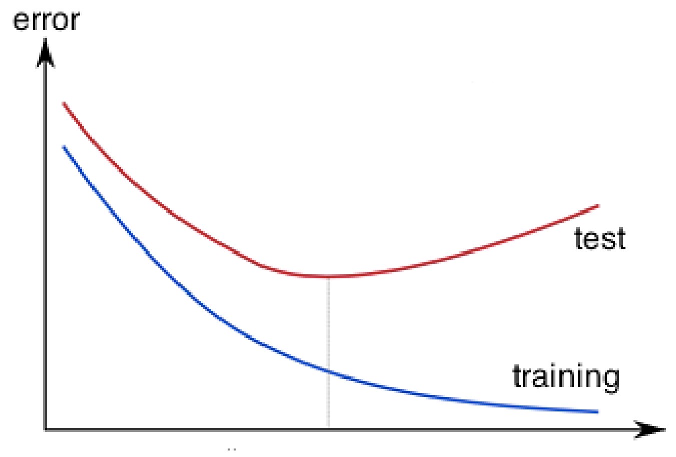


Overfitting

- A model with high capacity fits the noise in the data instead of the underlying relationship



- The model may fit the training data very well, but fails to **generalize** to new examples (test data)



Picture from: http://cs231n.github.io/assets/nn1/layer_sizes.jpeg

Regularization: Weight Decay

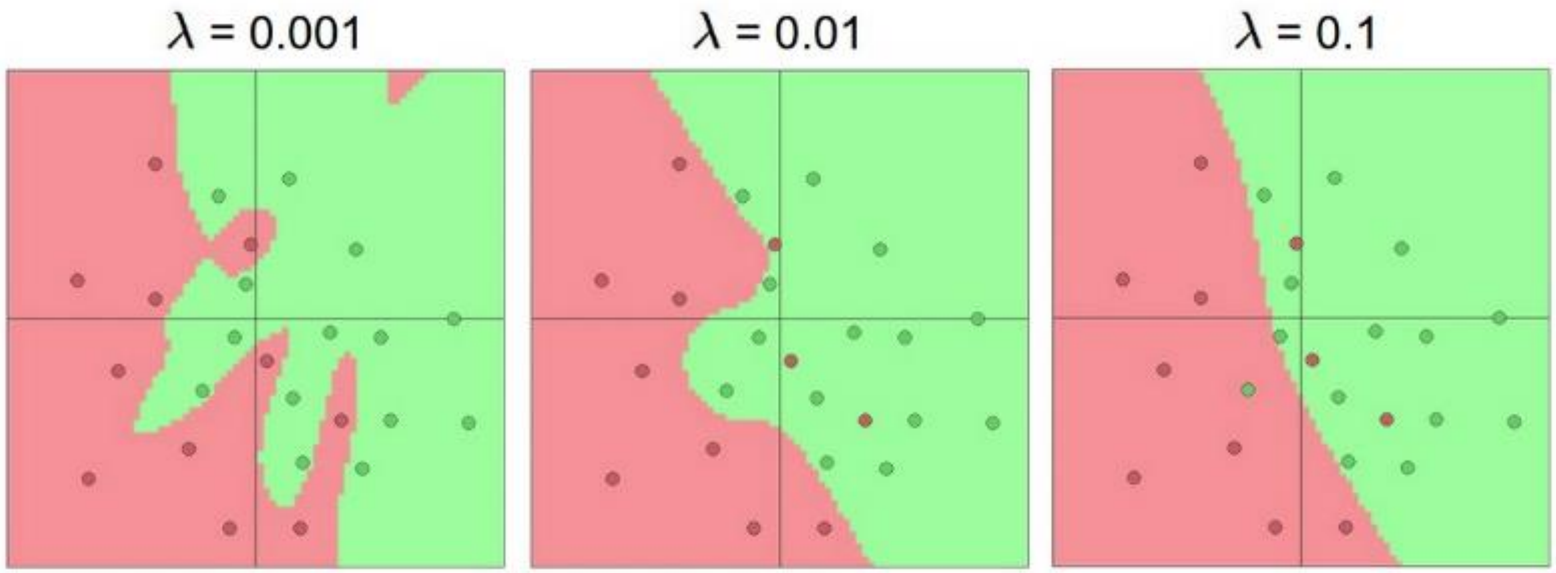
- ***ℓ_2 weight decay***
  - A regularization term that penalizes large weights is added to the loss function

$$\mathcal{L}_{reg}(\theta) = \overbrace{\mathcal{L}(\theta)}^{\text{Data loss}} + \overbrace{\lambda \sum_k \theta_k^2}^{\text{Regularization loss}}$$

  - For every weight in the network, we add the regularization term to the loss value
    - During gradient descent parameter update, every weight is decayed linearly toward zero
  - The weight decay coefficient λ determines how dominant the regularization is during the gradient computation

Regularization: Weight Decay

- Effect of the decay coefficient λ
  - Large weight decay coefficient → penalty for weights with large values



Regularization: Weight Decay

- ***ℓ_1 weight decay***
  - The regularization term is based on the ℓ_1 norm of the weights

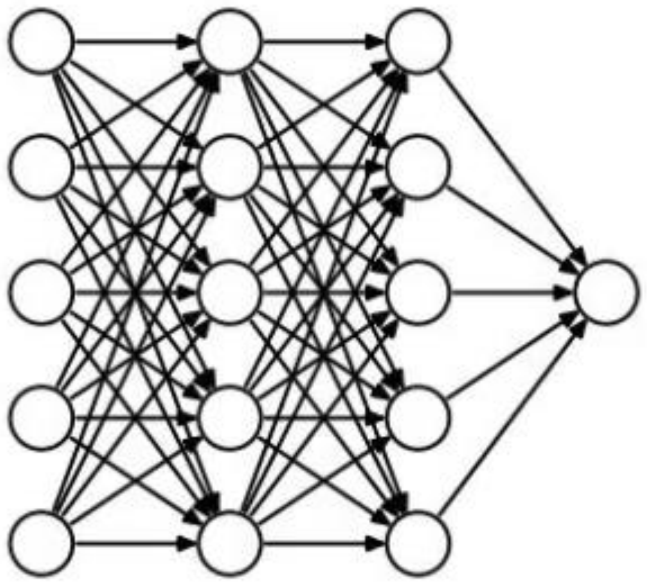
$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k |\theta_k|$$

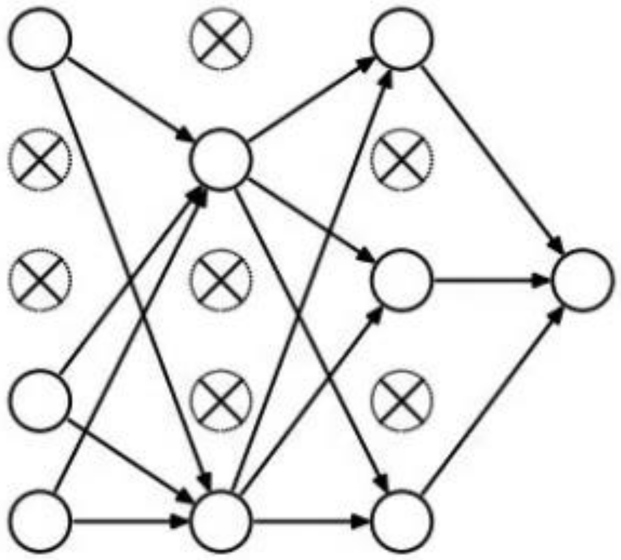
  - ℓ_1 weight decay is less common with NN
    - Often performs worse than ℓ_2 weight decay
  - It is also possible to combine ℓ_1 and ℓ_2 regularization
    - Called elastic net regularization

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda_1 \sum_k |\theta_k| + \lambda_2 \sum_k \theta_k^2$$

Regularization: Dropout

- *Dropout*
  - Randomly drop units (along with their connections) during training
  - Each unit is retained with a fixed dropout rate p, independent of other units
  - The hyper-parameter p to be chosen (tuned)
    - Often between 20 and 50 % of the units are dropped

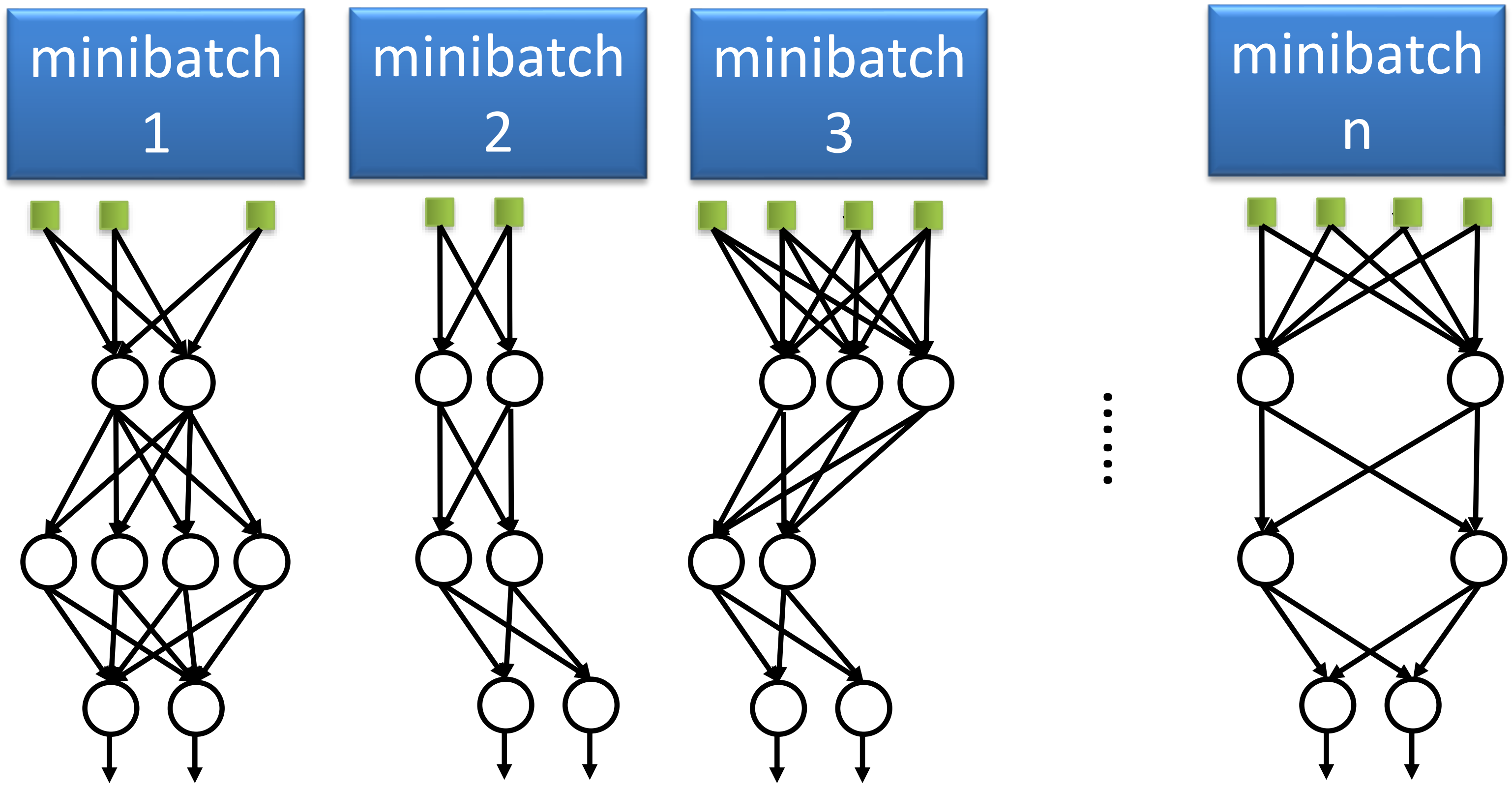




Slide credit: Hung-yi Lee – Deep Learning Tutorial

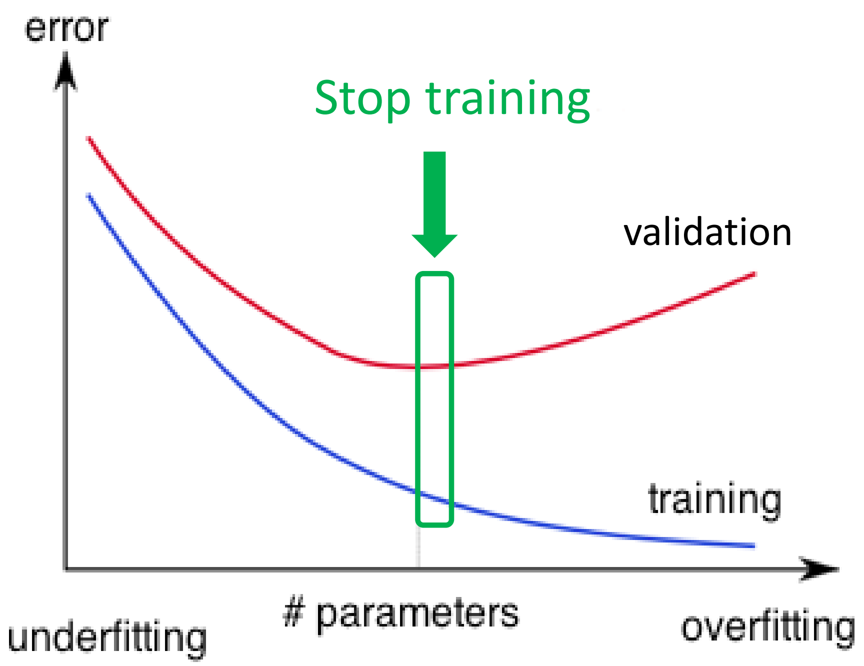
Regularization: Dropout

- Dropout is a kind of ensemble learning
  - Using one mini-batch to train one network with a slightly different architecture



Regularization: Early Stopping

- *Early-stopping*
  - During model training, use a validation set, along with a training set
    - A ratio of about 25% to 75% of the data is often used
  - Stop when the validation accuracy (or loss) has not improved after *n* subsequent epochs
    - The parameter *n* is called patience



Batch Normalization

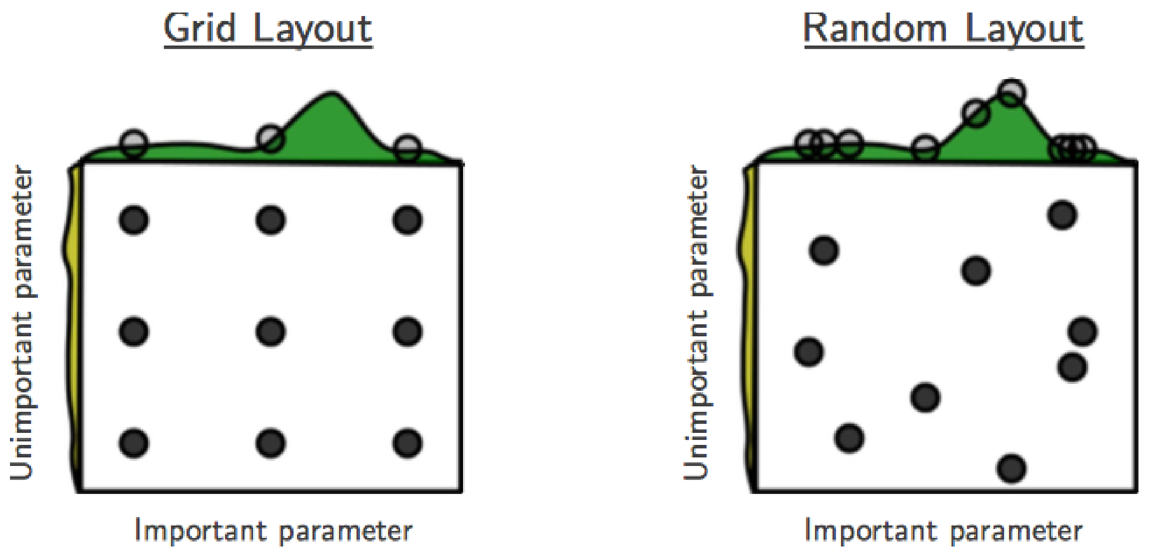
- ***Batch normalization layers*** act similar to the data preprocessing steps mentioned earlier
  - They calculate the mean μ and variance σ of a batch of input data, and normalize the data x to a zero mean and unit variance
  - I.e., $\hat{x} = \frac{x-\mu}{\sigma}$
- BatchNorm layers alleviate the problems of proper initialization of the parameters and hyper-parameters
  - Result in faster convergence training, allow larger learning rates
  - Reduce the internal covariate shift
- The BatchNorm layers are inserted immediately after convolutional layers or fully-connected layers, and before activation layers
  - They are very common with convolutional NNs

Hyper-parameter Tuning

- Training NNs can involve many hyper-parameter settings
- The most common hyper-parameters include:
  - Number of layers, and number of neurons per layer
  - Initial learning rate
  - Learning rate decay schedule (e.g., decay constant)
- Other hyper-parameters may include:
  - Regularization parameters (ℓ_2 penalty, dropout rate)
  - Batch size
- Hyper-parameter tuning can be time-consuming for larger NNs

Hyper-parameter Tuning

- Grid search
  - Check all values in a range with a step value
- Random search
  - Randomly sample values for the parameter
  - Often preferred to grid search
- Bayesian hyper-parameters optimization
  - Is an active area of research

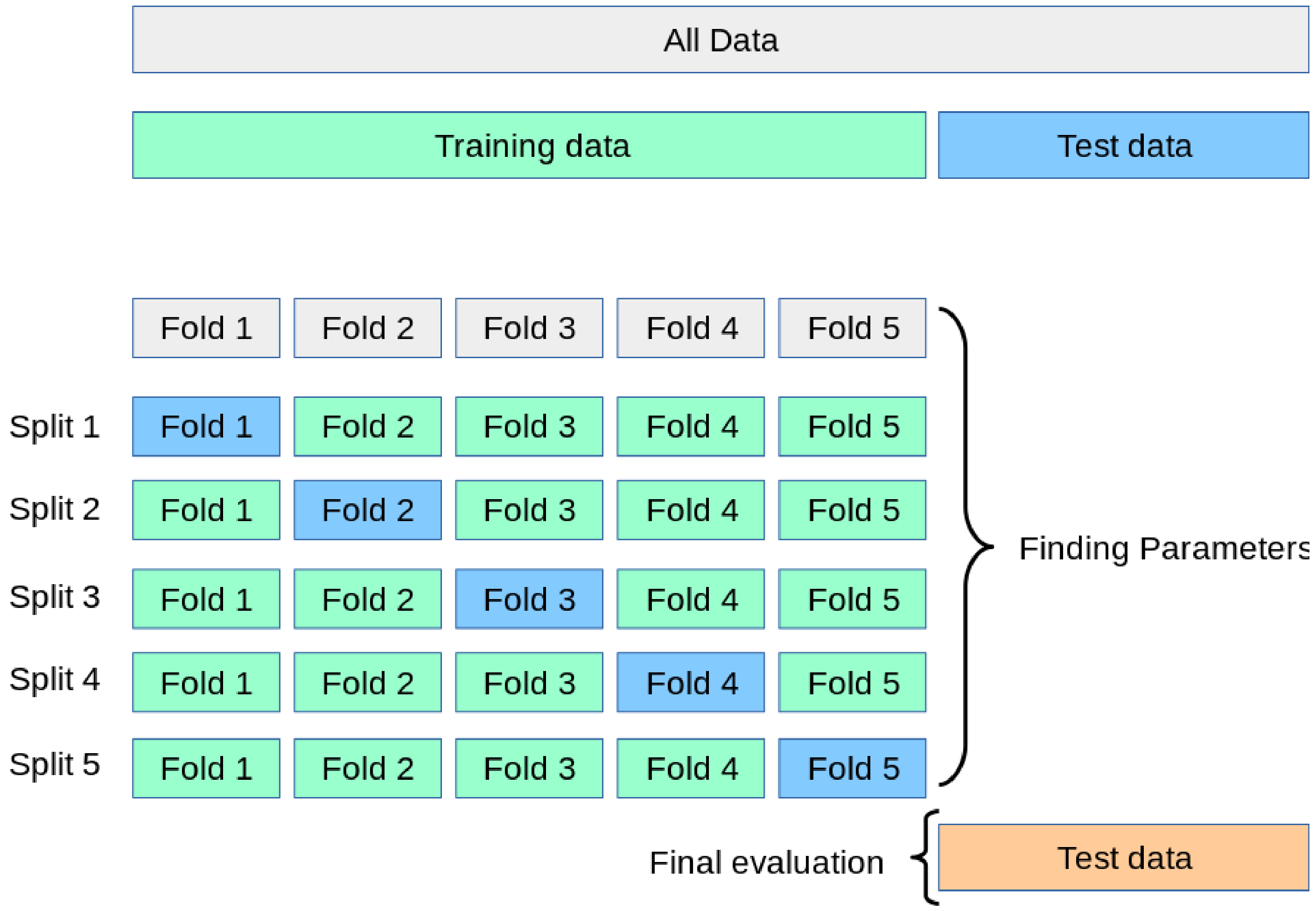


k-Fold Cross-Validation

- Using ***k-fold cross-validation*** for hyper-parameter tuning is common when the size of the training data is small
  - It leads to a better and less noisy estimate of the model performance by averaging the results across several folds
- E.g., 5-fold cross-validation (see the figure on the next slide)
  1. Split the train data into 5 equal folds
  2. First use folds 2-5 for training and fold 1 for validation
  3. Repeat by using fold 2 for validation, then fold 3, fold 4, and fold 5
  4. Average the results over the 5 runs
  5. Once the best hyper-parameters are determined, evaluate the model on the set aside test data

k-Fold Cross-Validation

- Illustration of a 5-fold cross-validation



Picture from: https://scikit-learn.org/stable/modules/cross_validation.html

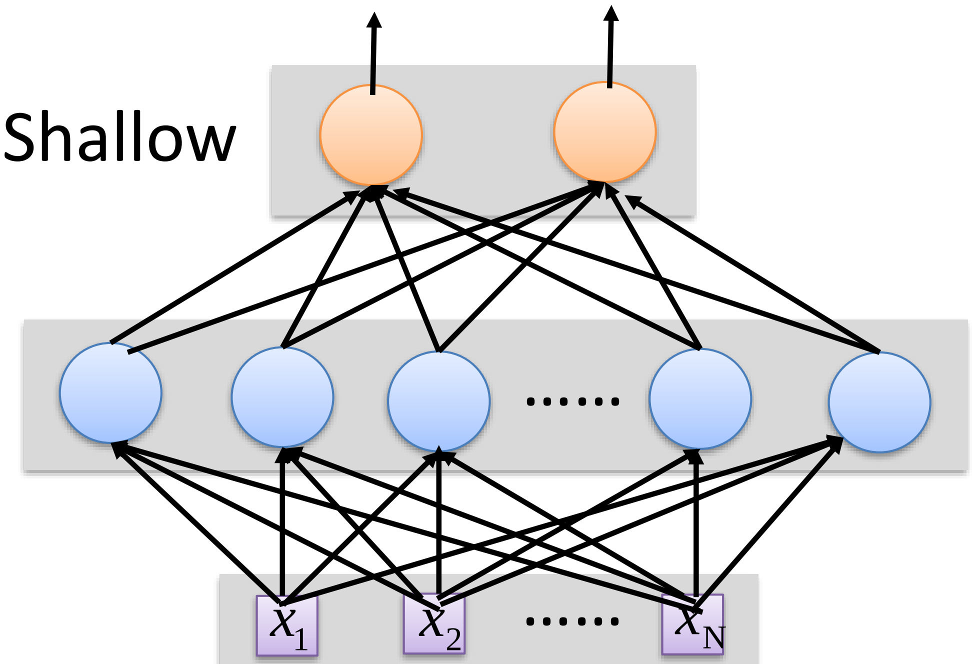
Ensemble Learning

- ***Ensemble learning*** is training multiple classifiers separately and combining their predictions
  - Ensemble learning often outperforms individual classifiers
  - Better results obtained with higher model variety in the ensemble
  - ***Bagging*** (**b**ootstrap **ag**gregating)
    - Randomly draw subsets from the training set (i.e., bootstrap samples)
    - Train separate classifiers on each subset of the training set
    - Perform classification based on the average vote of all classifiers
  - ***Boosting***
    - Train a classifier, and apply weights on the training set (apply higher weights on misclassified examples, focus on "hard examples")
    - Train new classifier, reweight training set according to prediction error
    - Repeat
    - Perform classification based on weighted vote of the classifiers

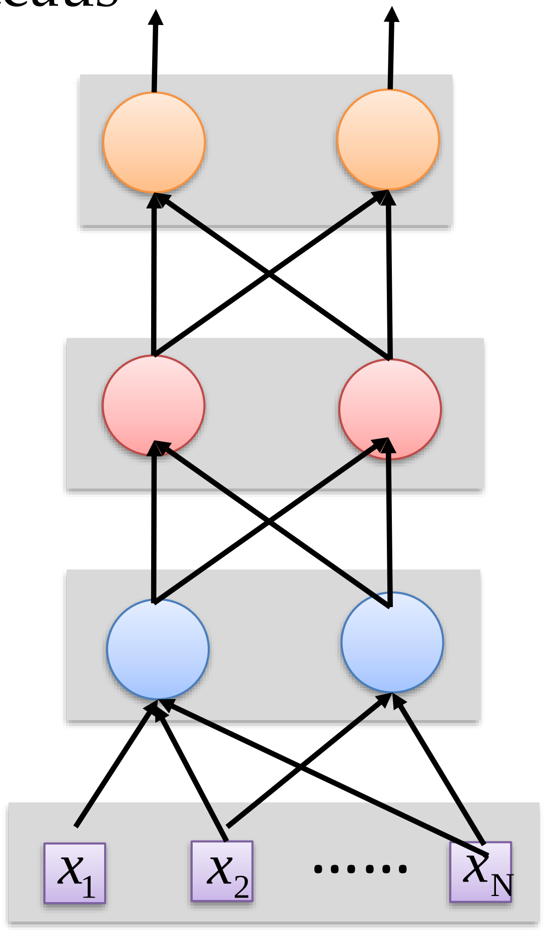
Deep vs Shallow Networks

- Deeper networks perform better than shallow networks
  - But only up to some limit: after a certain number of layers, the performance of deeper networks plateaus

Shallow

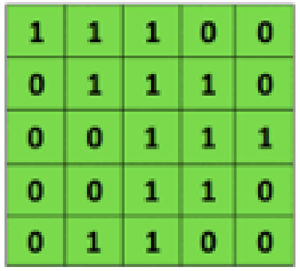


Deep

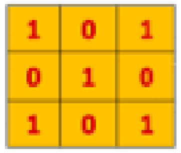


Convolutional Neural Networks (CNNs)

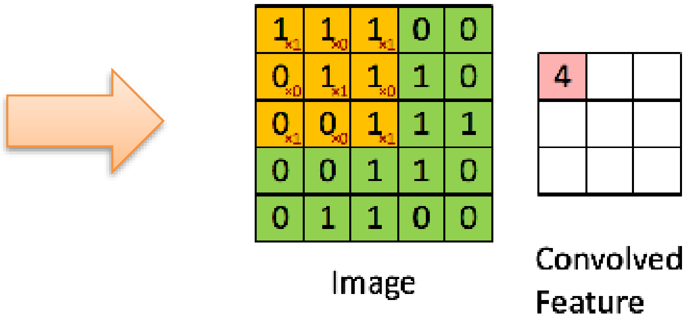
- *Convolutional neural networks* (CNNs) were primarily designed for image data
- CNNs use a convolutional operator for extracting data features
  - Allows parameter sharing
  - Efficient to train
  - Have less parameters than NNs with fully-connected layers
- CNNs are robust to spatial translations of objects in images
- A convolutional filter slides (i.e., convolves) across the image



Input matrix



Convolutional 3x3 filter



Picture from: http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

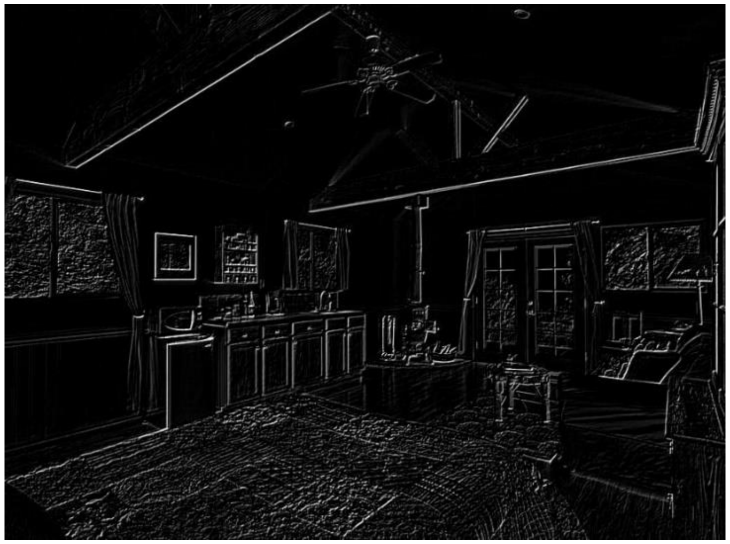
Convolutional Neural Networks (CNNs)

- When the convolutional filters are scanned over the image, they capture useful features
  - E.g., edge detection by convolutions

Filter

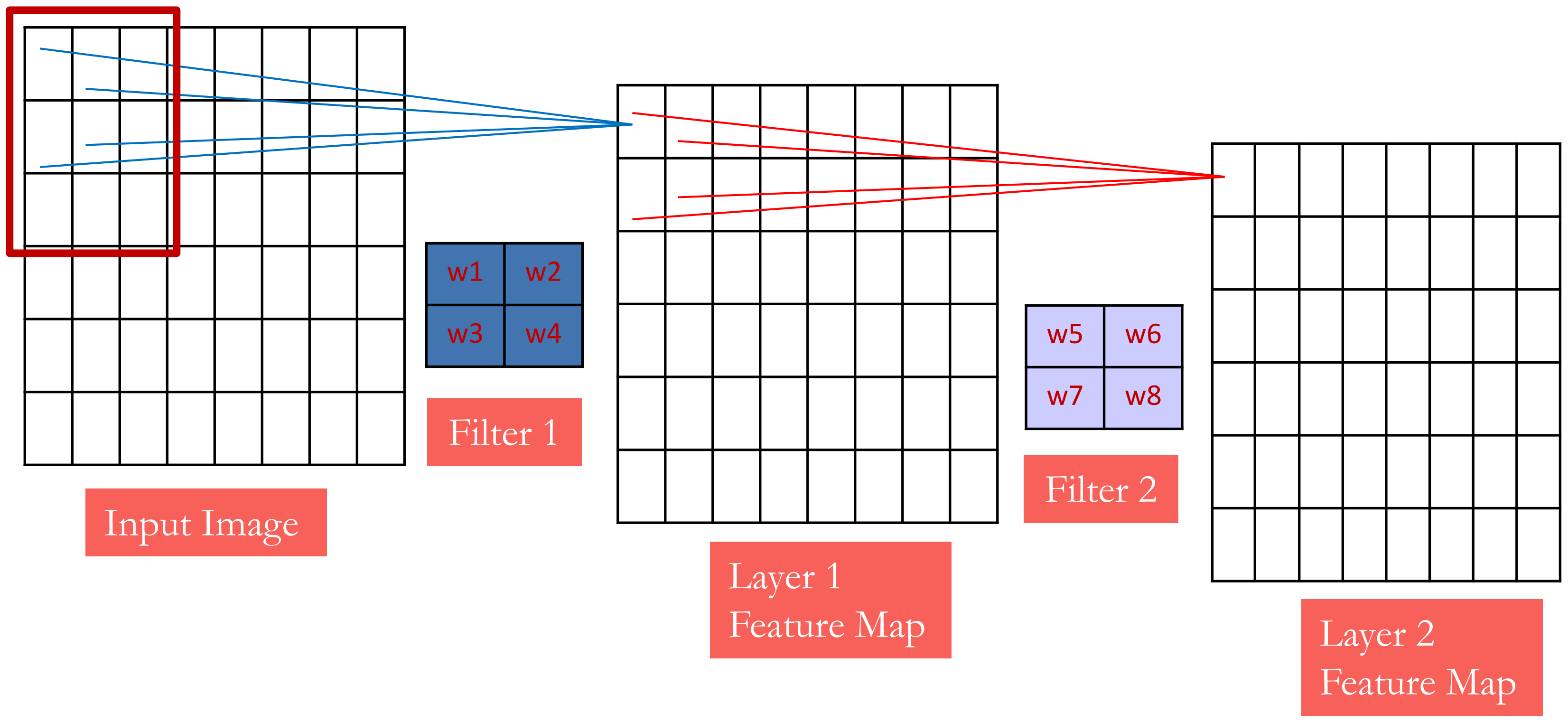
$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Input Image


Convoluted Image

Convolutional Neural Networks (CNNs)

- In CNNs, hidden units in a layer are only connected to a small region of the layer before it (called local **receptive field**)
  - The depth of each feature map corresponds to the number of convolutional filters used at each layer



Slide credit: Param Vir Singh – Deep Learning

Convolutional Neural Networks (CNNs)

- ***Max pooling***: reports the maximum output within a rectangular neighborhood
- ***Average pooling***: reports the average output of a rectangular neighborhood
- Pooling layers reduce the spatial size of the feature maps
  - Reduce the number of parameters, prevent overfitting

MaxPool with a 2×2 filter with stride of 2

1	3	5	3
4	2	3	1
3	1	1	3
0	1	0	4

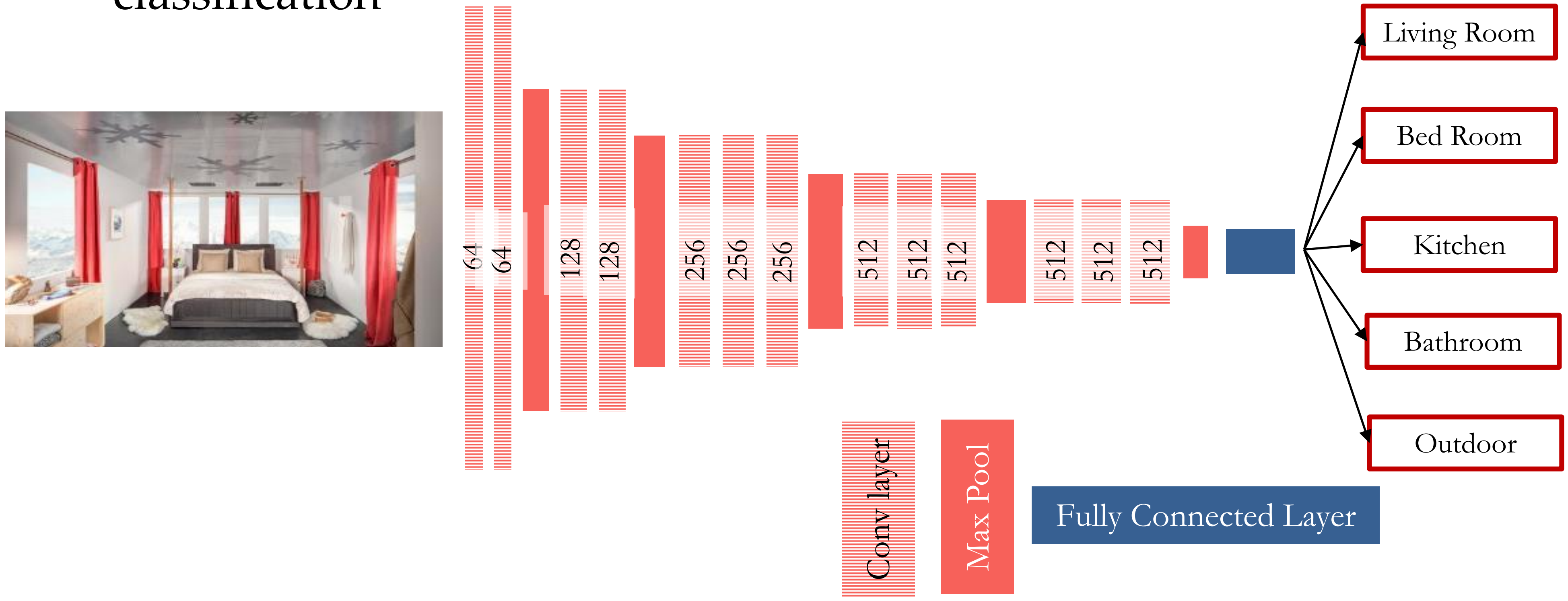
Input Matrix

4	5
3	4

Output Matrix

Convolutional Neural Networks (CNNs)

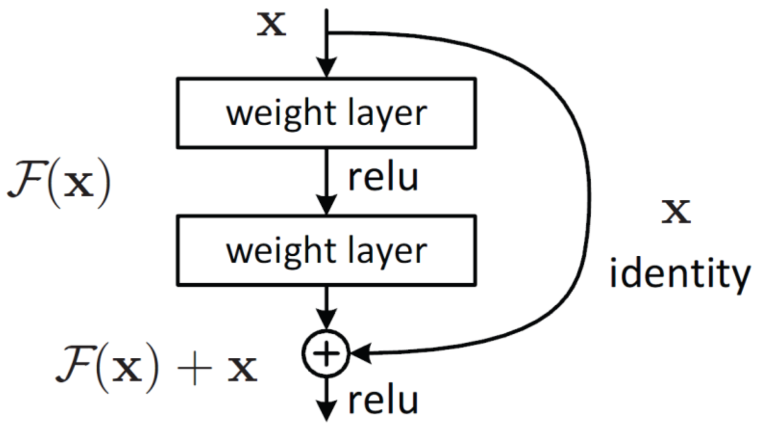
- Feature extraction architecture
  - After 2 convolutional layers, a max-pooling layer reduces the size of the feature maps (typically by 2)
  - A fully convolutional and a softmax layers are added last to perform classification



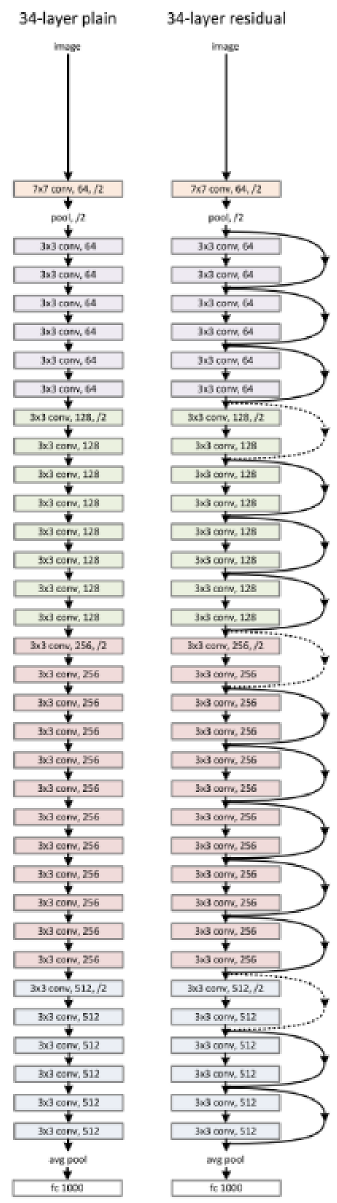
Slide credit: Param Vir Singh – Deep Learning

Residual CNNs

- *Residual networks* (ResNets)
  - Introduce “identity” skip connections
    - Layer inputs are propagated and added to the layer output
    - Mitigate the problem of vanishing gradients during training
    - Allow training very deep NN (with over 1,000 layers)
  - Several variants exist: 18, 34, 50, 101, 152, and 200 layers
  - Are used as base models of other state-of-the-art NNs
    - Other similar models: ResNeXT, DenseNet



x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

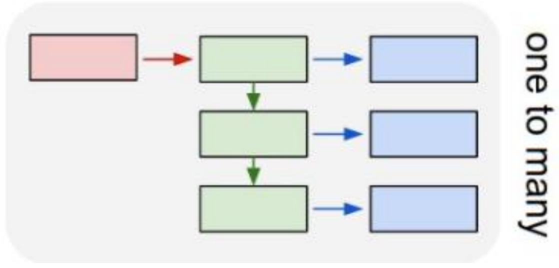
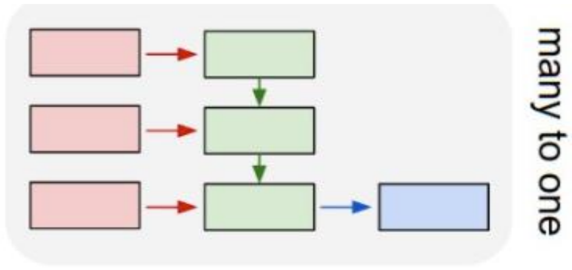
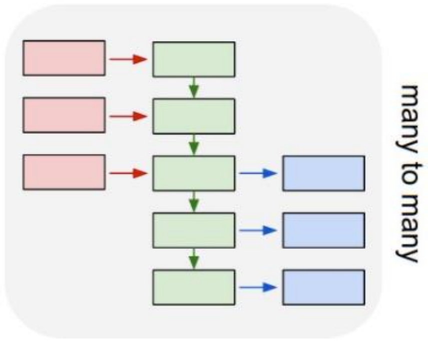


Recurrent Neural Networks (RNNs)

- *Recurrent NNs* are used for modeling sequential data and data with varying length of inputs and outputs
  - Videos, text, speech, DNA sequences, human skeletal data
- RNNs introduce recurrent connections between the neurons units
  - This allows processing sequential data one element at a time by selectively passing information across a sequence
  - Memory of the previous inputs is stored in the model's internal state and affect the model predictions
  - Can capture correlations in sequential data
- RNNs use backpropagation-through-time for training
- RNNs are more sensitive to the vanishing gradient problem

Recurrent Neural Networks (RNNs)

- RNNs can have one of many inputs and one of many outputs

RNN	Application	Input	Output
	Image Captioning		A person riding a motorbike on dirt road
	Sentiment Analysis	Awesome movie. Highly recommended.	Positive
	Machine Translation	Happy Diwali	शुभ दीपावली

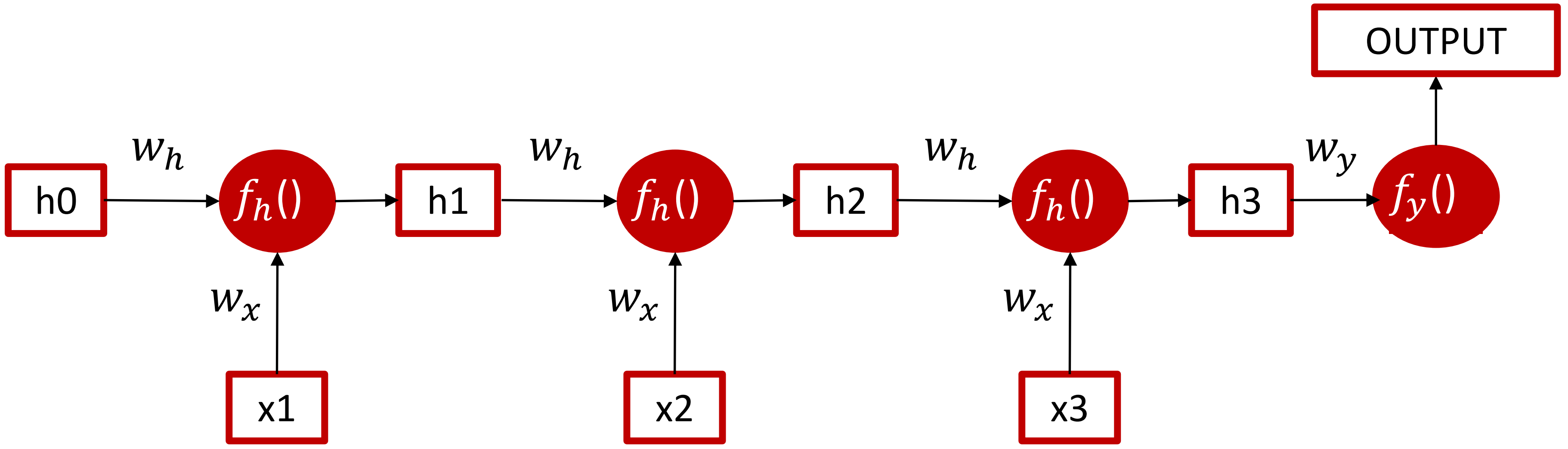
Slide credit: Param Vir SIngh– Deep Learning

Recurrent Neural Networks (RNNs)

- RNN use same set of weights w_h and biases w_x across all time steps

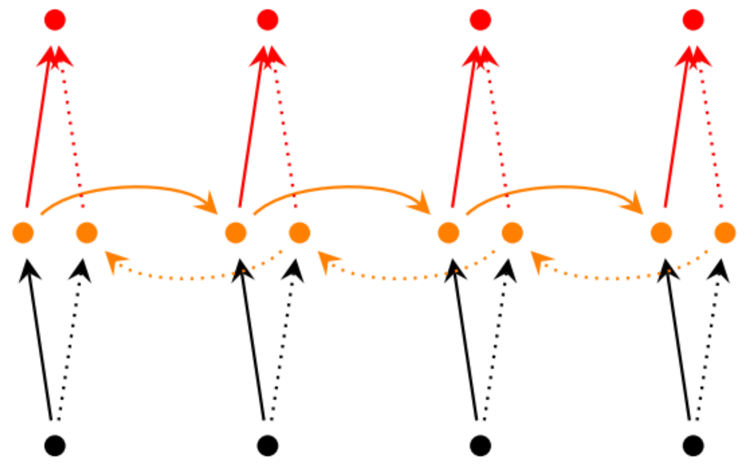
$$h(t) = f_h\big(w_h * h(t-1) + w_x * x(t)\big)$$

- An RNN is shown rolled over time



Bidirectional RNNs

- ***Bidirectional RNNs*** incorporate both forward and backward passes through sequential data
  - The output may not only depend on the previous elements in the sequence, but also on future elements in the sequence
  - It resembles two RNNs stacked on top of each other



$$\overrightarrow{h}_t = \sigma(\overrightarrow{W}^{(hh)}\overrightarrow{h}_{t-1} + \overrightarrow{W}^{(hx)}x_t)$$

$$\overleftarrow{h}_t = \sigma(\overleftarrow{W}^{(hh)}\overleftarrow{h}_{t+1} + \overleftarrow{W}^{(hx)}x_t)$$

$$y_t = f(\,[\overrightarrow{h}_t; \overleftarrow{h}_t\,]\,)$$

Output both past and future elements

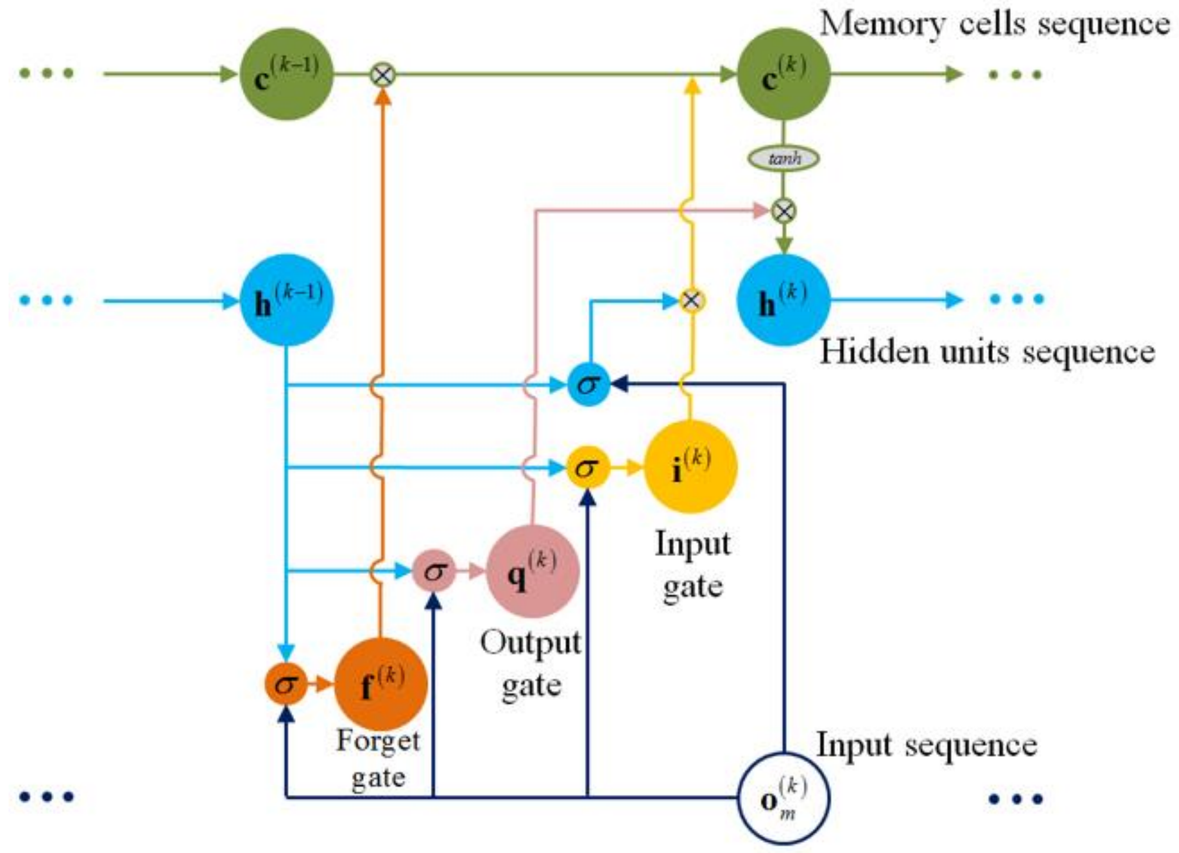
Slide credit: Param Vir Singh – Deep Learning

LSTM Networks

- *Long Short Term Memory (LSTM)* networks are a variant of RNNs
- LSTM mitigates the vanishing/exploding gradient problem
  - Solution: a Memory Cell, updated at each step in the sequence
- Three gates control the flow of information to and from the Memory Cell
  - Input Gate: protects the current step from irrelevant inputs
  - Output Gate: prevents current step from passing irrelevant information to later steps
  - Forget Gate: limits information passed from one cell to the next
- Most modern RNN models use either LSTM units or other more advanced types of recurrent units (e.g., GRU units)

LSTM Networks

- LSTM cell
  - Input gate, output gate, forget gate, memory cell
  - LSTM can learn long-term correlations within the data sequences



$$\mathbf{i}^{(k)} = \sigma\left(\mathbf{W}_{oi}\mathbf{o}_m^{(k)} + \mathbf{W}_{hi}\mathbf{h}^{(k-1)} + \mathbf{b}_i\right)$$

$$\mathbf{f}^{(k)} = \sigma\left(\mathbf{W}_{of}\mathbf{o}_m^{(k)} + \mathbf{W}_{hf}\mathbf{h}^{(k-1)} + \mathbf{b}_f\right)$$

$$\mathbf{q}^{(k)} = \sigma\left(\mathbf{W}_{oq}\mathbf{o}_m^{(k)} + \mathbf{W}_{hq}\mathbf{h}^{(k-1)} + \mathbf{b}_q\right)$$

$$\mathbf{c}^{(k)} = \mathbf{f}^{(k)}\mathbf{c}^{(k-1)} + \mathbf{i}^{(k)}\,\sigma\left(\mathbf{W}_{oc}\mathbf{o}_m^{(k)} + \mathbf{W}_{hc}\mathbf{h}^{(k-1)} + \mathbf{b}_c\right)$$

$$\mathbf{h}^{(k)} = \mathbf{q}^{(k)}\,tanh\left(\mathbf{c}^{(k)}\right)$$

References

1. Hung-yi Lee – Deep Learning Tutorial
2. Ismini Lourentzou – Introduction to Deep Learning
3. CS231n Convolutional Neural Networks for Visual Recognition (Stanford CS course) (link)
4. James Hays, Brown – Machine Learning Overview
5. Param Vir Singh, Shunyuan Zhang, Nikhil Malik – Deep Learning
6. Sebastian Ruder – An Overview of Gradient Descent Optimization Algorithms (link)