University of Idaho
Department of Computer Science

CS 487/587 Adversarial Machine Learning

Dr. Alex Vakanski

Lecture 2

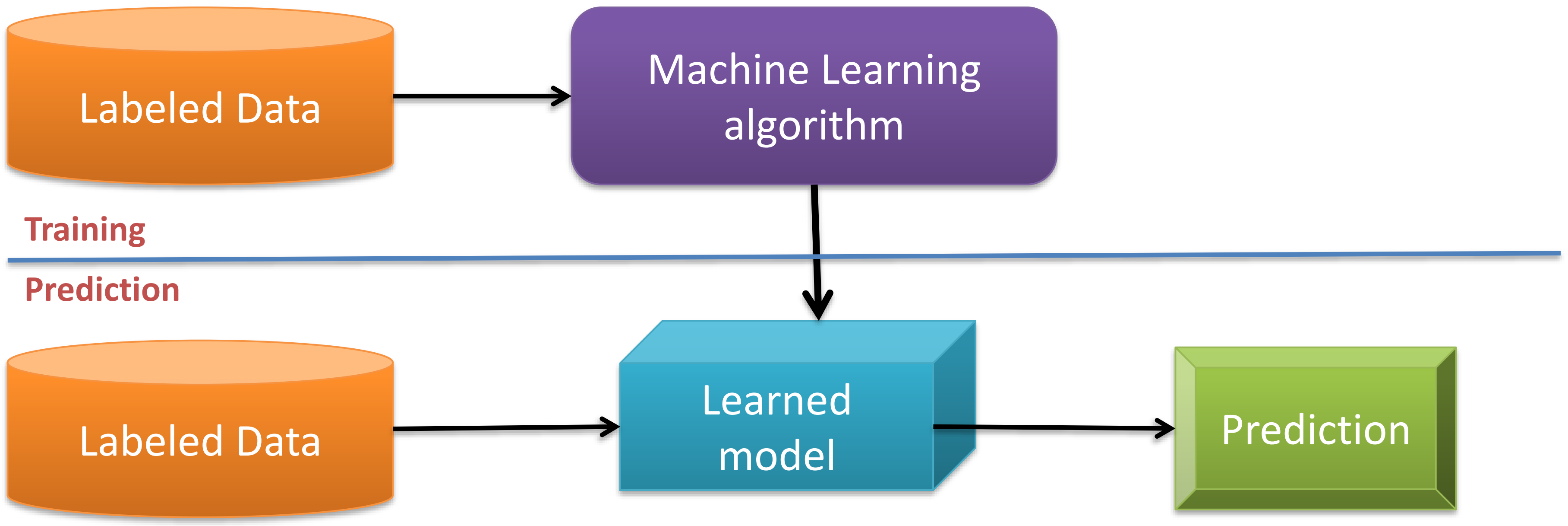
Deep Learning Overview

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, Transformer Networks

Machine Learning Basics

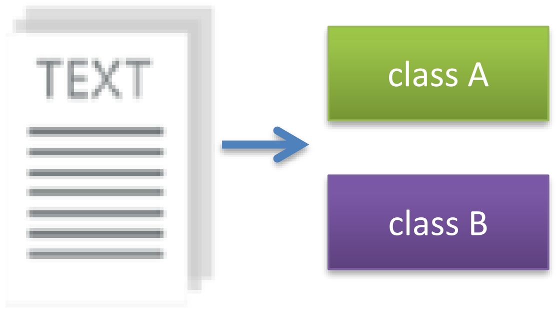
- *Artificial Intelligence* is a scientific field concerned with the development of algorithms that allow computers to learn without being explicitly programmed
- *Machine Learning* is a branch of Artificial Intelligence, which focuses on methods that learn from data and make predictions on unseen data



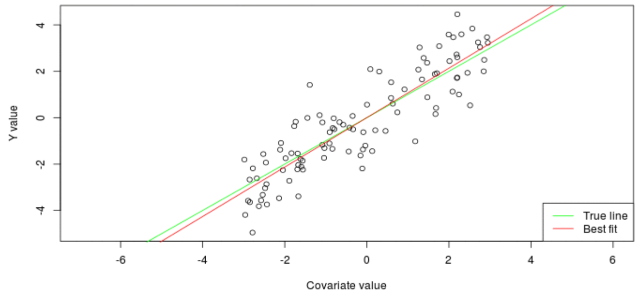
Machine Learning Types

Machine Learning Basics

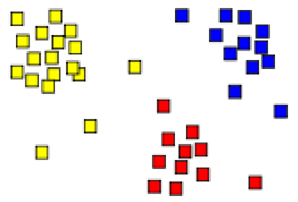
- *Supervised*: learning with labeled data
  - Example: image classification, email classification
  - Example: regression for predicting real-valued outputs
- *Unsupervised*: discover patterns in unlabeled data
  - Example: cluster similar data points
- Subcategories: **semi-supervised**, **self-supervised**,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



Classification



Regression



Clustering

Supervised Learning

- *Supervised learning* categories and techniques
  - **Numerical classifier functions**
    - Linear classifier, perceptron, logistic regression, support vector machines (SVM), neural networks
  - **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, XGBoost), random forest

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

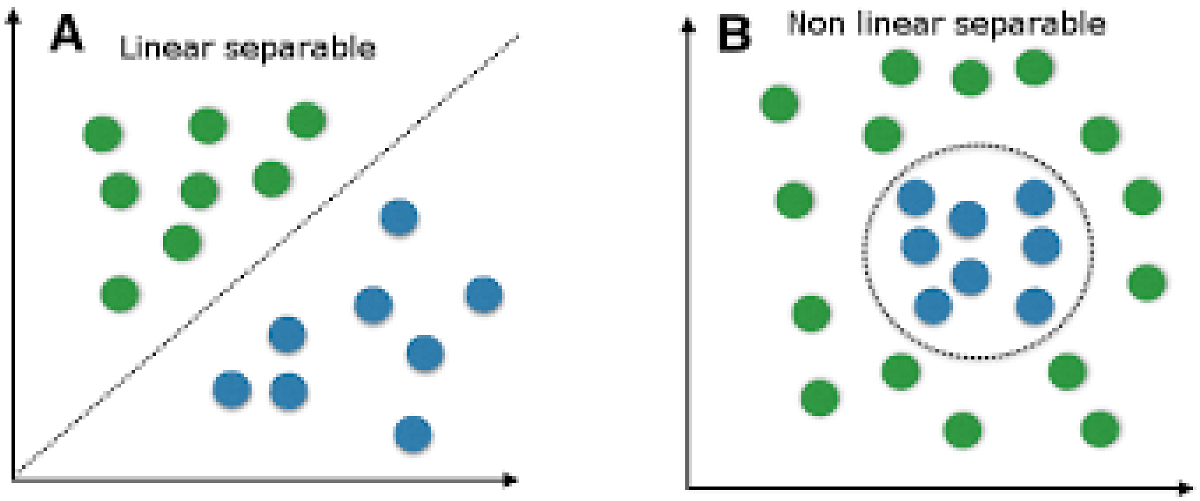
Linear vs Non-linear Techniques

- 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

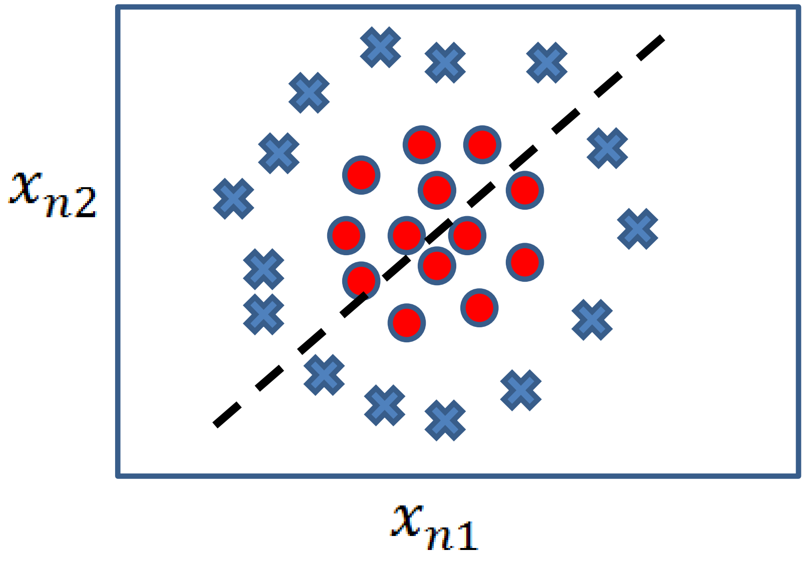
Linear vs Non-linear Techniques

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

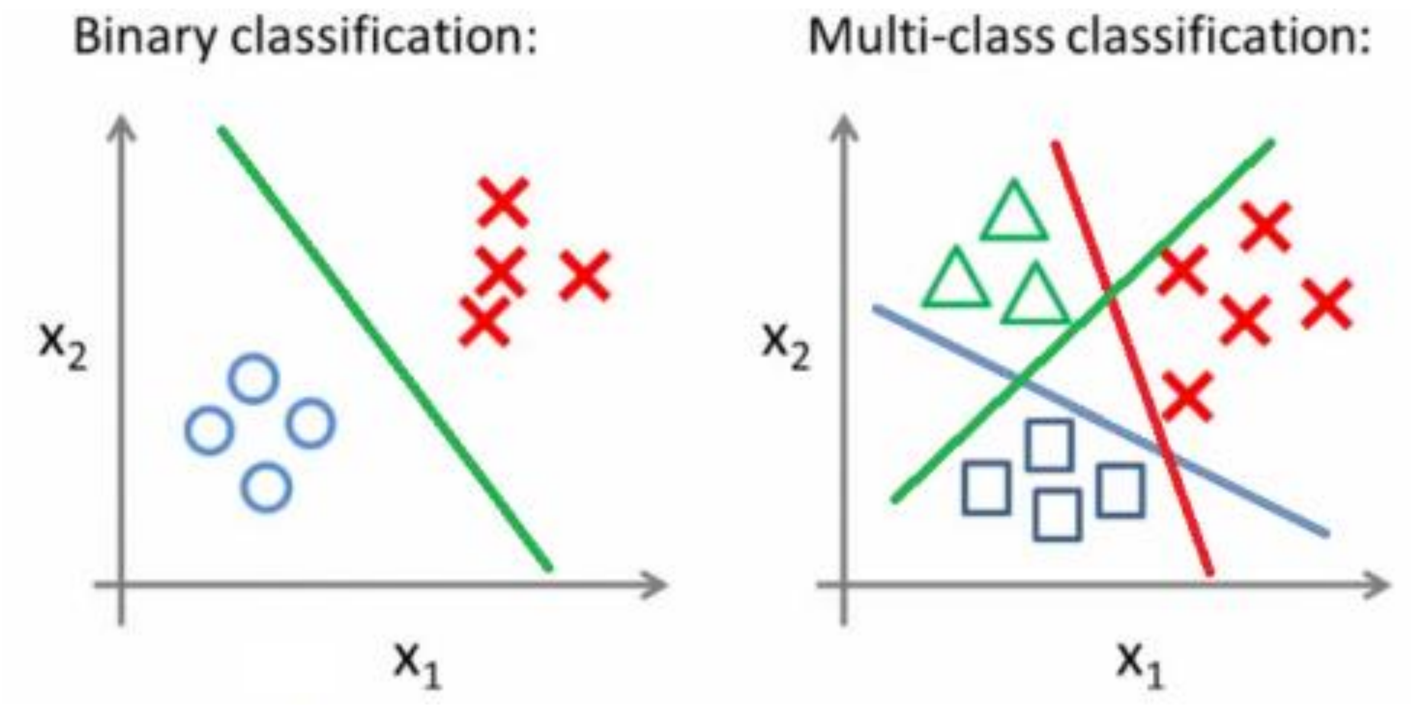


Picture from: Y-Fan Chang – An Overview of Machine Learning

Binary vs Multi-class Classification

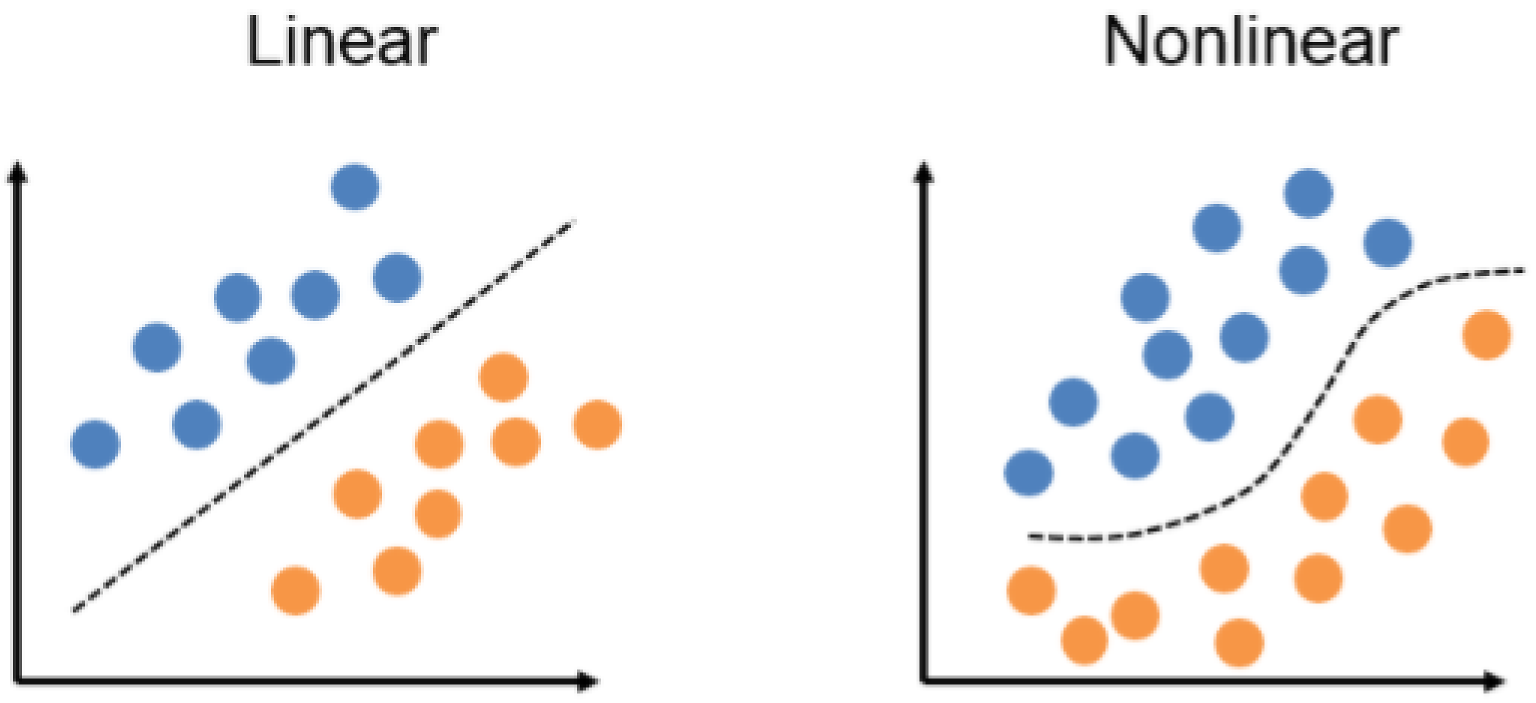
Binary vs Multi-class Classification

- A classification 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

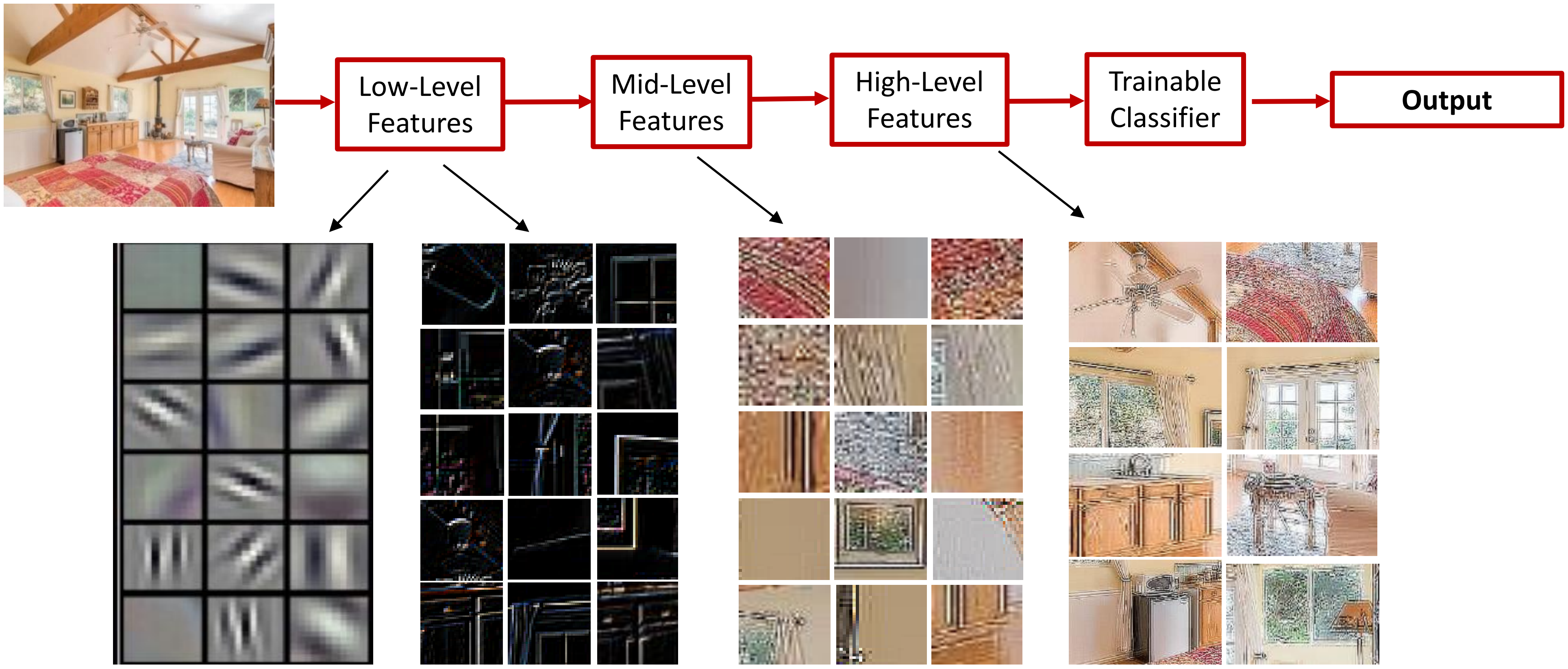
- Both the binary and multi-class classification problems can be linearly or non-linearly separated
  - Figure: linearly and non-linearly separated data for binary classification problem



Deep Learning

Introduction to Deep Learning

- ***Deep learning*** (DL) is a machine learning subfield that uses multi-layer Neural Networks for learning data representations
  - The layers in NN learn hierarchical features at different levels of abstraction
  - Input image pixels → Edges → Textures → Parts → Objects

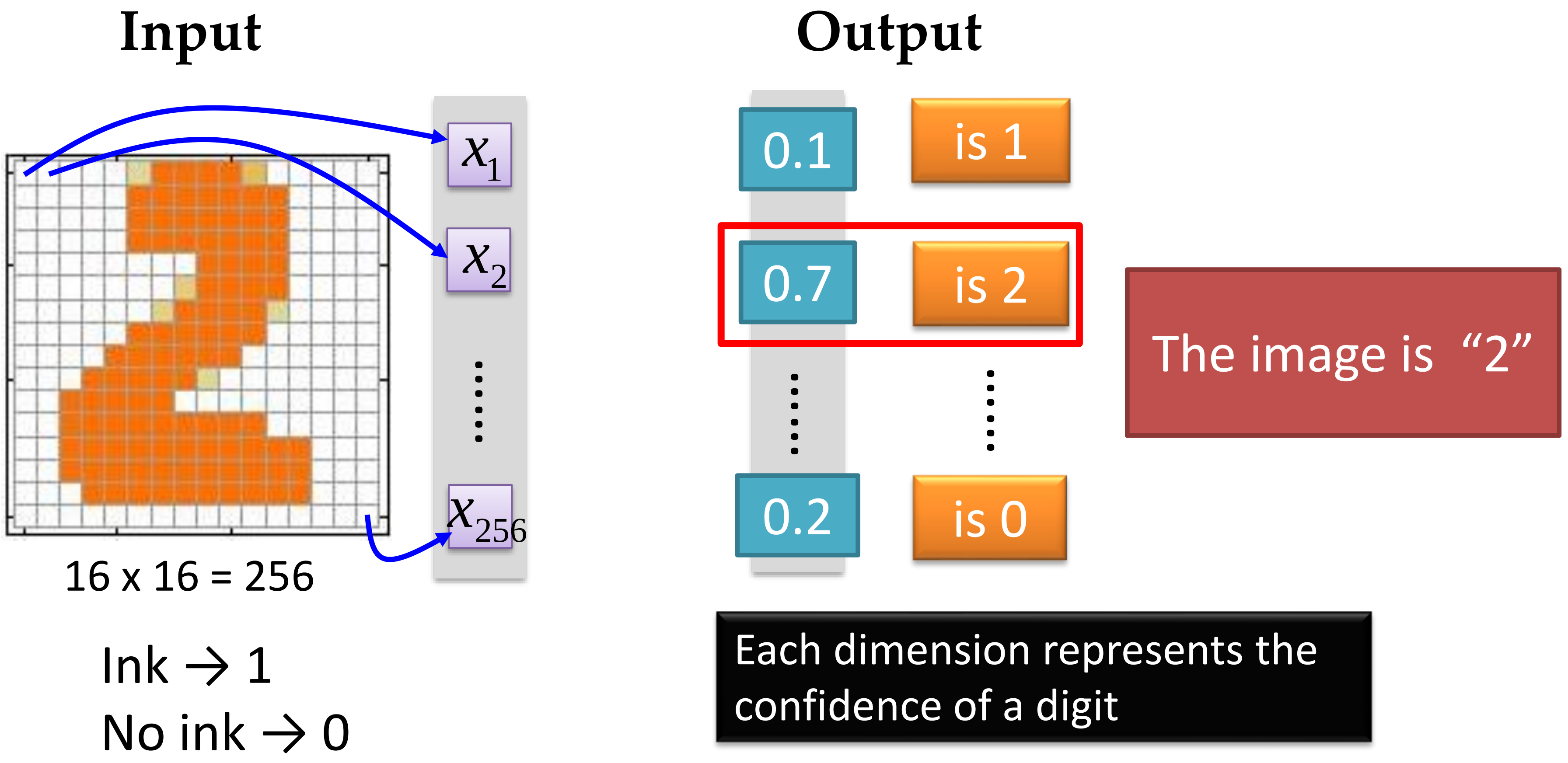


Why is DL Useful?

- DL provides a flexible, learnable framework for representing visual, text, linguistic information
  - Can learn in supervised and unsupervised manner
- DL represents an effective end-to-end learning system
- Requires large amounts of training data
- Since about 2012, DL has outperformed other ML techniques
  - First in vision and speech, then NLP, and other applications

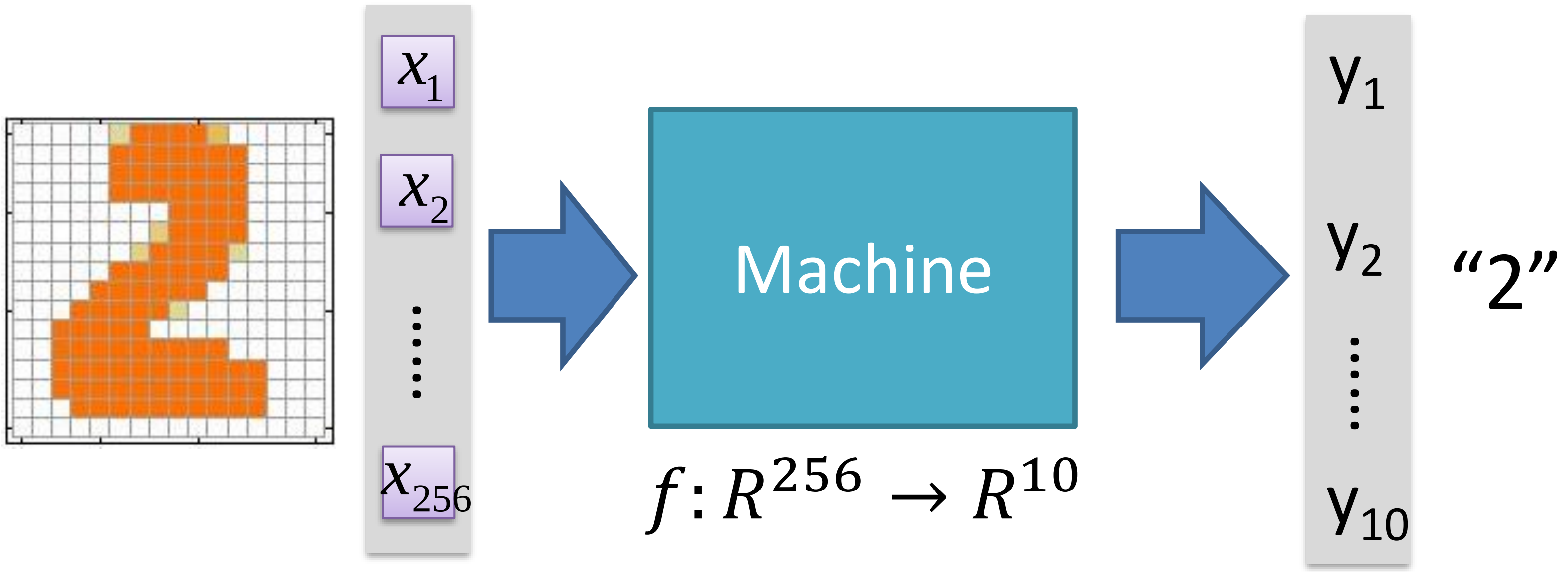
Introduction to Neural Networks

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



Introduction to Neural Networks

- Handwritten digit recognition



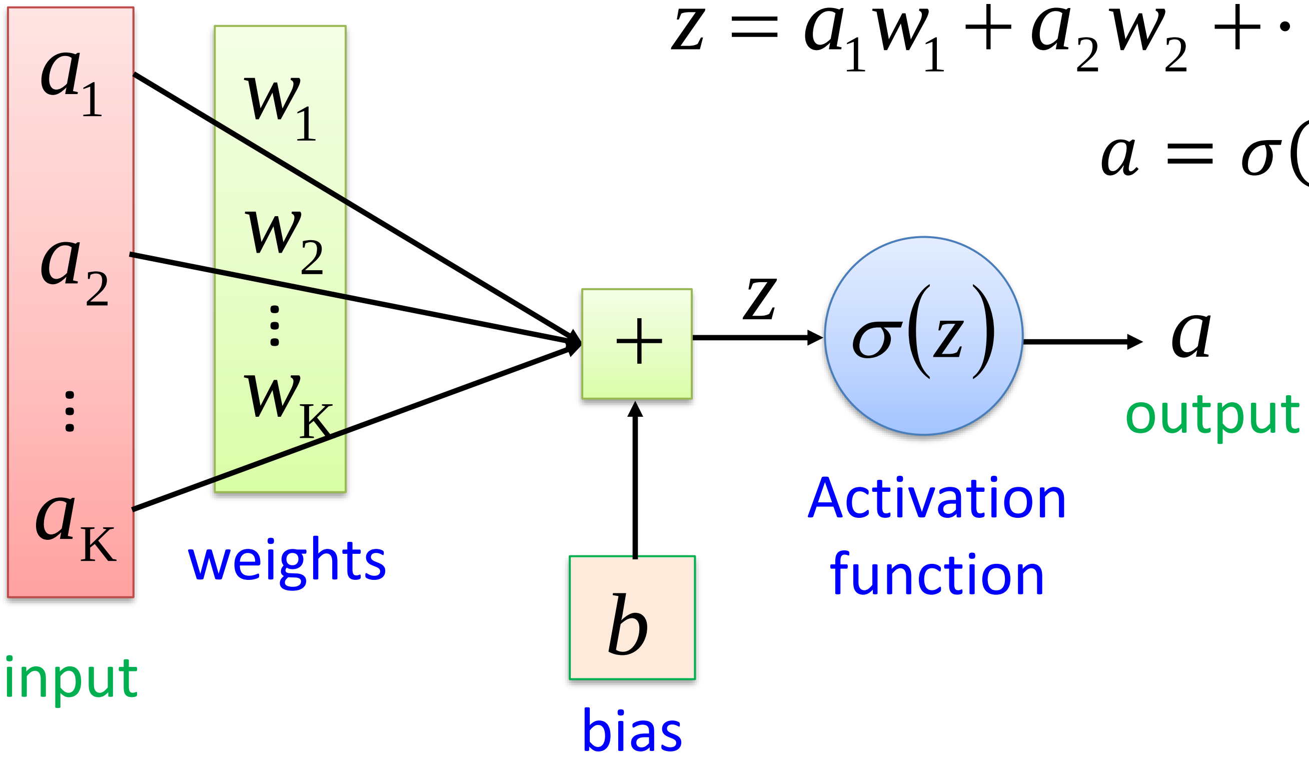
The function f is represented by a neural network

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 $f: R^K \to R$

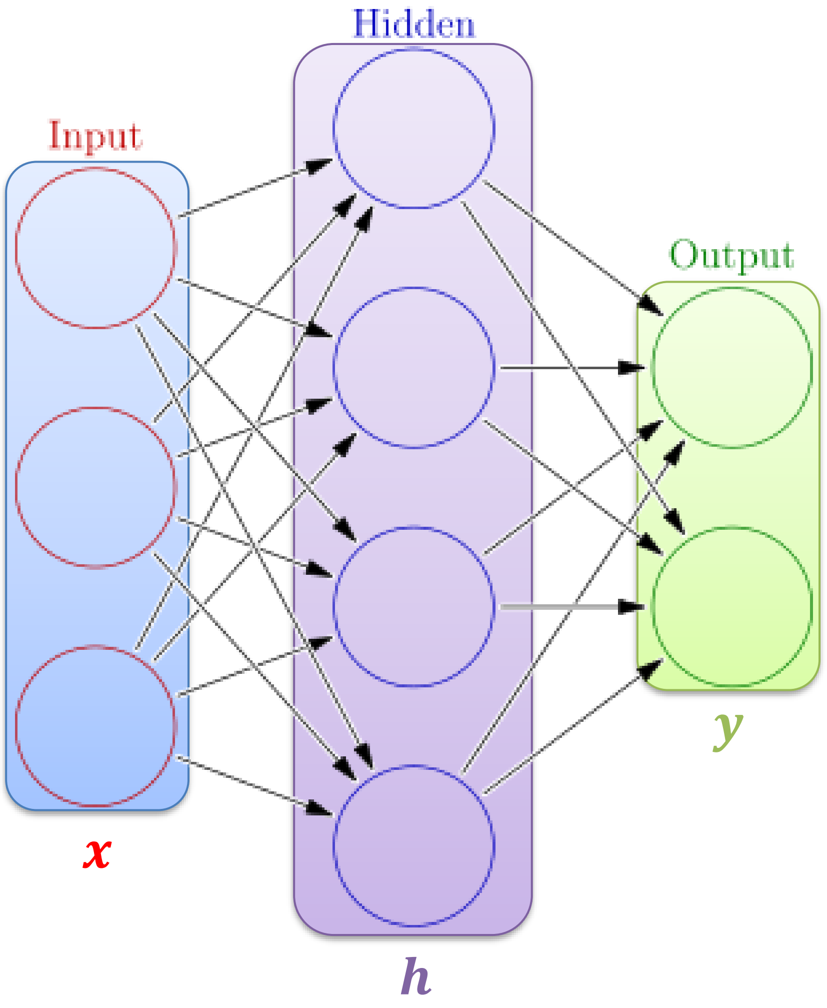
$$z = a_1 w_1 + a_2 w_2 + \cdots + a_K w_K + b$$

$$a = \sigma(z)$$



Elements of Neural Networks

- A NN with one hidden layer and one output layer



Weights Biases

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

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

Activation functions

4 + 2 = 6 neurons (not counting inputs)

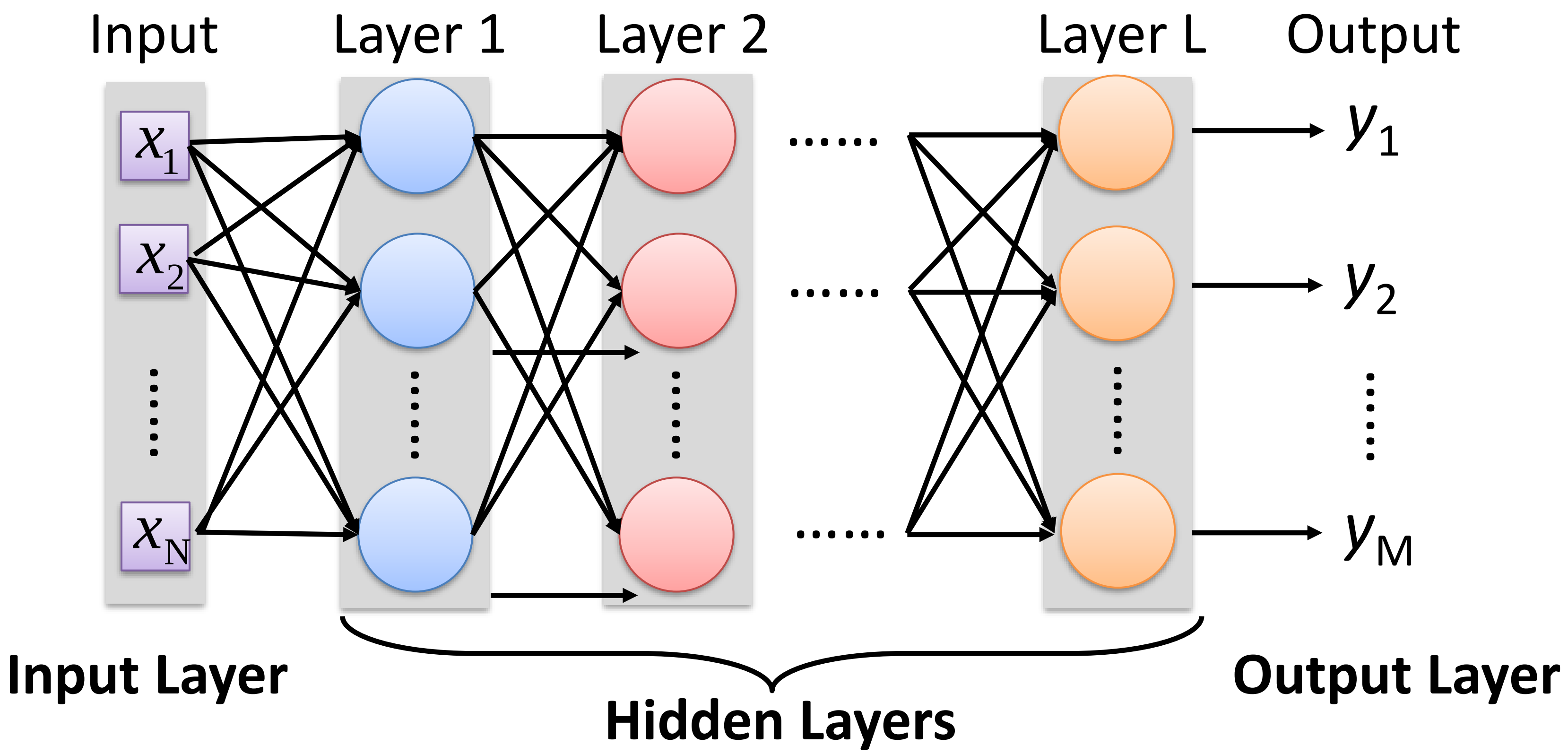
[3 × 4] + [4 × 2] = 20 weights

4 + 2 = 6 biases

26 learnable parameters

Elements of Neural Networks

- Deep NNs have many hidden layers
  - In the shown architecture, each neuron is connected to all neurons in the succeeding layer
  - Fully-connected (dense, linear) layers, a.k.a. Multi-Layer Perceptron (MLP)

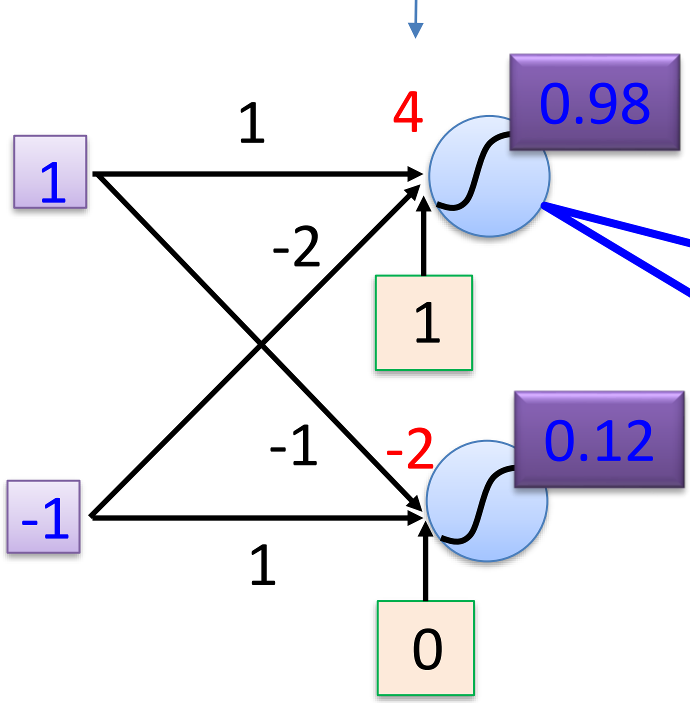


Elements of Neural Networks

Introduction to Neural Networks

- A simple network, toy example

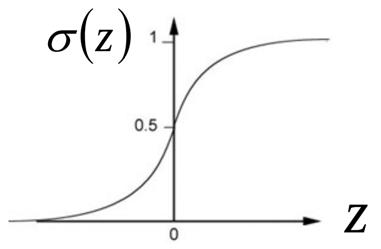
$\boxed{(1 \cdot 1) + (-1) \cdot (-2) + 1 = 4}$



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

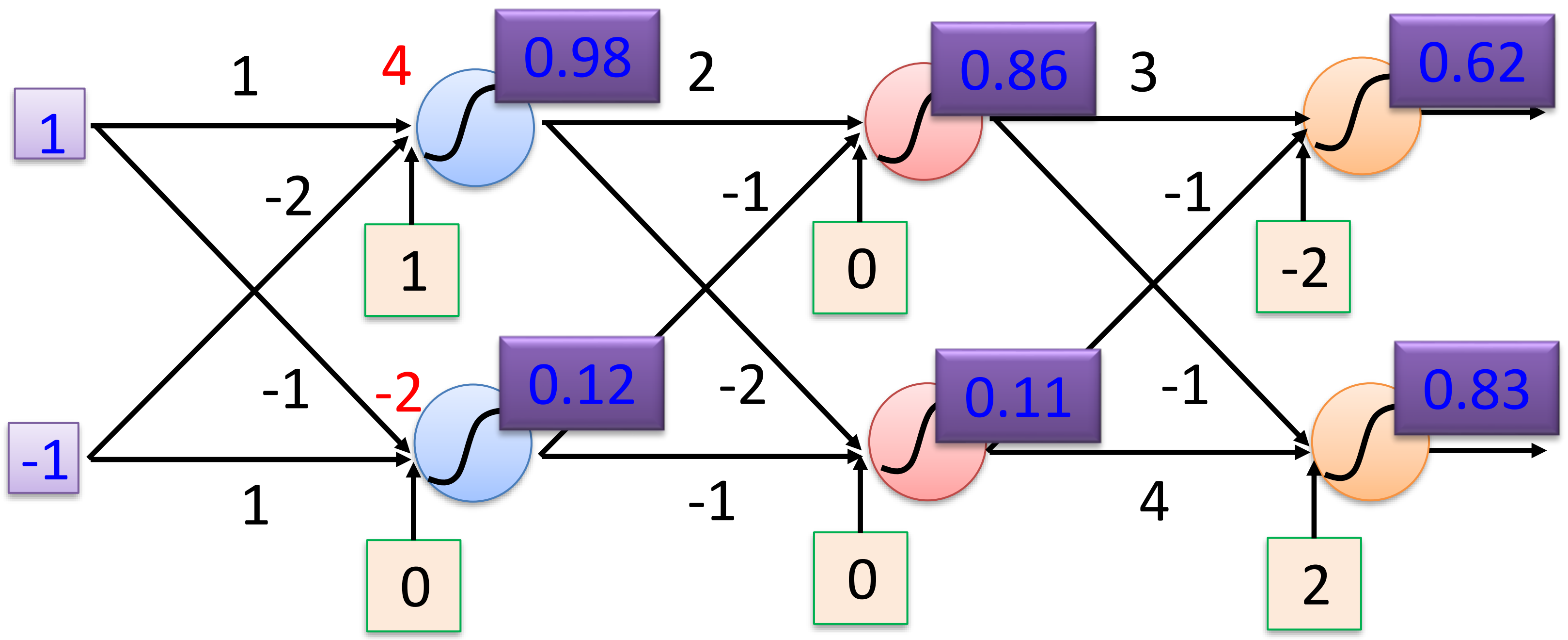
Sigmoid Function

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



Elements of Neural Networks

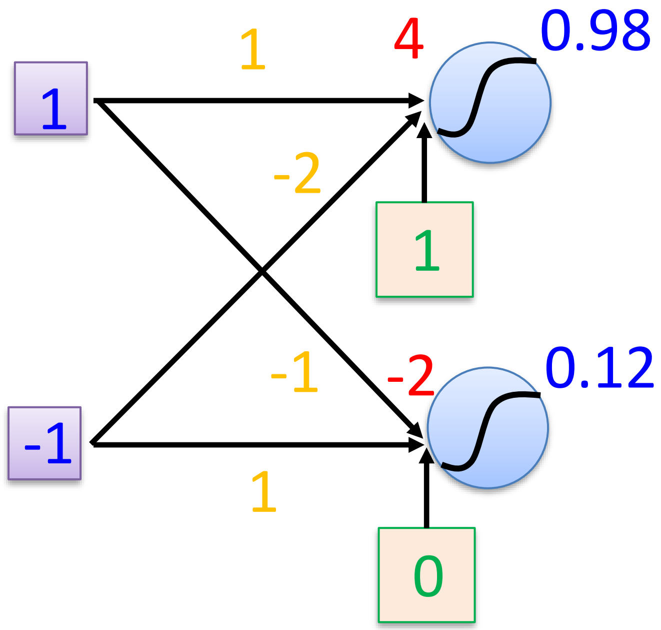
- A simple network, toy example (cont'd)
  - For an input vector $[1 \quad -1]^T$, the output is $[0.62 \quad 0.83]^T$



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

Matrix Operation

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

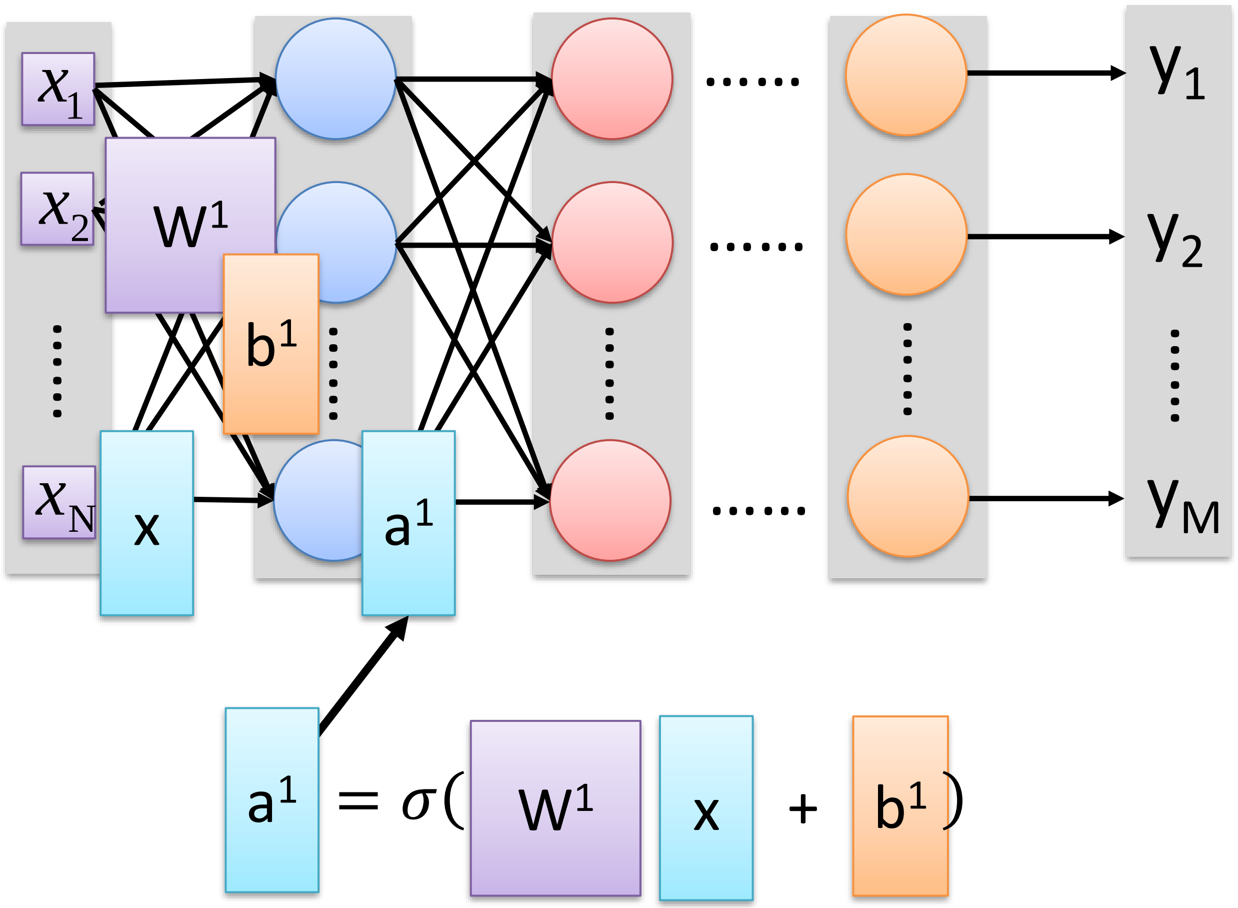


$$\sigma(\boxed{W}\ \boxed{x} + \boxed{b}) = \boxed{a}$$

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

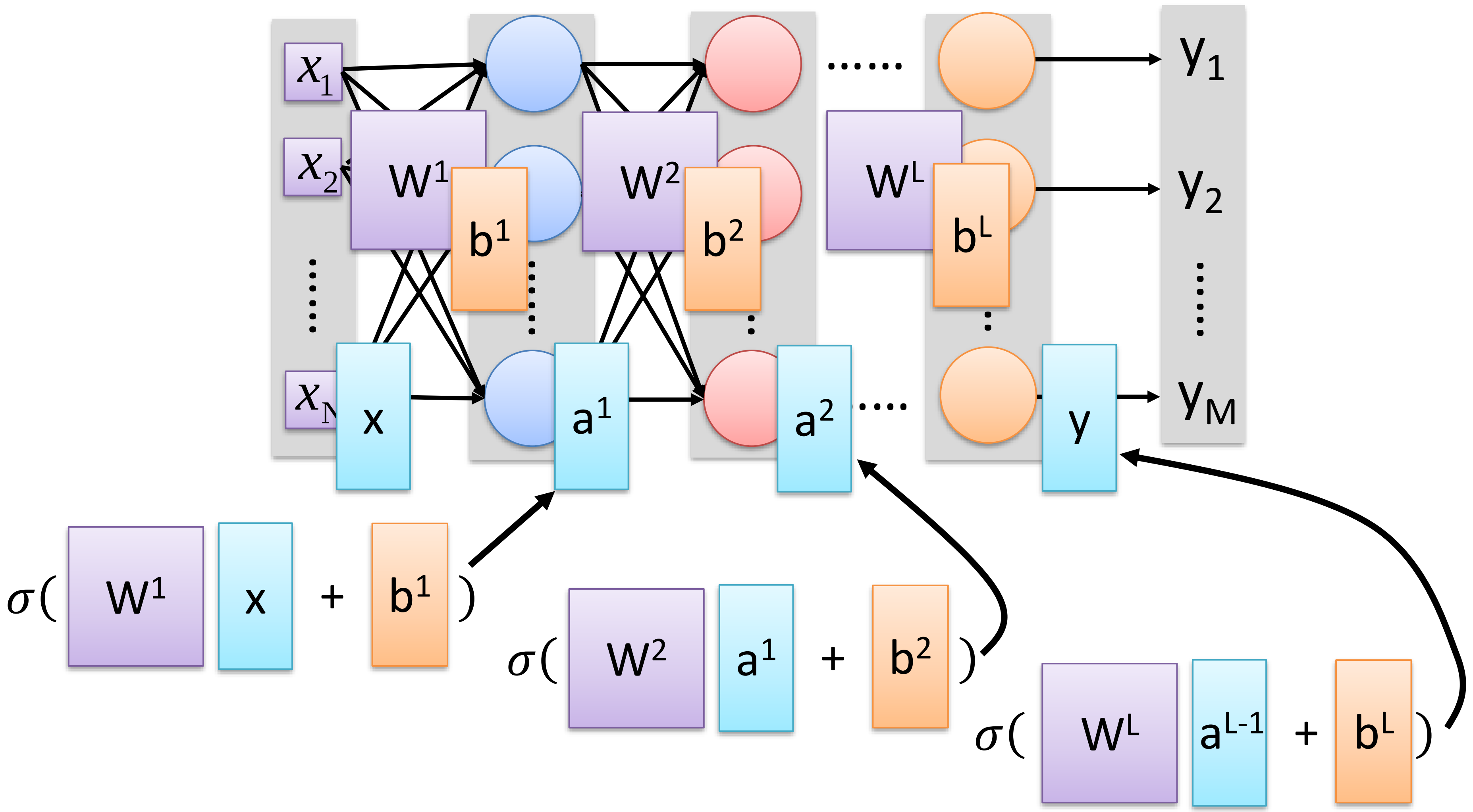
Matrix Operation

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



Matrix Operation

- Multilayer NN, matrix calculations for all layers

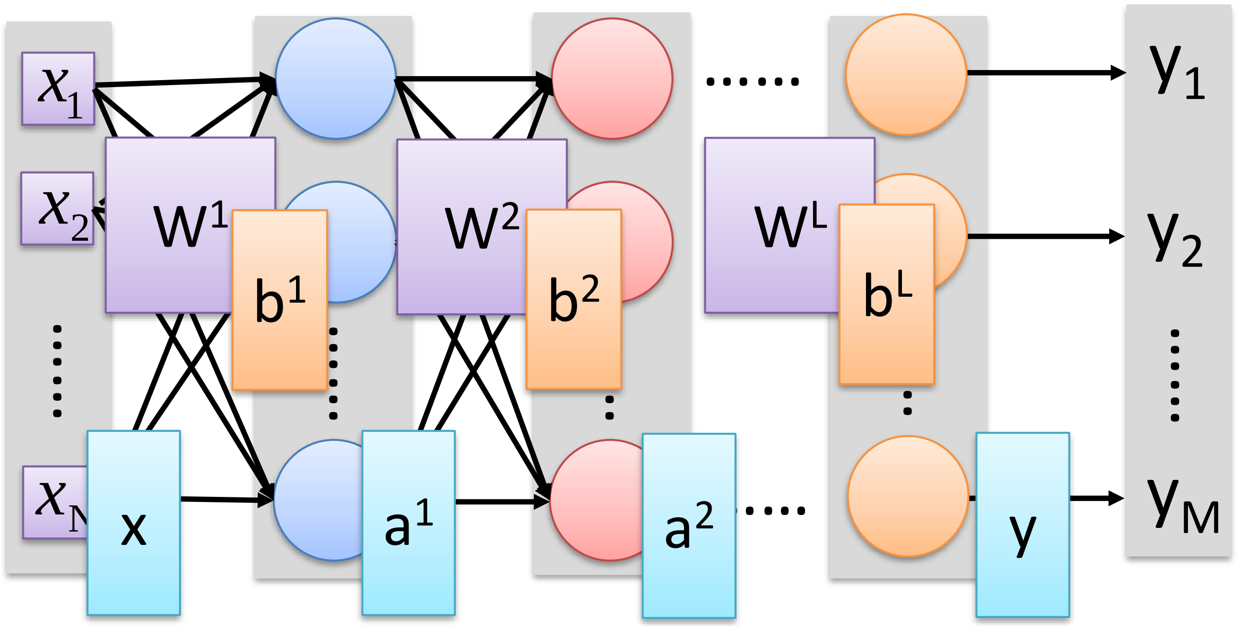


x_1
x_2
x_N
W^1
b^1
W^2
b^2
W^L
b^L
x
a^1
a^2
y
y_1
y_2
y_M
$\sigma($ W^1 x + b^1 $)$
$\sigma($ W^2 a^1 + b^2 $)$
$\sigma($ W^L a^{L-1} + b^L $)$

Matrix Operation

Introduction to Neural Networks

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

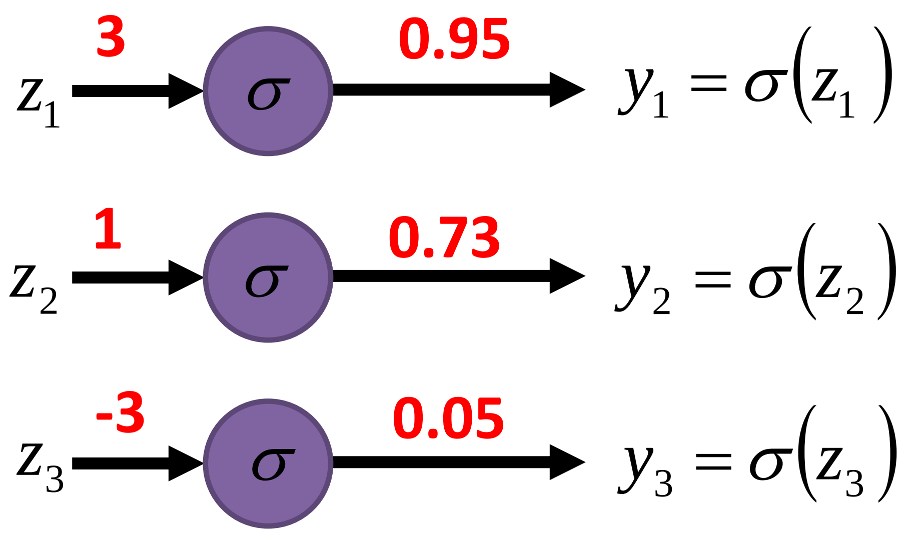


$$\boxed{y} = f\left(\boxed{x}\right) = \sigma\left(\boxed{W^L} \cdots \sigma\left(\boxed{W^2}\,\sigma\left(\boxed{W^1}\,\boxed{x} + \boxed{b^1}\right) + \boxed{b^2}\right) \cdots + \boxed{b^L}\right)$$

Softmax Layer

- In multi-class classification tasks, the output layer is typically a ***softmax layer***
  - I.e., it employs a ***softmax activation function***
  - If a layer with a sigmoid activation function is used as the output layer instead, the predictions by the NN may not be easy to interpret
    - Note that an output layer with sigmoid activations can still be used for binary classification

A Layer with Sigmoid Activations



Softmax Layer

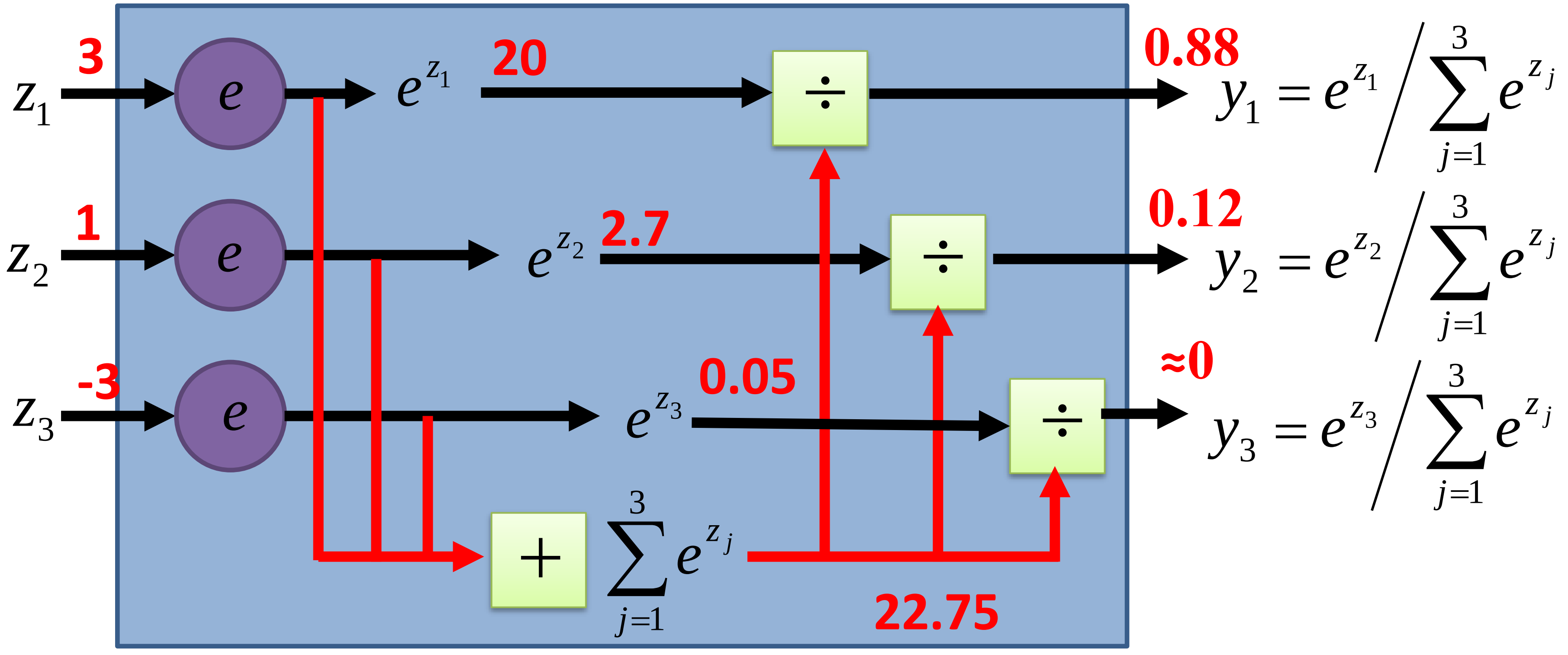
Introduction to Neural Networks

- 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



$$y_1 = e^{z_1} \Big/ \sum_{j=1}^{3} e^{z_j}$$

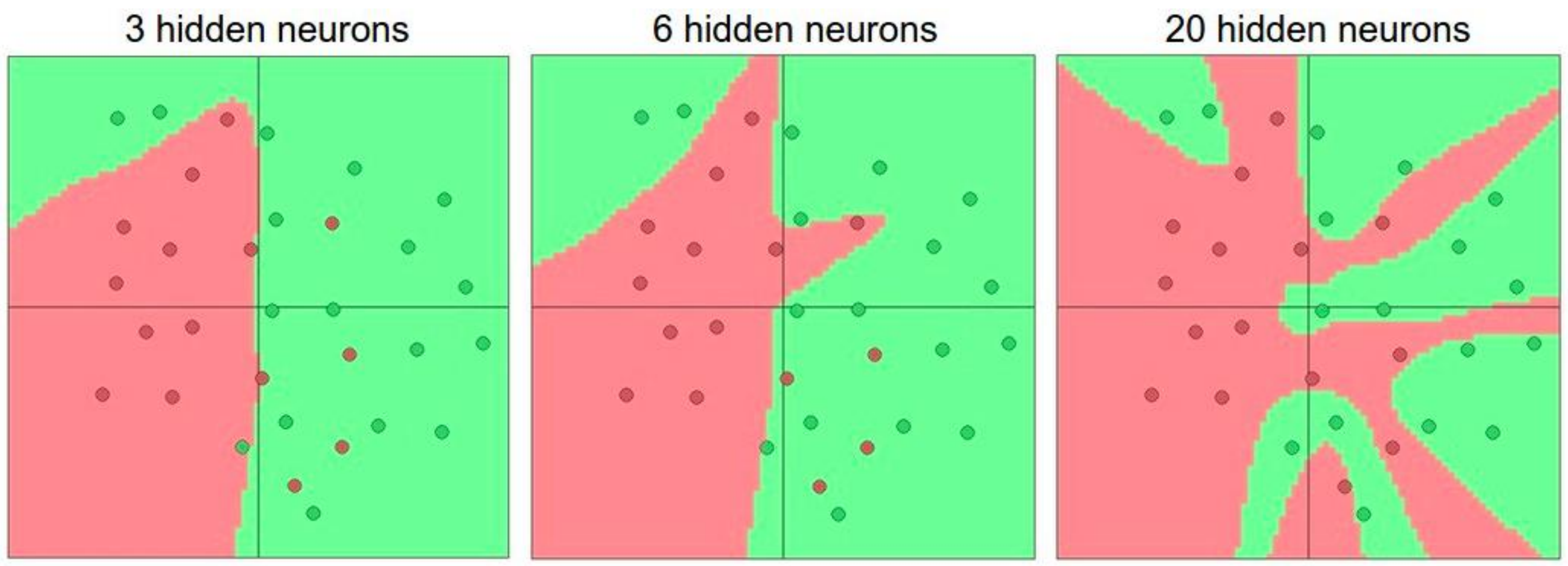
$$y_2 = e^{z_2} \Big/ \sum_{j=1}^{3} e^{z_j}$$

$$y_3 = e^{z_3} \Big/ \sum_{j=1}^{3} e^{z_j}$$

Activation Functions

Introduction to Neural Networks

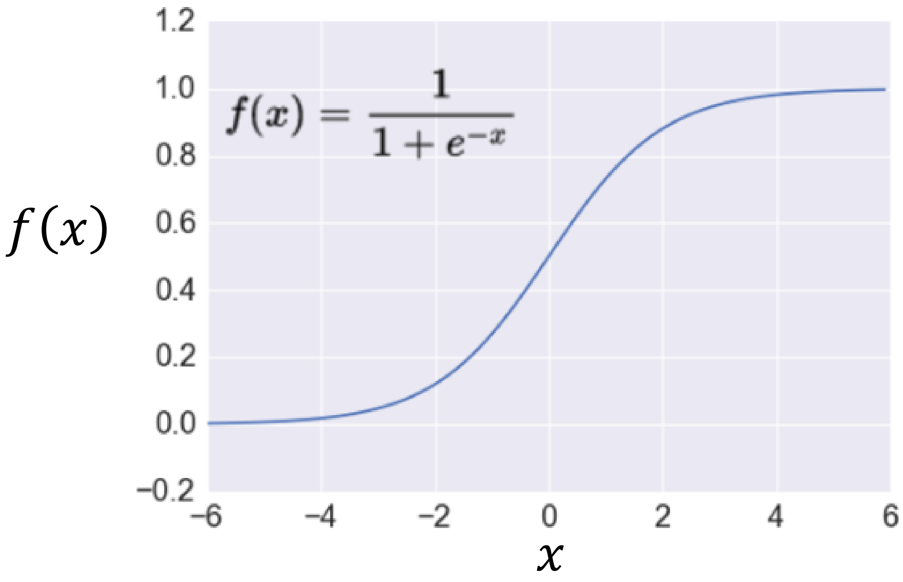
- Non-linear activations are needed to learn complex (non-linear) data representations
  - Otherwise, NNs would be just a linear function (such as $W_1 W_2 x = Wx$)
  - NNs with large number of layers (and neurons) can approximate more complex functions
    - Figure: more neurons improve representation (but, may overfit)



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

Activation: Sigmoid

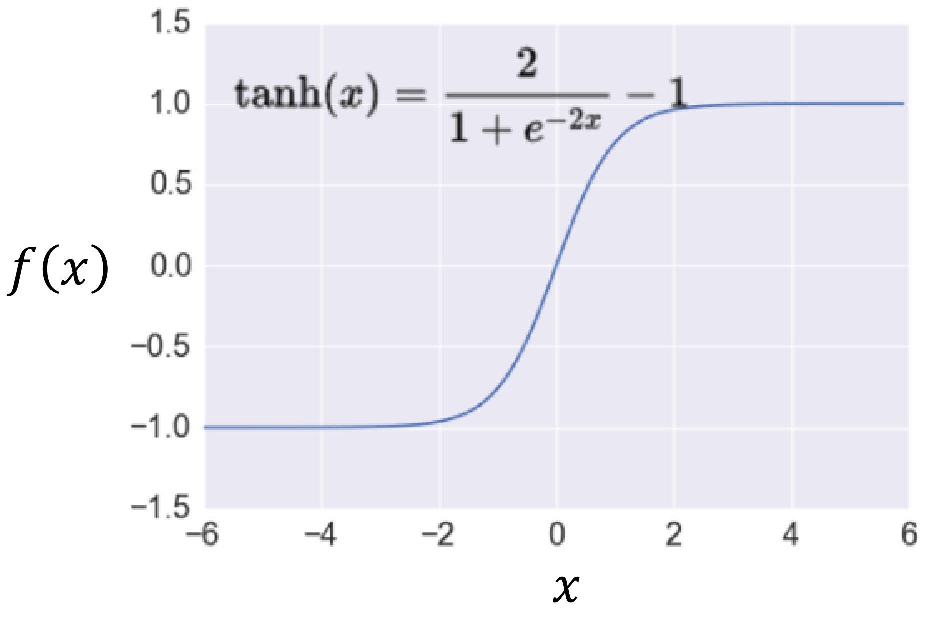
- *Sigmoid function* σ: takes a real-valued number and "squashes" it into the range between 0 and 1
  - The output can be interpreted as the firing rate of a biological neuron
    - Not firing = 0; Fully firing = 1
  - When the neuron's activation are 0 or 1, sigmoid neurons saturate
    - Gradients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



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

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: $\tanh(x) = 2 \cdot \sigma(2x) - 1$



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

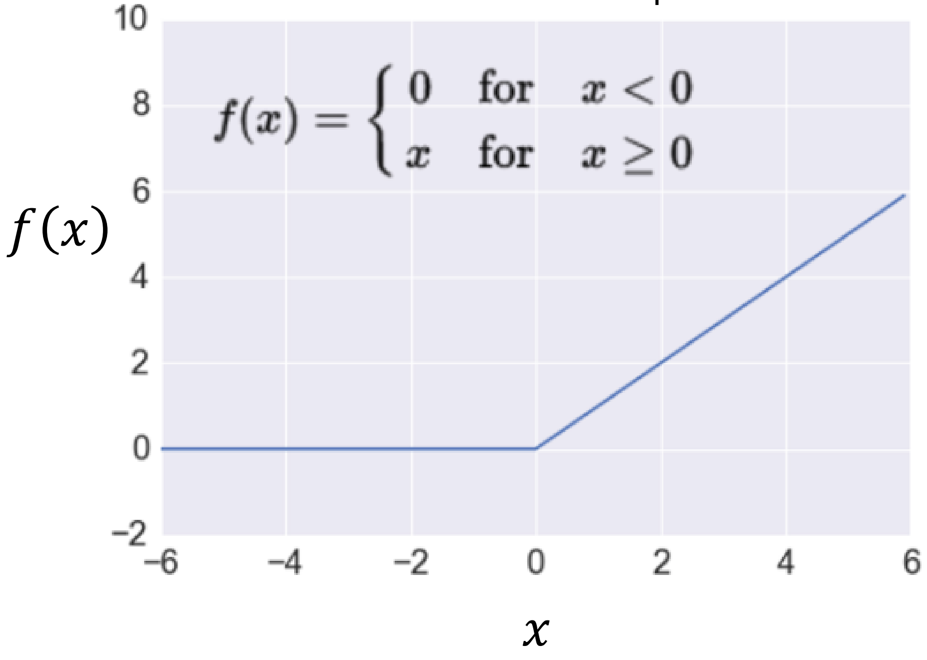
Activation: ReLU

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

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

  - 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

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

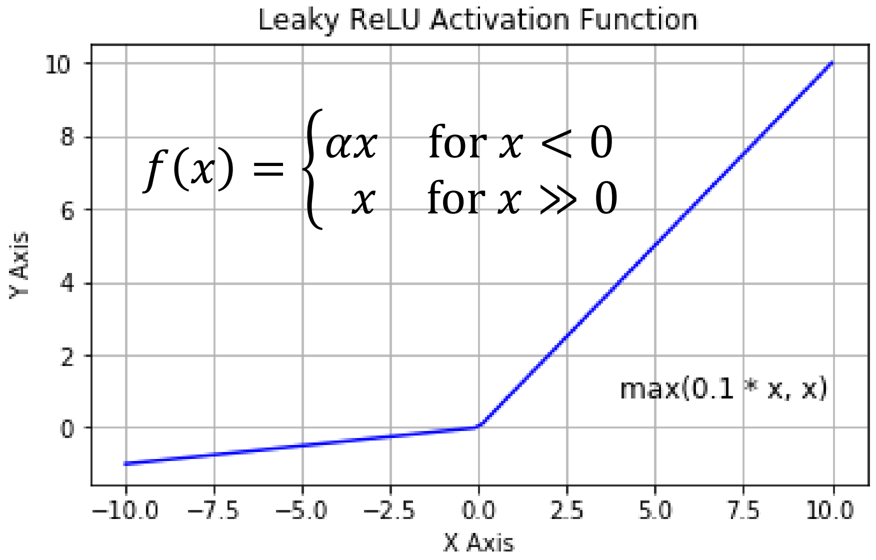


Activation: Leaky ReLU

Introduction to Neural Networks

- 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

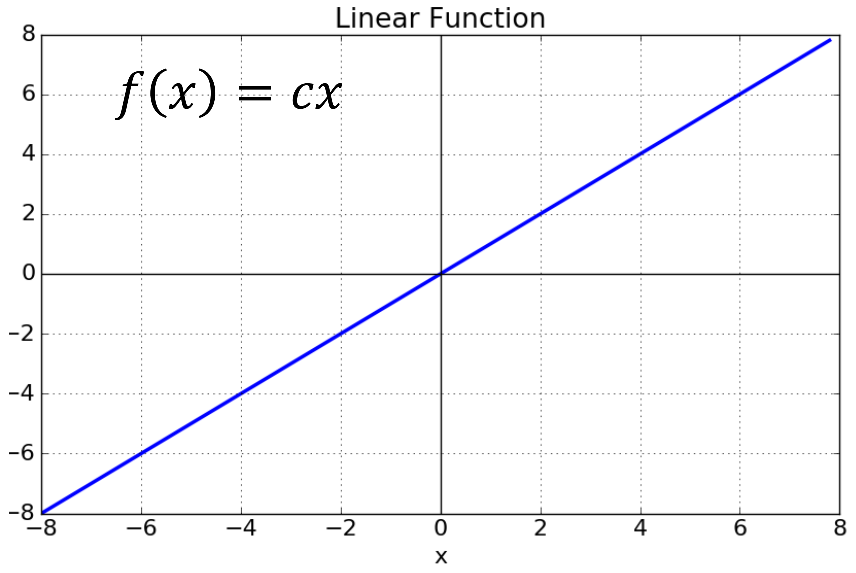


$$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \gg 0 \end{cases}$$

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)

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



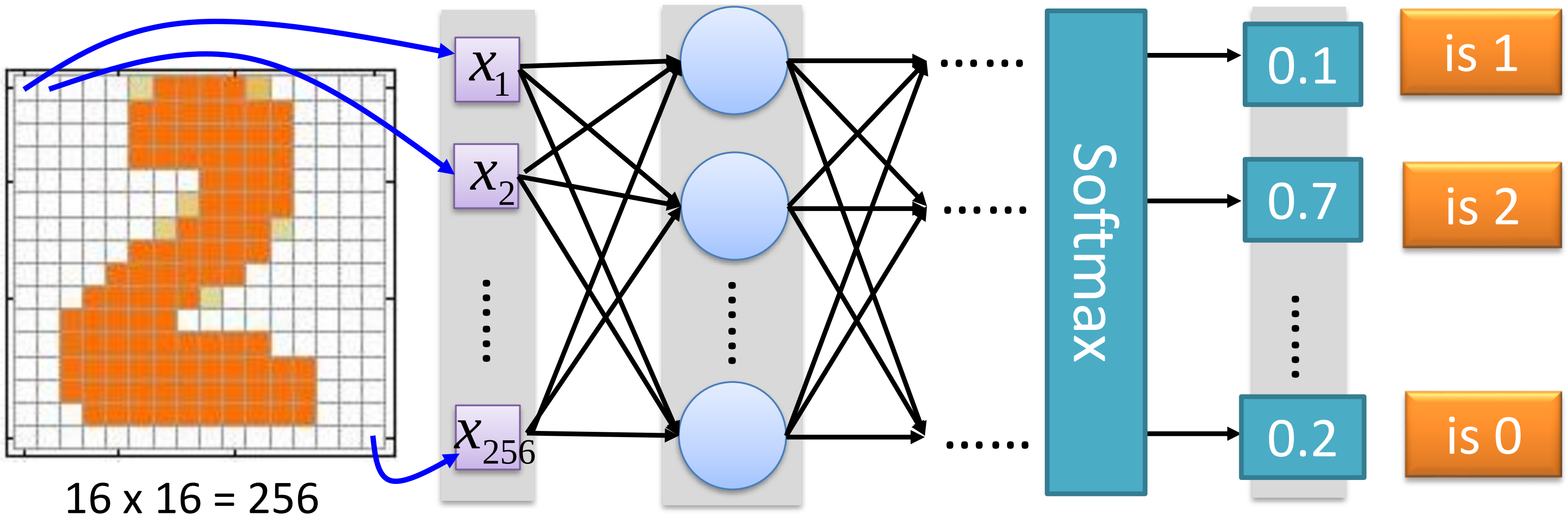
Training NNs

Training Neural Networks

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

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

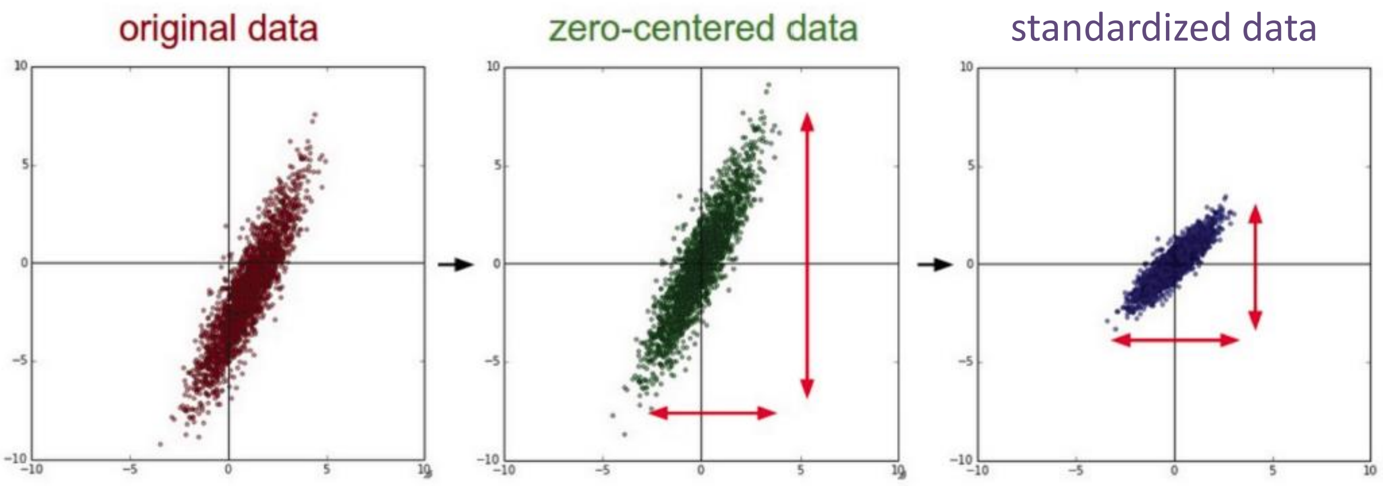
  - Often, the model parameters θ are referred to as weights
- Training a model to learn a set of parameters θ that are optimal (according to a criterion) is one of the greatest challenges in ML



Training NNs

Training Neural Networks

- *Data preprocessing* – helps convergence during training
  - Mean subtraction, to obtain zero-centered data
    - Subtract the mean for each individual data dimension (feature)
  - Standardization
    - In zero-centered data, divide each feature by its standard deviation
      - To obtain standard deviation of 1 and mean of 0 for each data dimension (feature)
  - Normalization
    - Scale the data within the range [0,1] or [-1, 1]
      - E.g., image pixel intensities are divided by 255 to be scaled in the [0,1] range

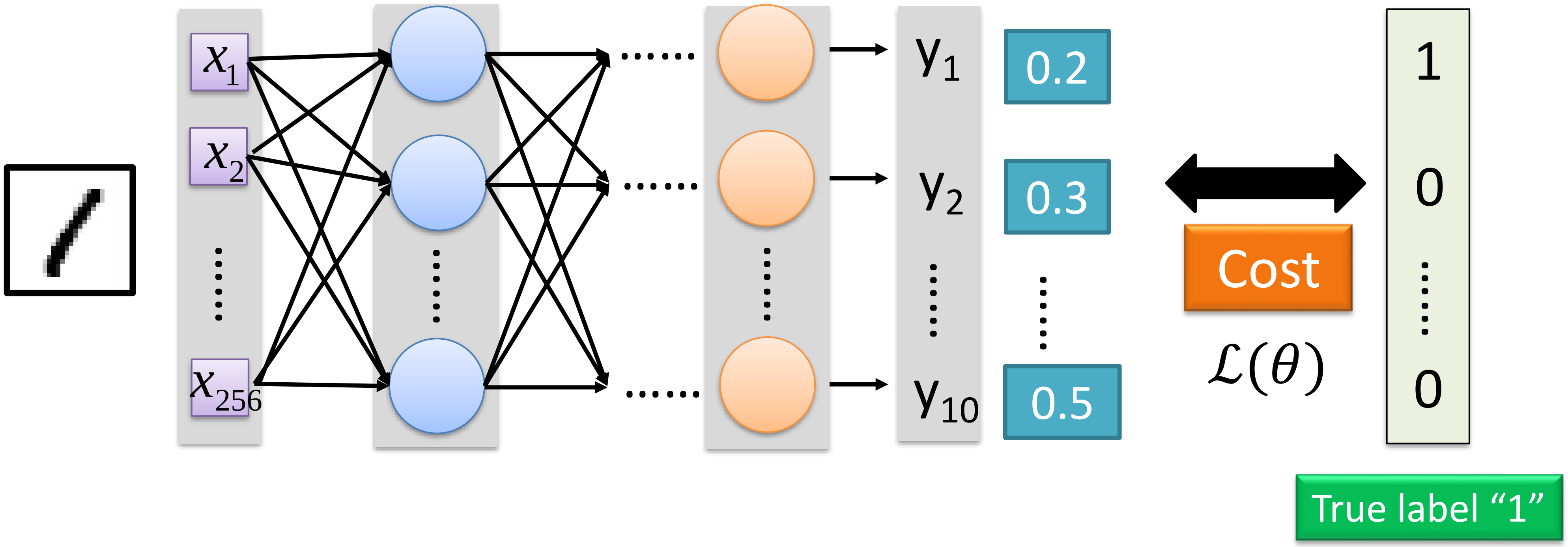


Picture from: https://cs231n.github.io/neural-networks-2/

Training NNs

Training Neural Networks

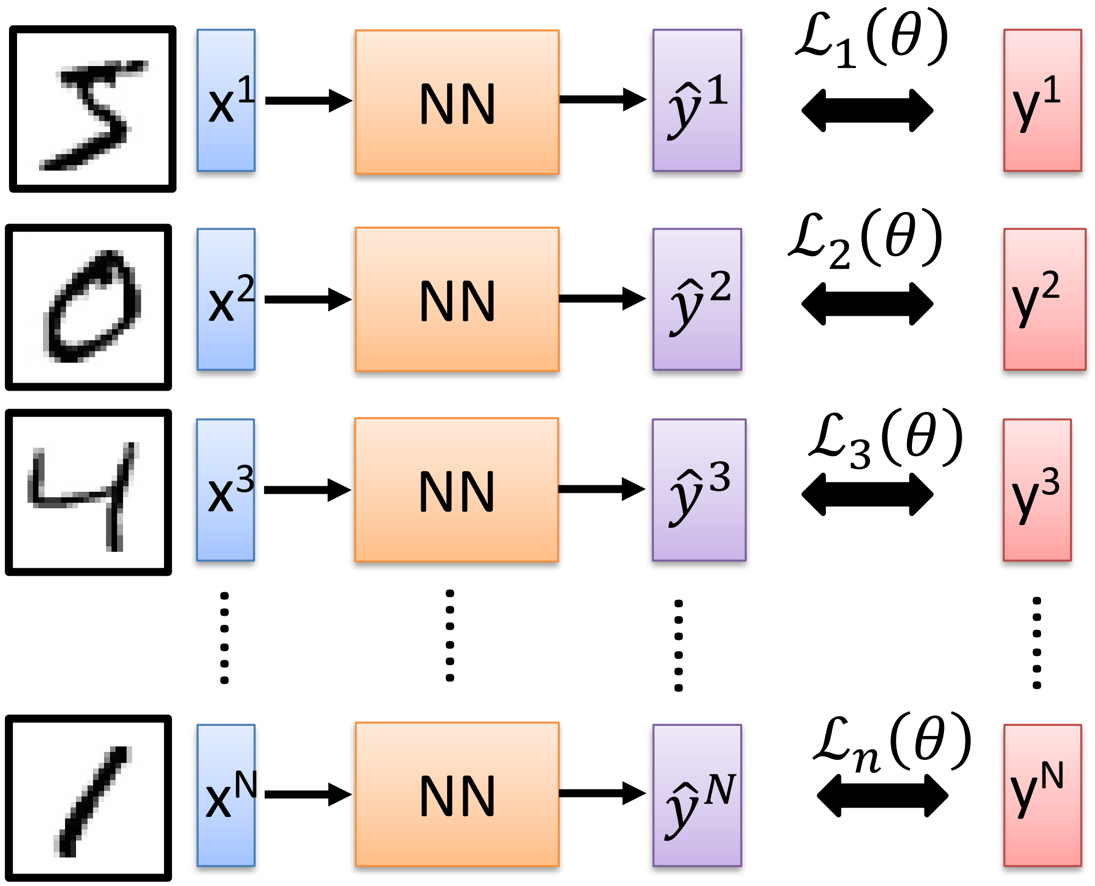
- Define a ***loss function***/objective function/cost function $\mathcal{L}(\theta)$ that calculates the difference (error) between the model prediction and the true label
  - E.g., $\mathcal{L}(\theta)$ can be mean-squared error, cross-entropy, etc.



Training NNs

Training Neural Networks

- For a training set of N images, calculate the total loss overall all images: $\mathcal{L}(\theta) = \sum_{n=1}^{N} \mathcal{L}_n(\theta)$
- Then, find the optimal parameters θ^* that minimize the total loss $\mathcal{L}(\theta)$



Loss Functions

Training Neural Networks

- ***Classification tasks***

Training examples	Pairs of N inputs x_i and ground-truth class labels y_i	
Output Layer	Softmax Activations [maps 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***	$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[y_k^{(i)} \log \hat{y}_k^{(i)}\right]$

Ground-truth class labels y_i and model predicted class labels $\hat{y}_i$

Loss Functions

Training Neural Networks

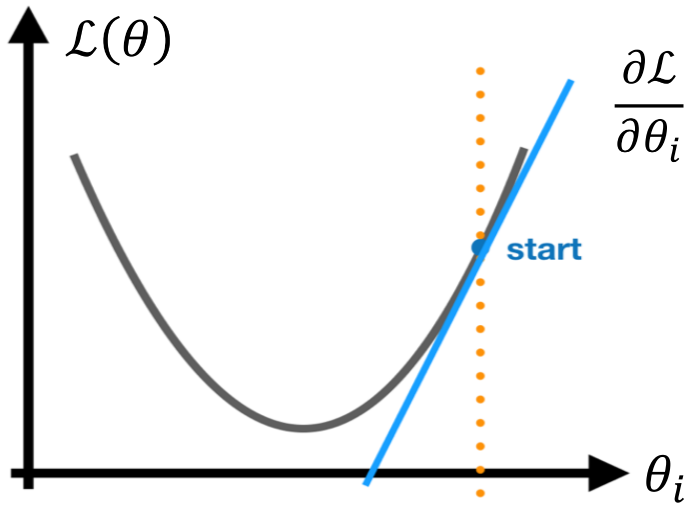
- *Regression tasks*

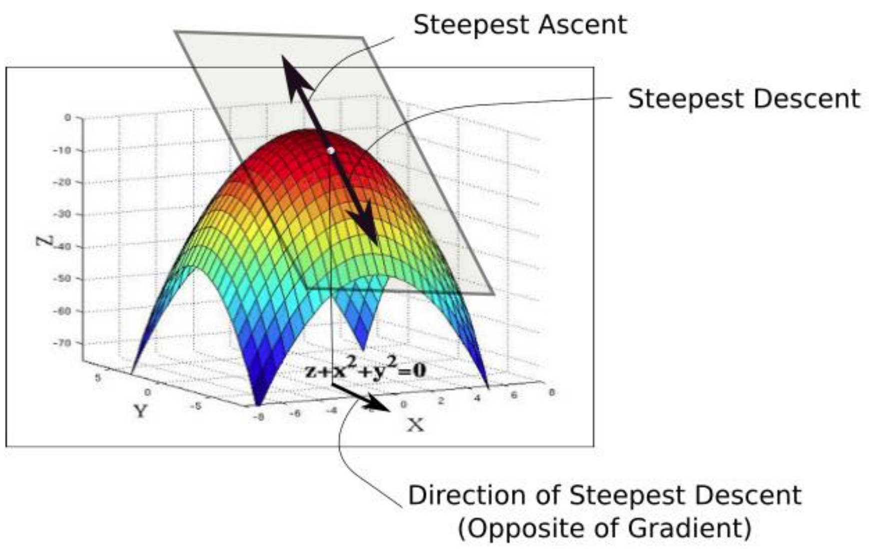
Training examples	Pairs of N inputs x_i and ground-truth output values y_i
Output Layer	Linear (Identity) Activation
Loss function	***Mean Squared Error*** $\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$
	Mean Absolute Error $\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left\|y^{(i)} - \hat{y}^{(i)}\right\|$

Training NNs

Training Neural Networks

- 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

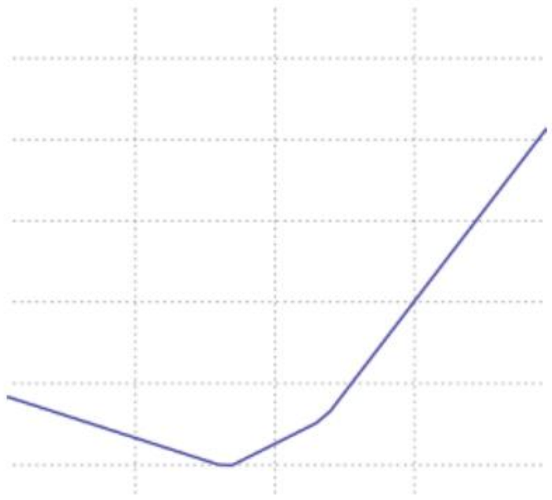


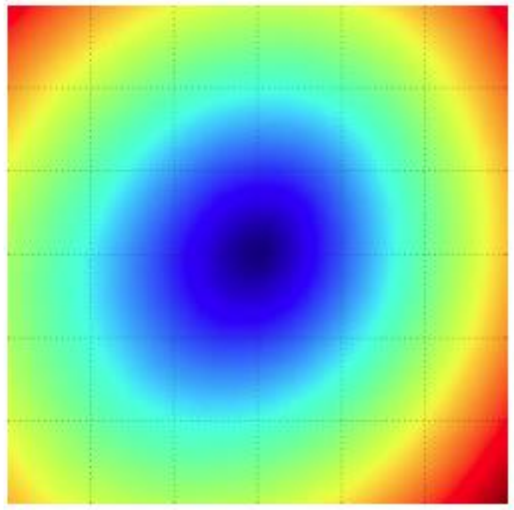


Training NNs

Training Neural Networks

- 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)

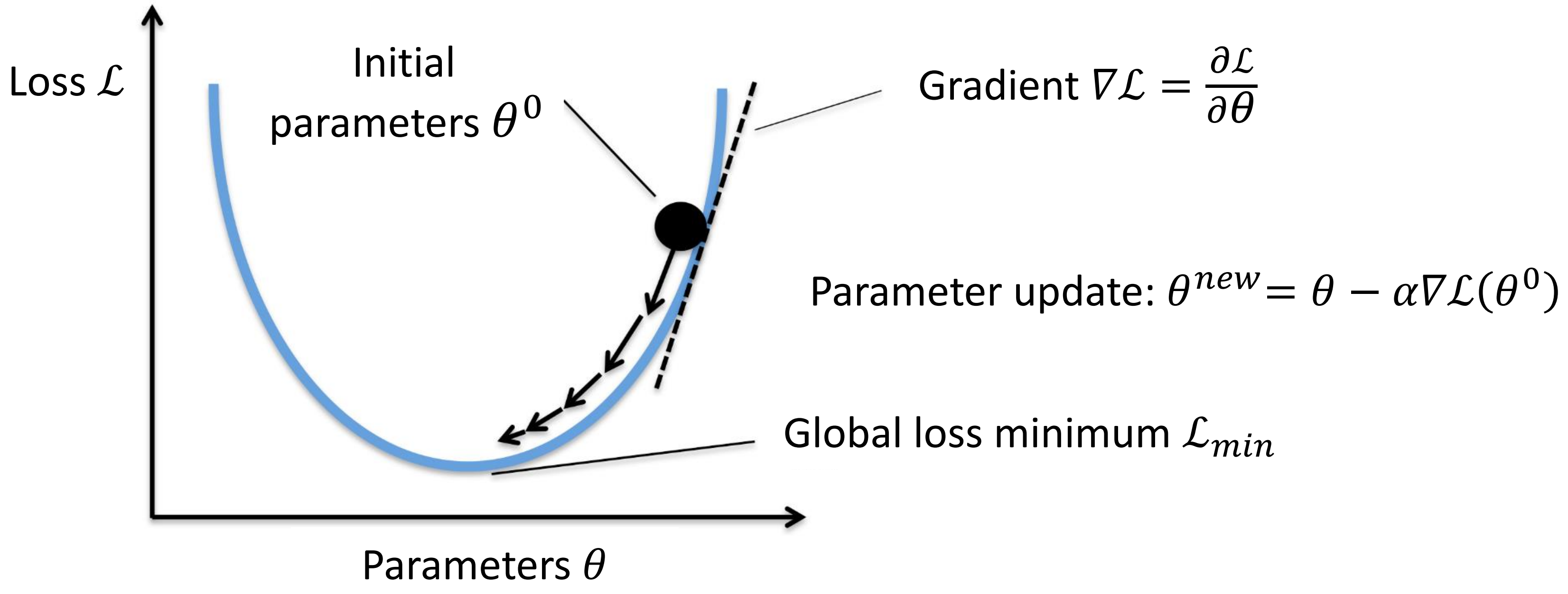

2D loss (blue = low loss, red = high loss)

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

Gradient Descent Algorithm

Training Neural Networks

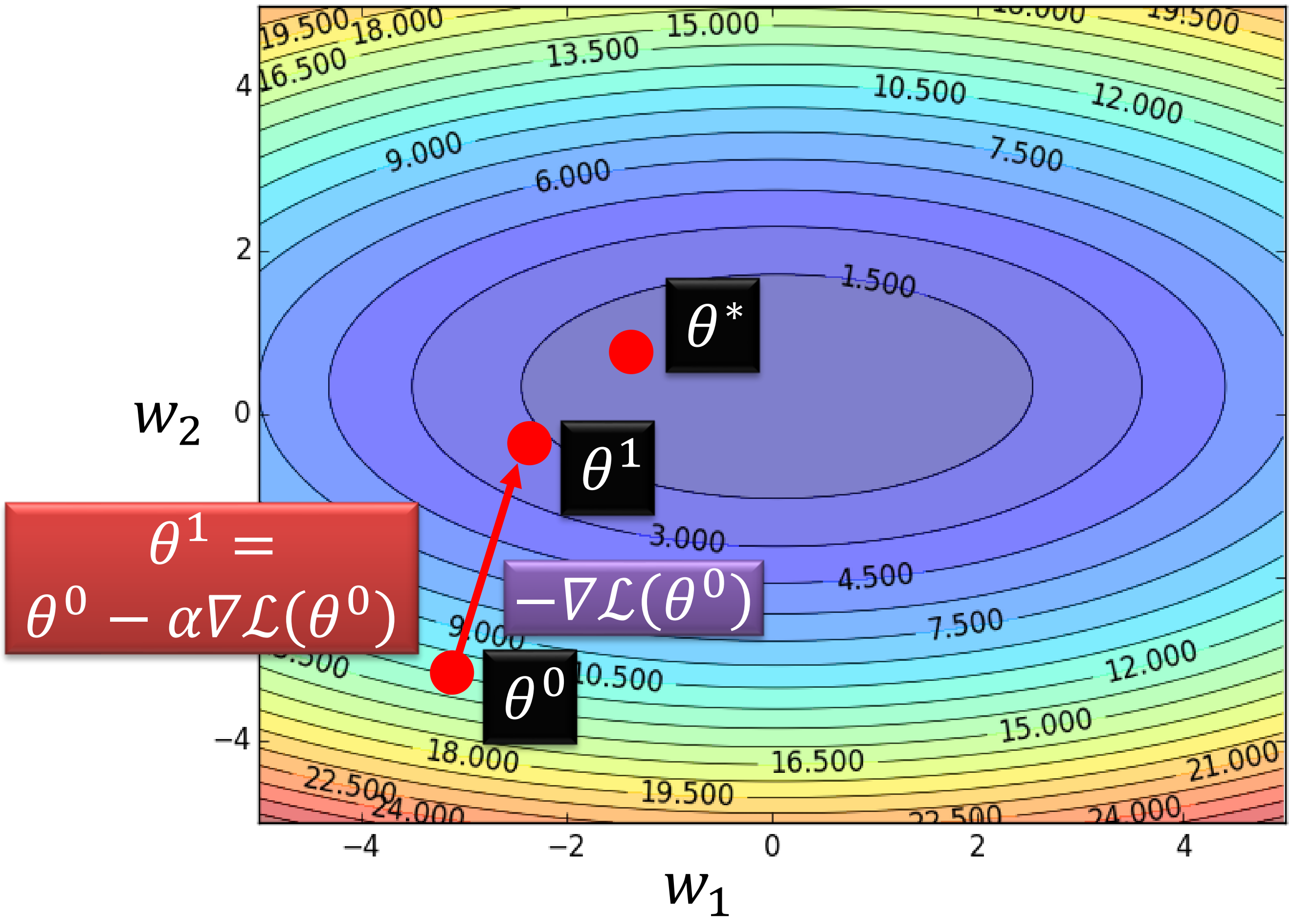
- Steps in the ***gradient descent algorithm***:
  1. Randomly initialize the model parameters, θ^0
  2. Compute the gradient of the loss function at the initial parameters θ^0: $\nabla\mathcal{L}(\theta^0)$
  3. Update the parameters as: $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$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


Loss $\mathcal{L}$
Initial parameters θ^0
Gradient $\nabla\mathcal{L} = \frac{\partial\mathcal{L}}{\partial\theta}$
Parameter update: $\theta^{new} = \theta - \alpha\nabla\mathcal{L}(\theta^0)$
Global loss minimum $\mathcal{L}_{min}$
Parameters θ

Gradient Descent Algorithm

Training Neural Networks

- Example: a NN with only 2 parameters w_1 and w_2, i.e., $\theta = \{w_1, w_2\}$
  - The different colors represent 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 α, and update θ,
   $\theta^1 = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$
4. Go to step 2, repeat

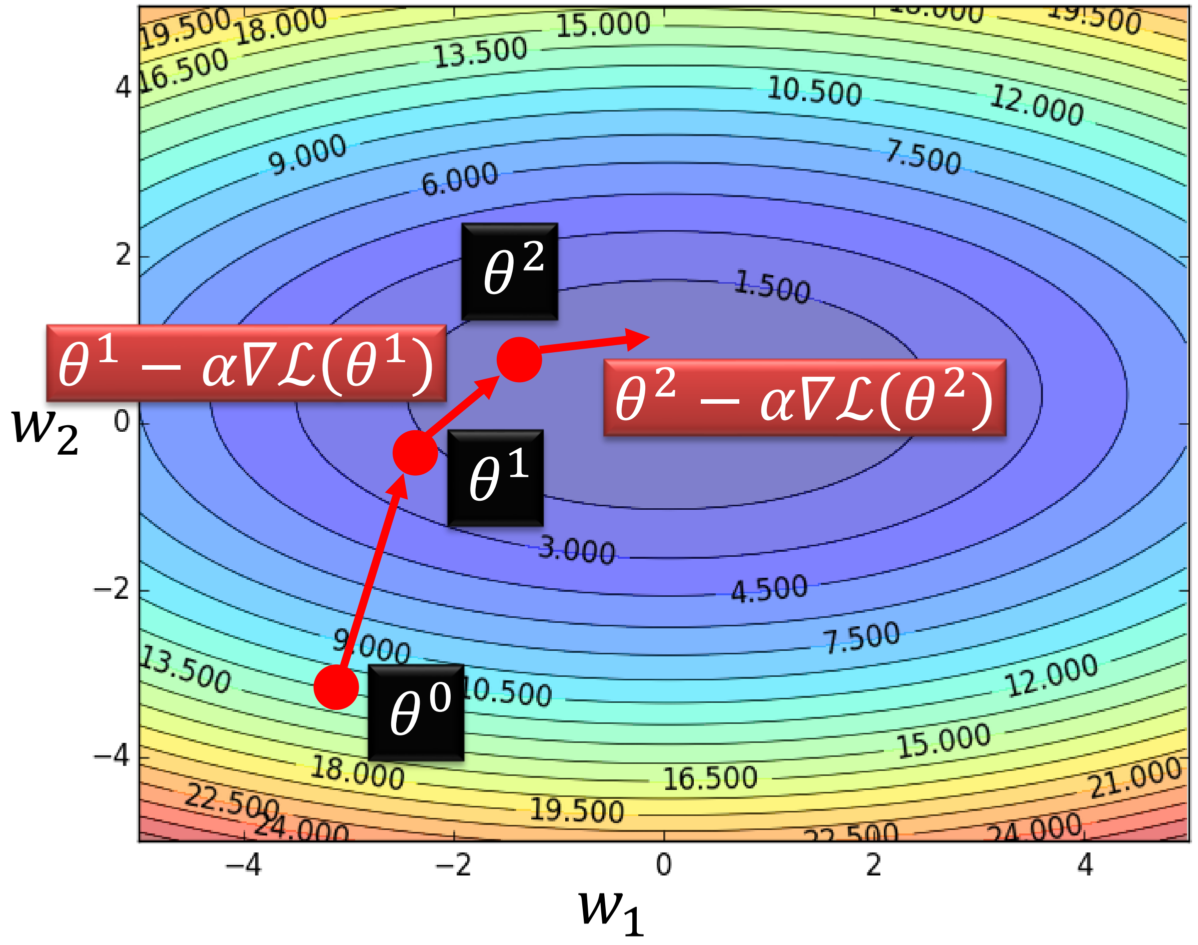
$$\nabla\mathcal{L}(\theta^0) = \begin{bmatrix} \partial\mathcal{L}(\theta^0)/\partial w_1 \\ \partial\mathcal{L}(\theta^0)/\partial w_2 \end{bmatrix}$$

Gradient Descent Algorithm

Training Neural Networks

- Example (contd.)

Eventually, we would reach a minimum …..



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

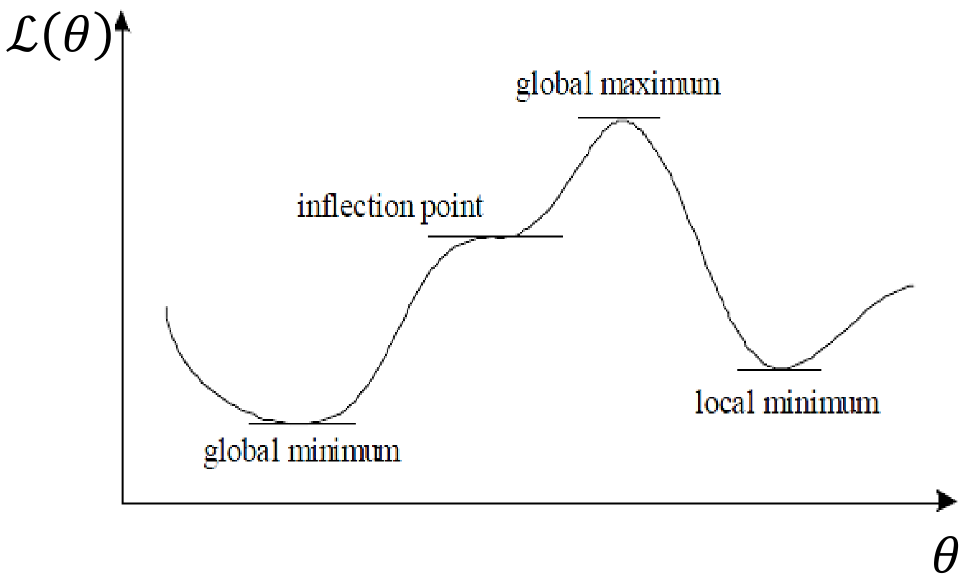
3. Times the learning rate α, and update θ,
$\theta^{new} = \theta^{old} - \alpha\nabla\mathcal{L}(\theta^{old})$

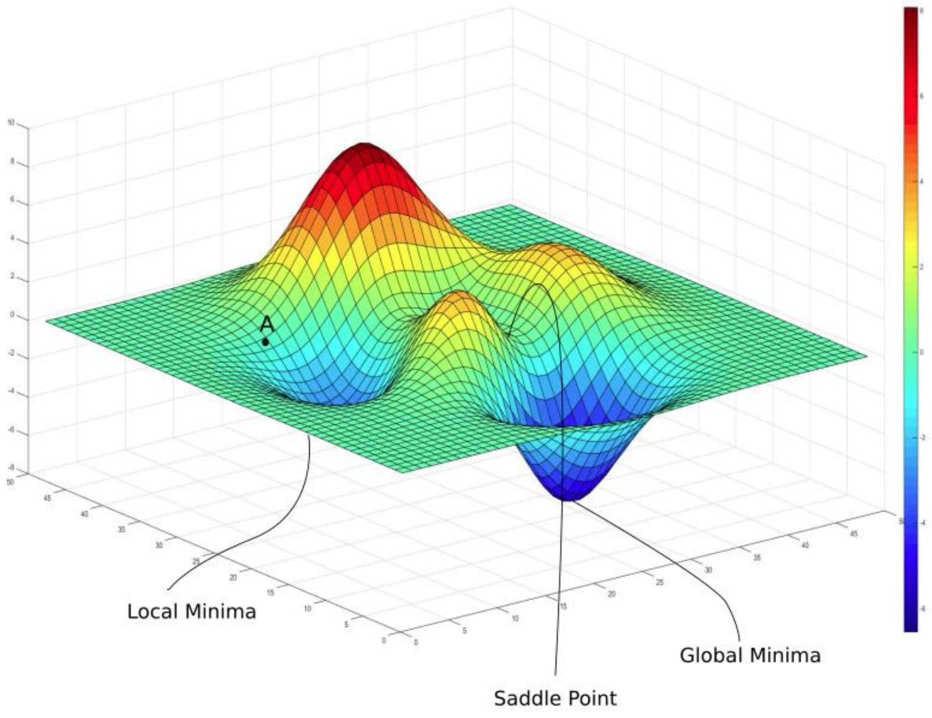
4. Go to step 2, repeat

Gradient Descent Algorithm

Training Neural Networks

- 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



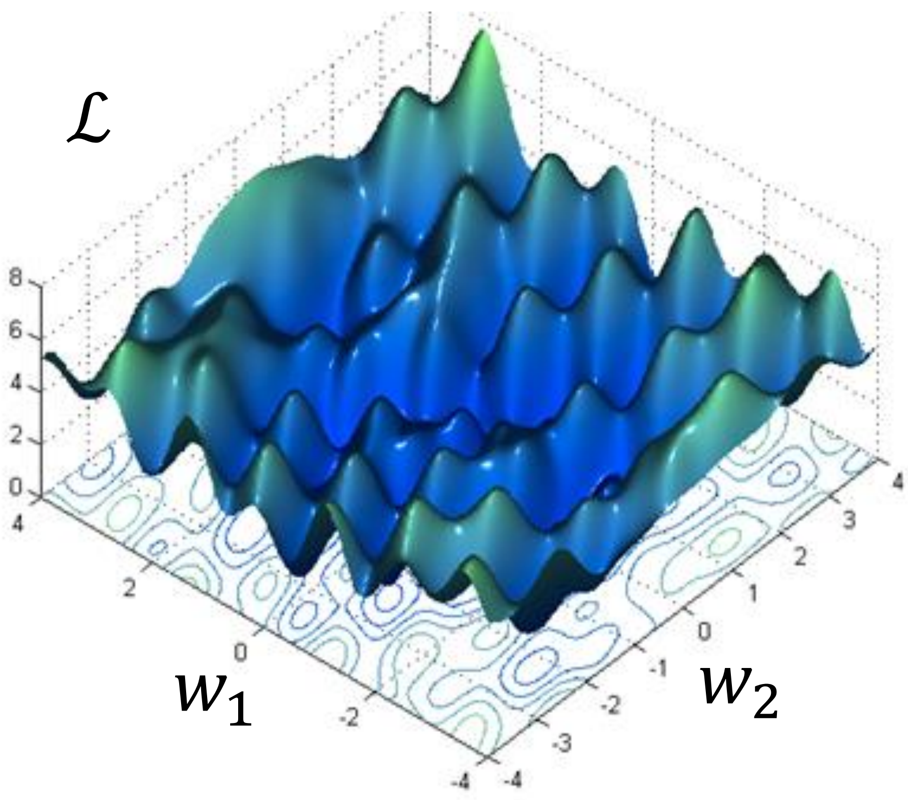


Picture from: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

Gradient Descent Algorithm

Training Neural Networks

- 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 every time the NN is trained
  - Gradient descent may reach different minima at every run
  - Therefore, NN will produce different predicted outputs
- In addition, currently we don't have algorithms that guarantee reaching a global minimum for an arbitrary loss function



Backpropagation

Training Neural Networks

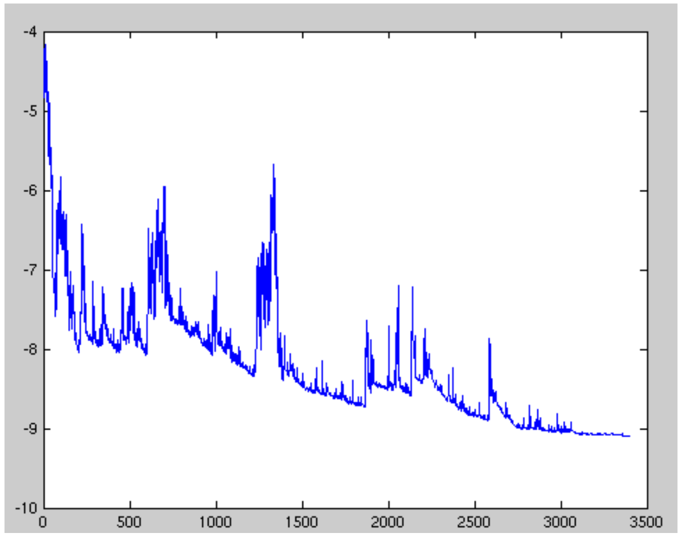
- Modern NNs employ the ***backpropagation*** method for calculating the gradients of the loss function $\nabla\mathcal{L}(\theta) = \partial\mathcal{L}/\partial\theta_i$
  - Backpropagation is short for "backward propagation"
- For training NNs, forward propagation (forward pass) refers to passing the inputs x through the hidden layers to obtain the model outputs (predictions) y
  - The loss $\mathcal{L}(y, \hat{y})$ function is then calculated
  - Backpropagation traverses the network in reverse order, from the outputs y backward toward the inputs x to calculate the gradients of the loss $\nabla\mathcal{L}(\theta)$
  - The chain rule is used for calculating the partial derivatives of the loss function with respect to the parameters θ in the different layers in the network
- Each update of the model parameters θ during training takes one forward and one backward pass (e.g., of a batch of inputs)
- Automatic calculation of the gradients (automatic differentiation) is available in all current deep learning libraries
  - It significantly simplifies the implementation of deep learning algorithms, since it obviates deriving the partial derivatives of the loss function by hand

Mini-batch Gradient Descent

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

Stochastic Gradient Descent

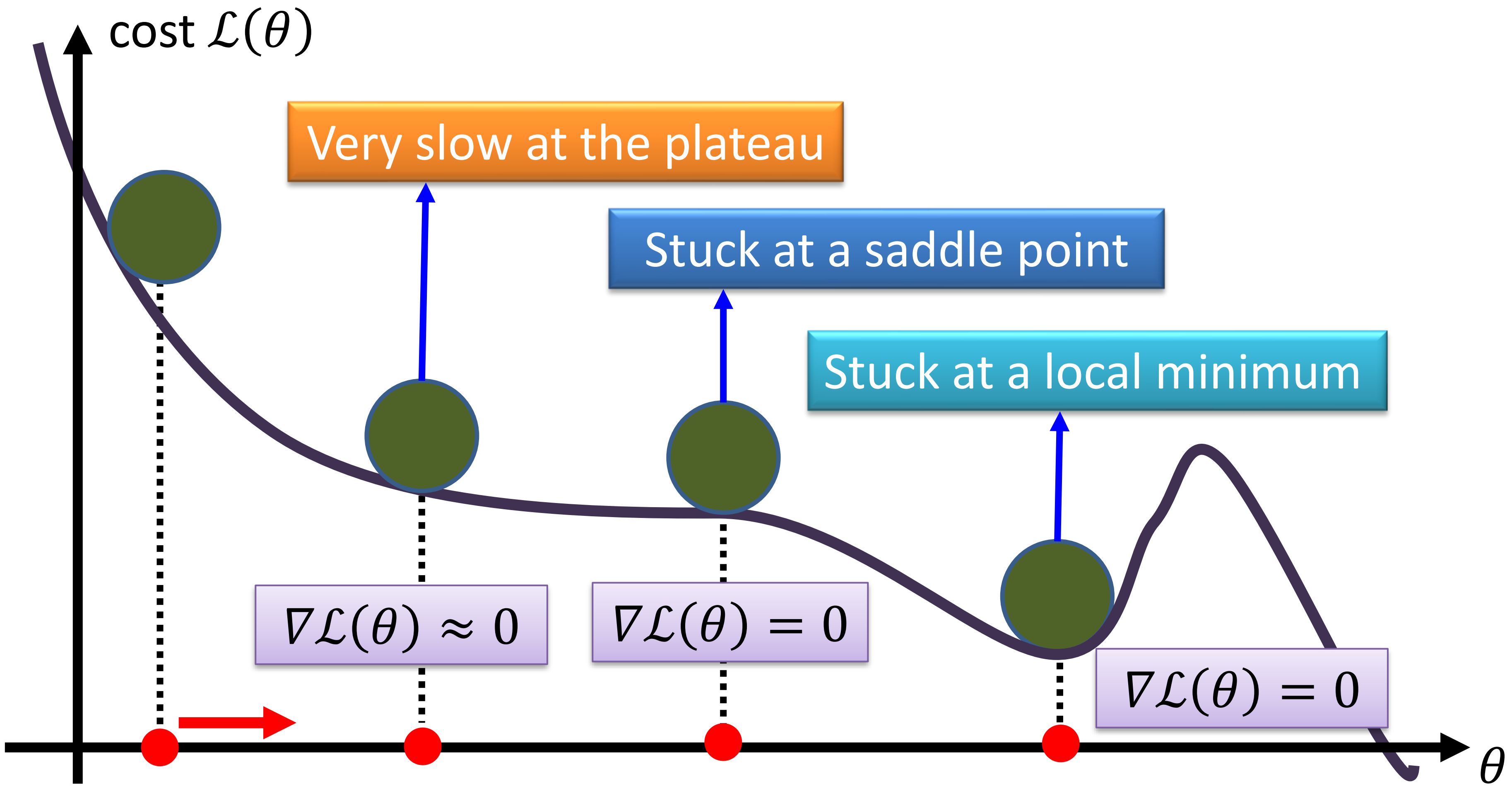
- ***Stochastic gradient descent***
  - SGD uses mini-batches that consist of a single input example
    - E.g., one image mini-batch
  - Although this method is very fast, it may cause significant fluctuations in the loss function
    - Therefore, it is less commonly used, and mini-batch GD is preferred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



Problems with Gradient Descent

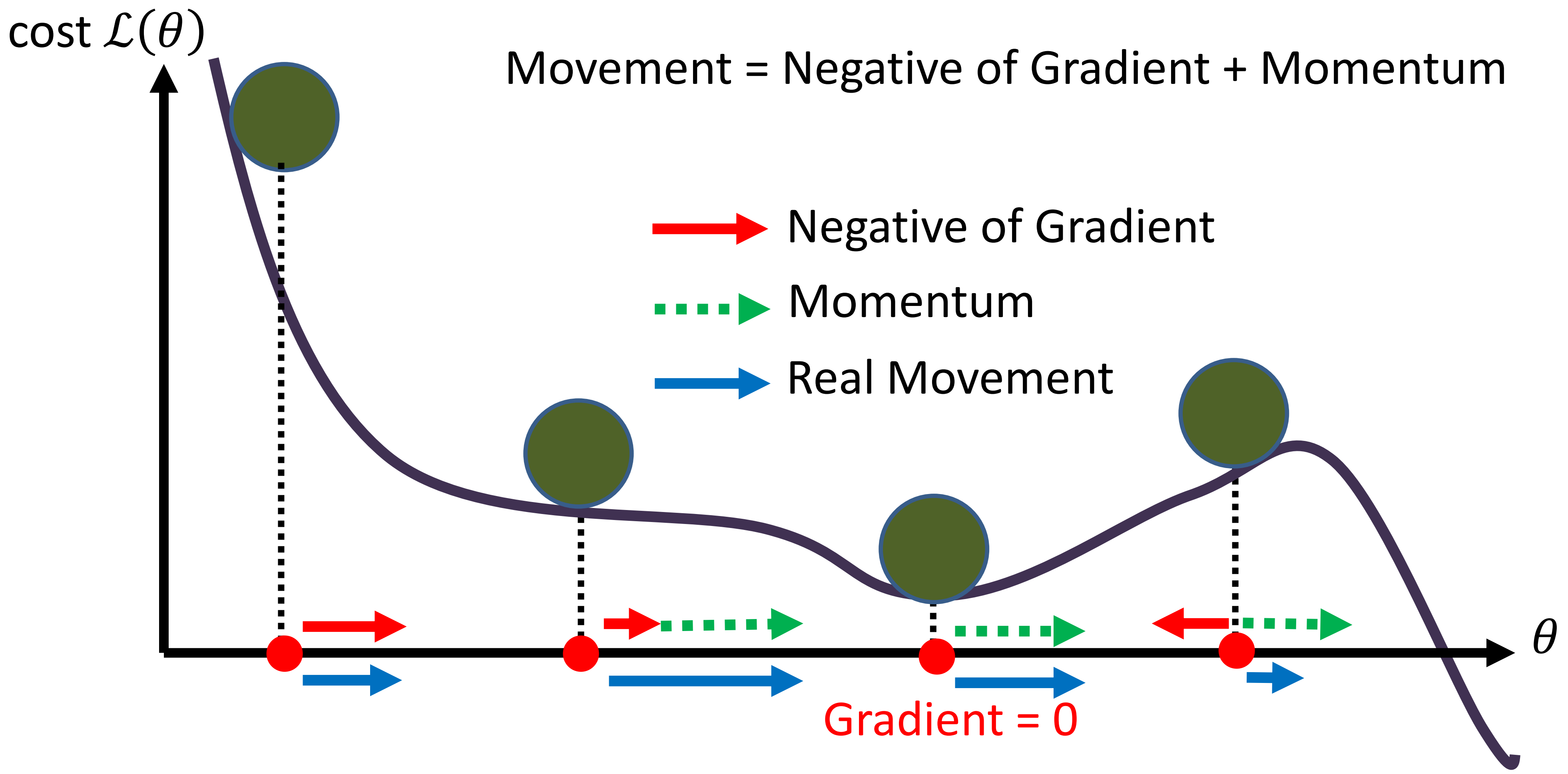
Training Neural Networks

- 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* at iteration t: $\theta^t = \theta^{t-1} - V^t$
  - Where: $V^t = \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1})$
  - I.e., $\theta^t = \theta^{t-1} - \alpha \nabla \mathcal{L}(\theta^{t-1}) - \beta V^{t-1}$
- Compare to vanilla GD: $\theta^t = \theta^{t-1} - \alpha \nabla \mathcal{L}(\theta^{t-1})$
  - Where θ^{t-1} are the parameters from the previous iteration $t-1$
- The term V^t is called momentum
  - This term accumulates the gradients from the past several steps, i.e.,

$$\begin{aligned} V^t &= \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta\left(\beta V^{t-2} + \alpha \nabla \mathcal{L}(\theta^{t-2})\right) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta^2 V^{t-2} + \beta\alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta^3 V^{t-3} + \beta^2\alpha \nabla \mathcal{L}(\theta^{t-3}) + \beta\alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \end{aligned}$$

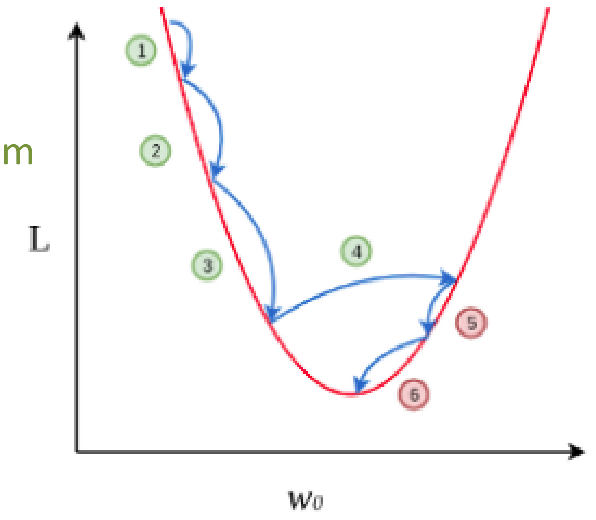
  - This term is analogous to a momentum of a heavy ball rolling down the hill
- The parameter β is referred to as a coefficient of momentum
  - A typical value of the parameter β is 0.9
- This method updates the parameters θ in the direction of the weighted average of the past gradients

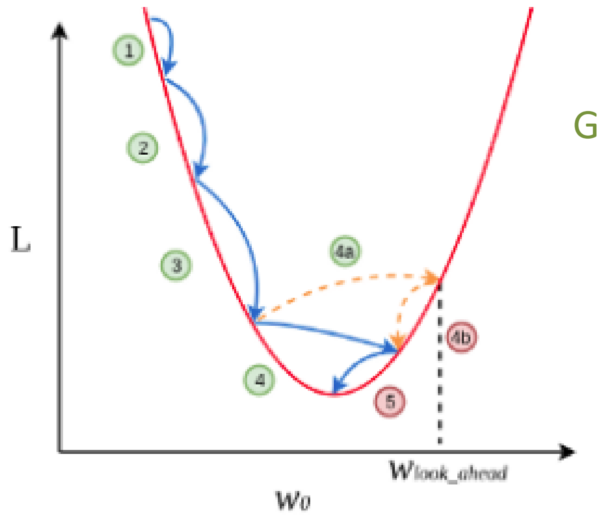
Nesterov Accelerated Momentum

Training Neural Networks

- ***Gradient descent with Nesterov accelerated momentum***
  - Parameter update: $\theta^t = \theta^{t-1} - V^t$
    - Where: $V^t = \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1} + \beta V^{t-1})$
  - The term $\theta^{t-1} + \beta V^{t-1}$ allows to predict the position of the parameters in the next step (i.e., $\theta^t \approx \theta^{t-1} + \beta V^{t-1}$)
  - The gradient is calculated with respect to the approximate future position of the parameters in the next iteration, θ^t, calculated at iteration $t-1$

GD with momentum





GD with Nesterov momentum

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

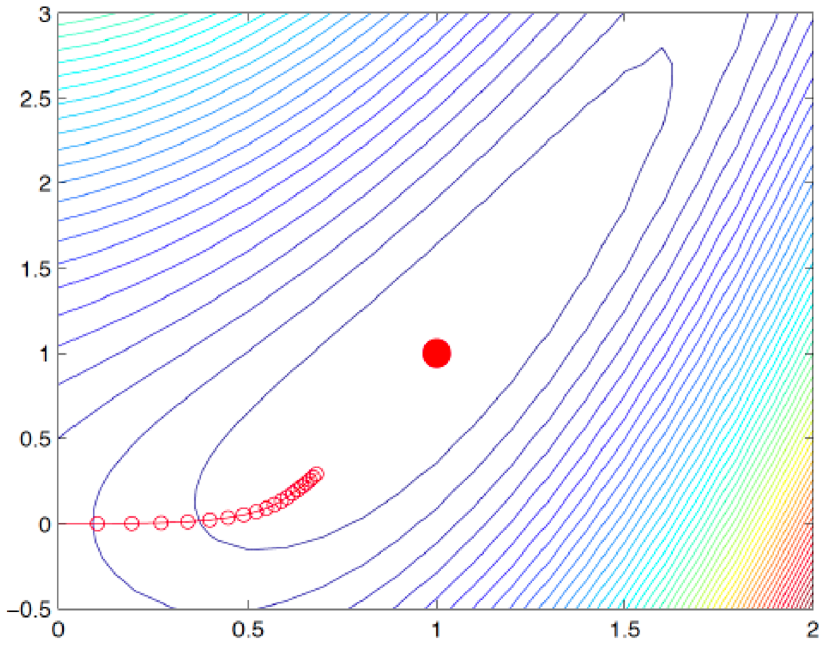
Adam

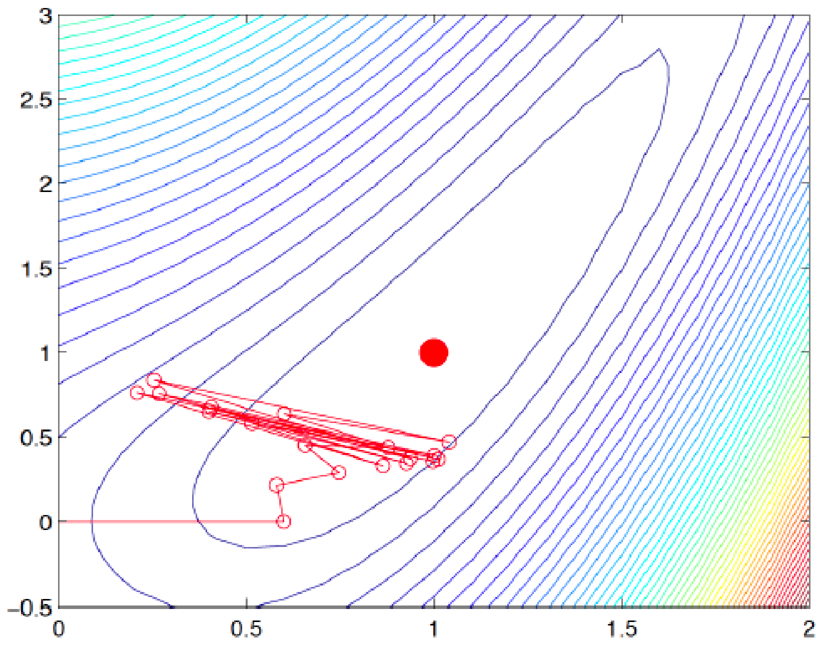
- ***Adaptive Moment Estimation (Adam)***
  - Adam combines insights from the momentum optimizers that accumulate the values of past gradients, and it also introduces new terms based on the second moment of the gradient
    - Similar to GD with momentum, Adam computes a **weighted average of past gradients** (first moment of the gradient), i.e., $V^t = \beta_1 V^{t-1} + (1-\beta_1)\nabla\mathcal{L}(\theta^{t-1})$
    - Adam also computes a **weighted average of past squared gradients** (second moment of the gradient), i.e., $U^t = \beta_2 U^{t-1} + (1-\beta_2)\left(\nabla\mathcal{L}(\theta^{t-1})\right)^2$
  - The parameter update is: $\theta^t = \theta^{t-1} - \alpha \frac{\widehat{V}^t}{\sqrt{\widehat{U}^t} + \epsilon}$
    - Where: $\widehat{V}^t = \frac{V^t}{1-\beta_1}$ and $\widehat{U}^t = \frac{U^t}{1-\beta_2}$
    - 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 commonly used optimizers nowadays are Adam and SGD with momentum

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

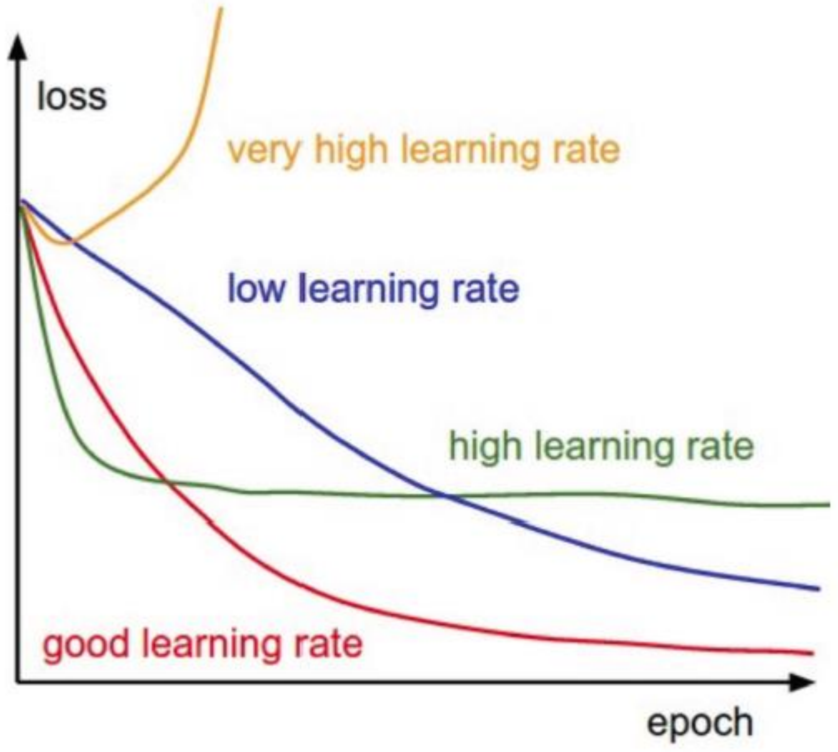




LR too large

Learning Rate

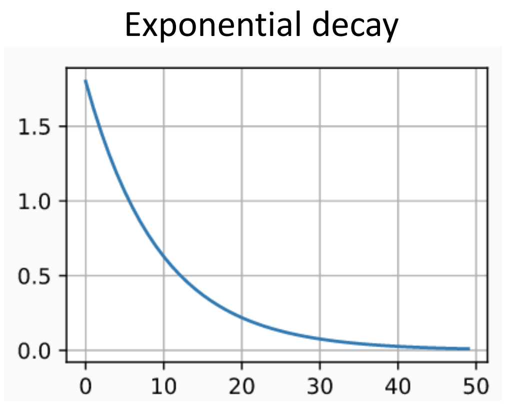
- Training loss for different learning rates
  - High learning rate: the loss increases or plateaus too quickly
  - Low learning rate: the loss decreases too slowly (takes many epochs to reach a solution)

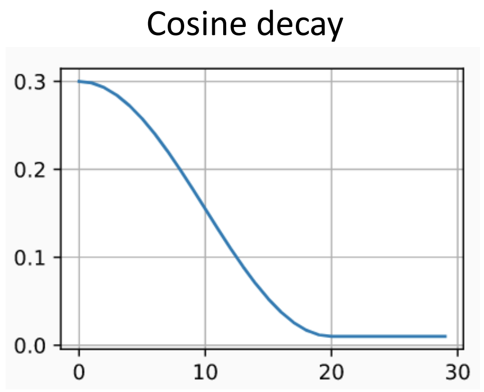


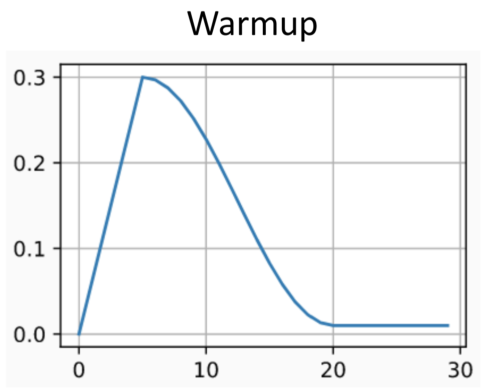
Picture from: https://cs231n.github.io/neural-networks-3/

Learning Rate Scheduling

- ***Learning rate scheduling*** is applied to change the values of the learning rate during the training
  - ***Annealing*** is reducing the learning rate over time (a.k.a. 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 divide by 10 every 20 epochs
    - Approach 2: exponential or cosine decay gradually reduce the learning rate over time
    - Approach 3: 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)
  - ***Warmup*** is gradually increasing the learning rate initially, and afterward let it cool down until the end of the training

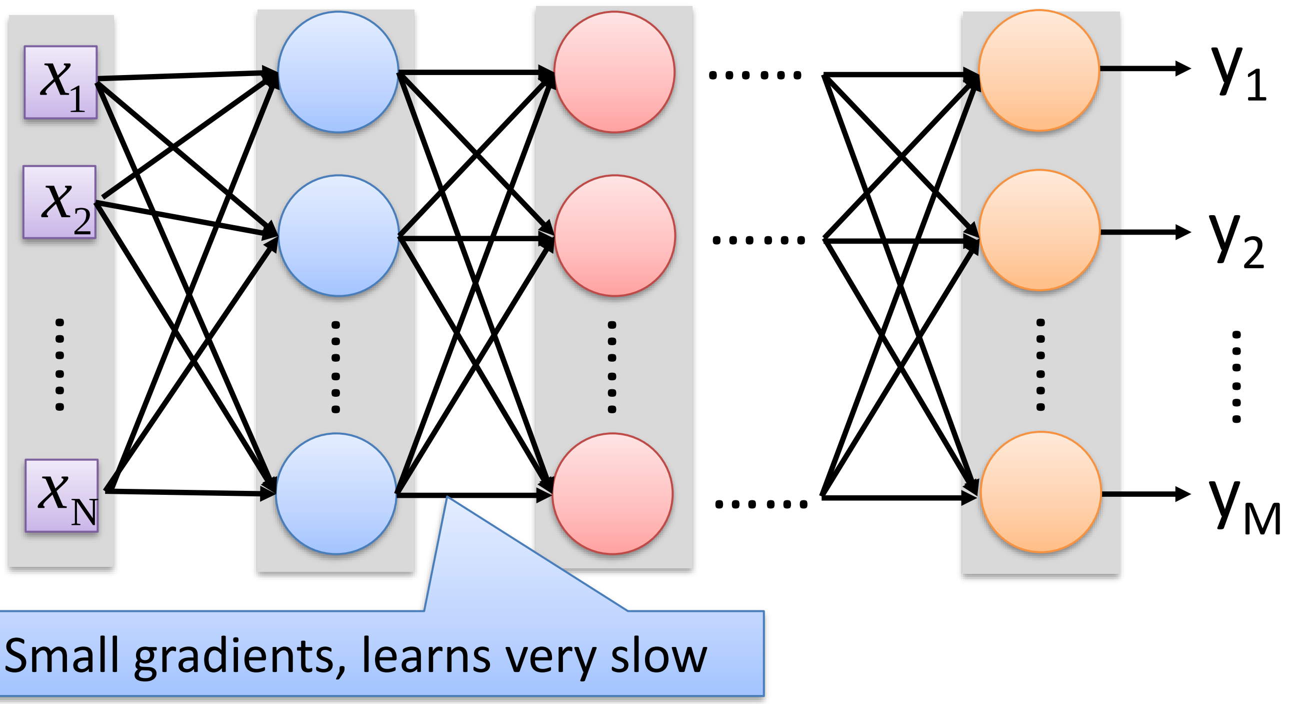






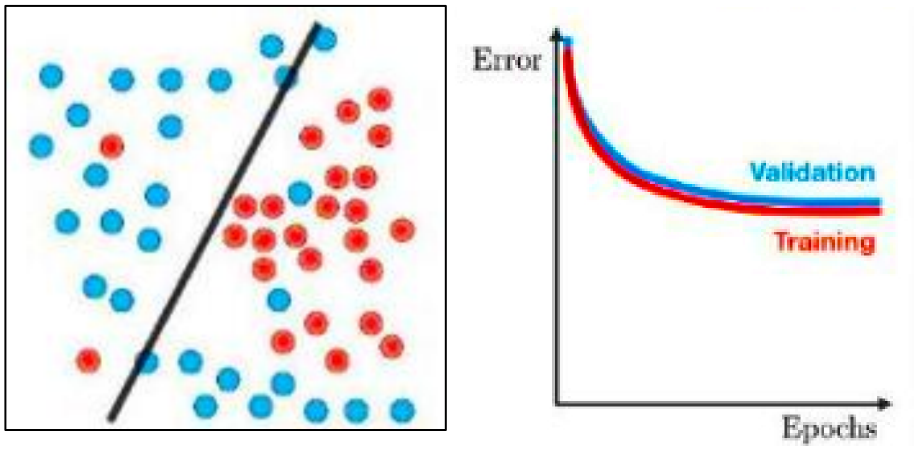
Vanishing Gradient Problem

- 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: change learning rate, ReLU activations, regularization, LSTM units in RNNs

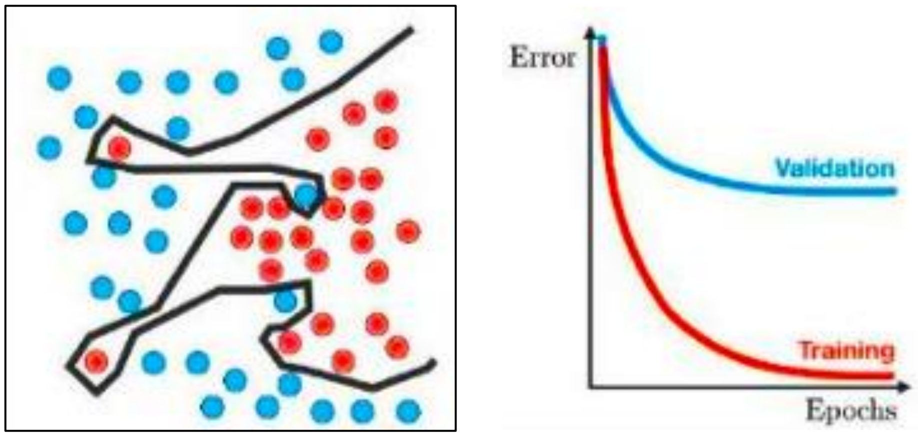


Generalization

- ***Underfitting***
  - The model is too "simple" to represent all the relevant class characteristics
  - E.g., model with too few parameters
  - Produces high error on the training set and high error on the validation set



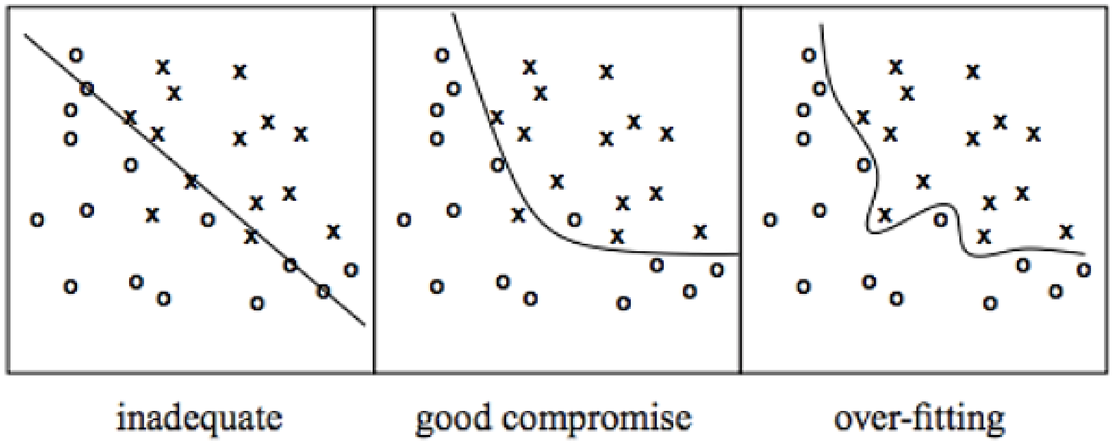
- ***Overfitting***
  - The model is too "complex" and fits irrelevant characteristics (noise) in the data
  - E.g., model with too many parameters
  - Produces low error on the training error and high error on the validation set



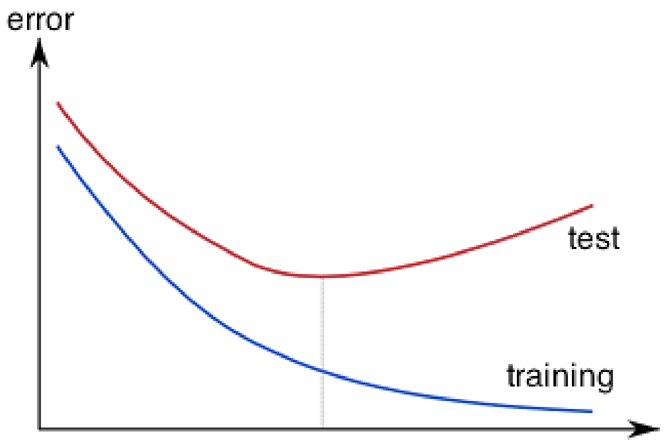
Overfitting

Generalization

- 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 or validation data)



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

Regularization: Weight Decay

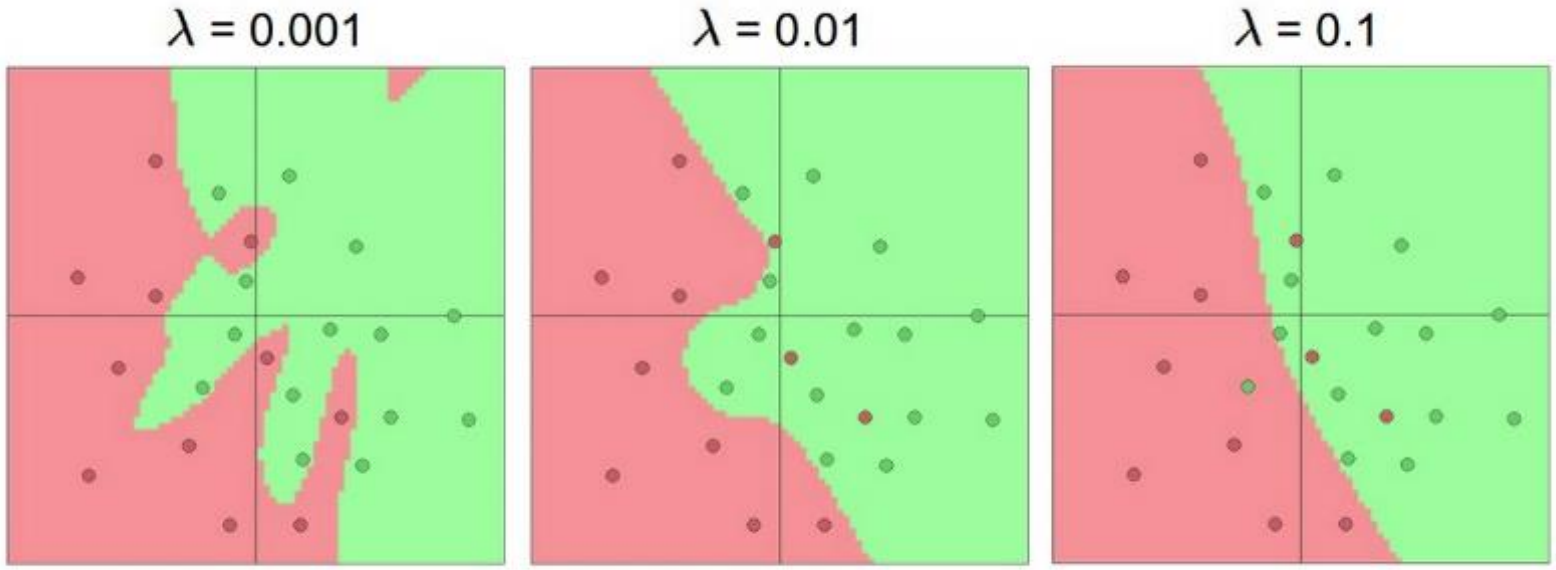
- ***ℓ_2 weight decay***
  - A regularization term that penalizes large weights is added to the loss function

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

  - 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

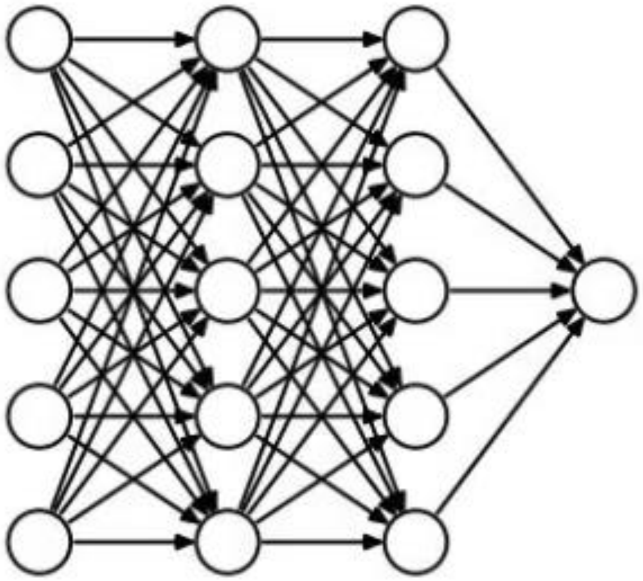
$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k |\theta_k|$$

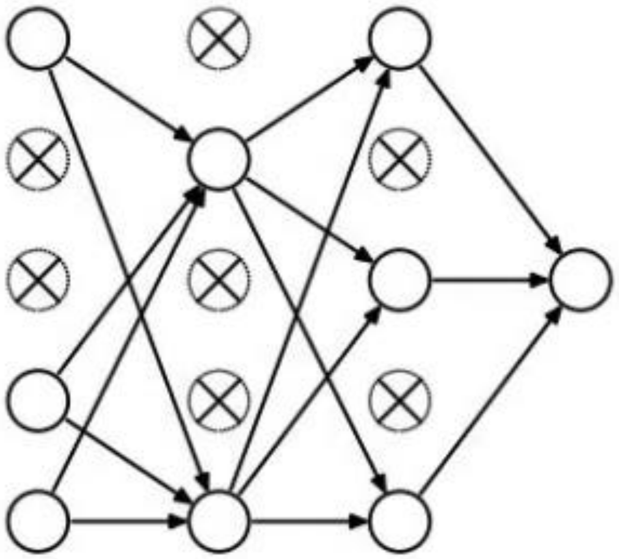
  - ℓ_1 weight decay is less common with NN
    - Often performs worse than ℓ_2 weight decay
  - It is also possible to combine ℓ_1 and ℓ_2 regularization
    - Called elastic net regularization

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

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* needs to be chosen (tuned)
    - Often, between 20% and 50% of the units are dropped

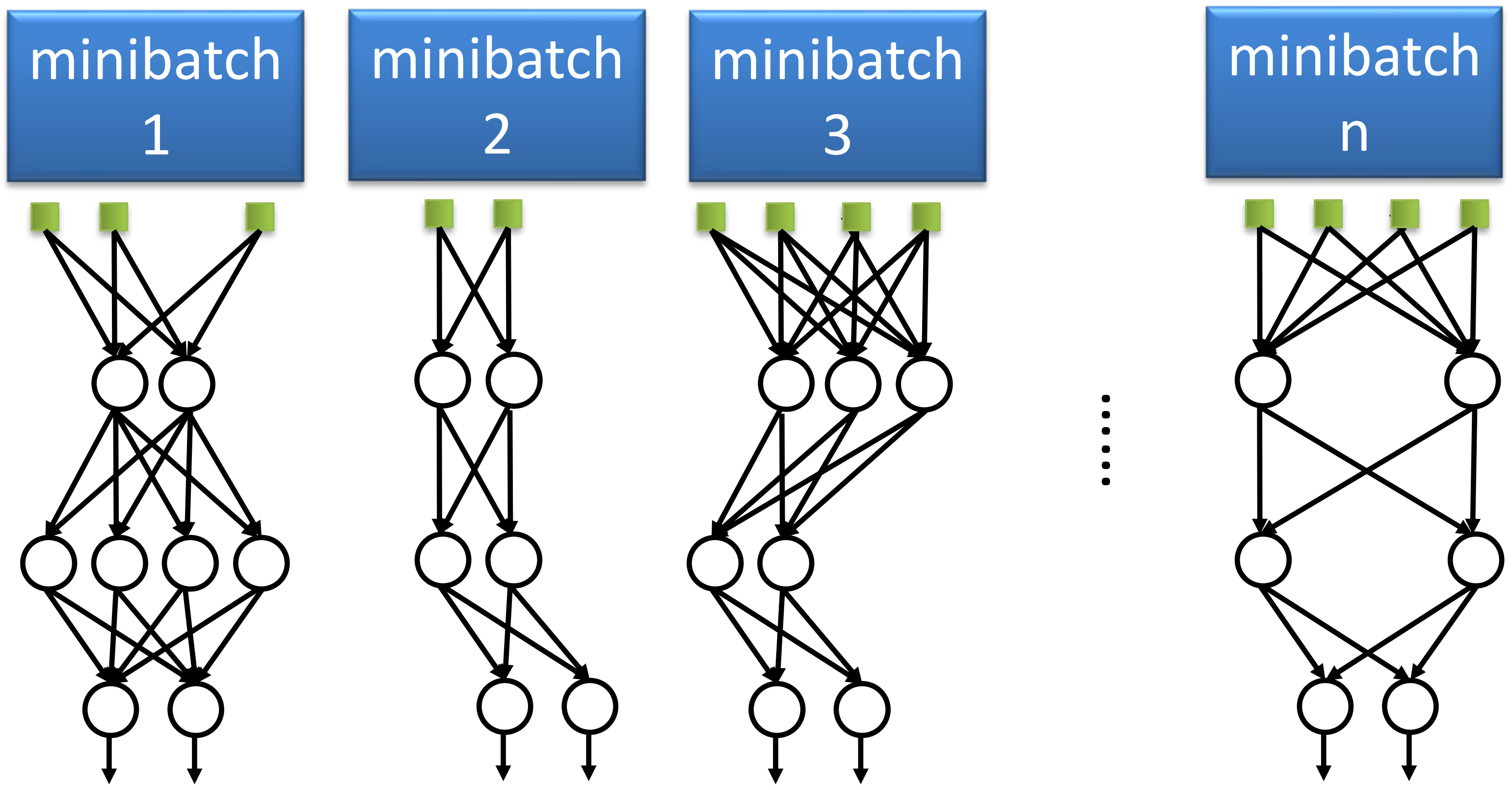




Regularization: Dropout

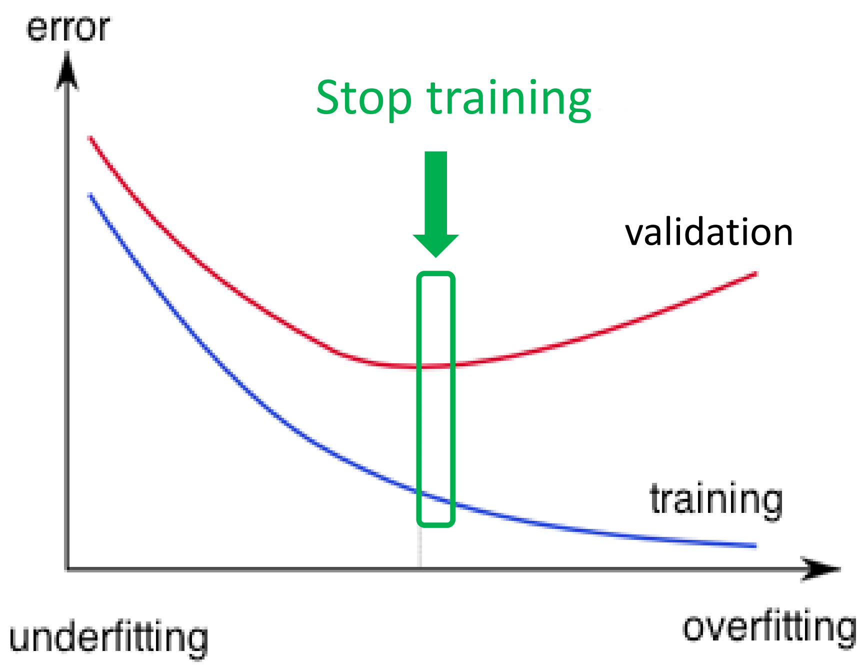
Regularization

- 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 validation/train 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



Regularization: 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
- BatchNorm layers are inserted immediately after convolutional layers or fully-connected layers, and before activation layers
  - They are very common with convolutional NNs

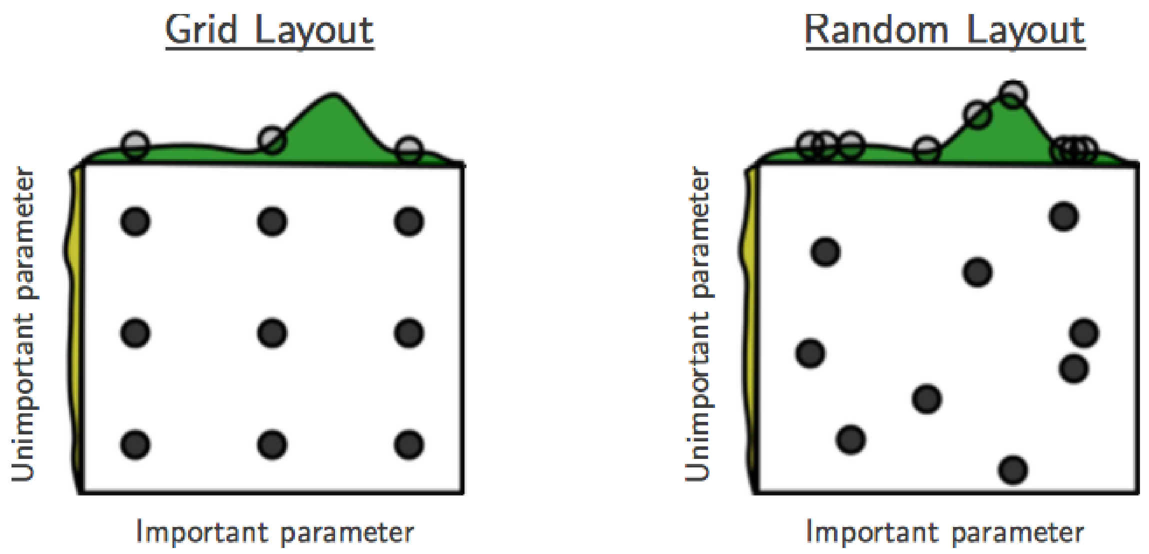
Hyper-parameter Tuning

- Training NNs can involve setting many ***hyper-parameters***
- 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)
  - Optimizer type
- Other hyper-parameters may include:
  - Regularization parameters (ℓ_2 penalty, dropout rate)
  - Batch size
  - Activation functions
  - Loss function
- Hyper-parameter tuning can be time-consuming for larger NNs

Hyper-parameter Tuning

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-parameter optimization
  - Is an active area of research

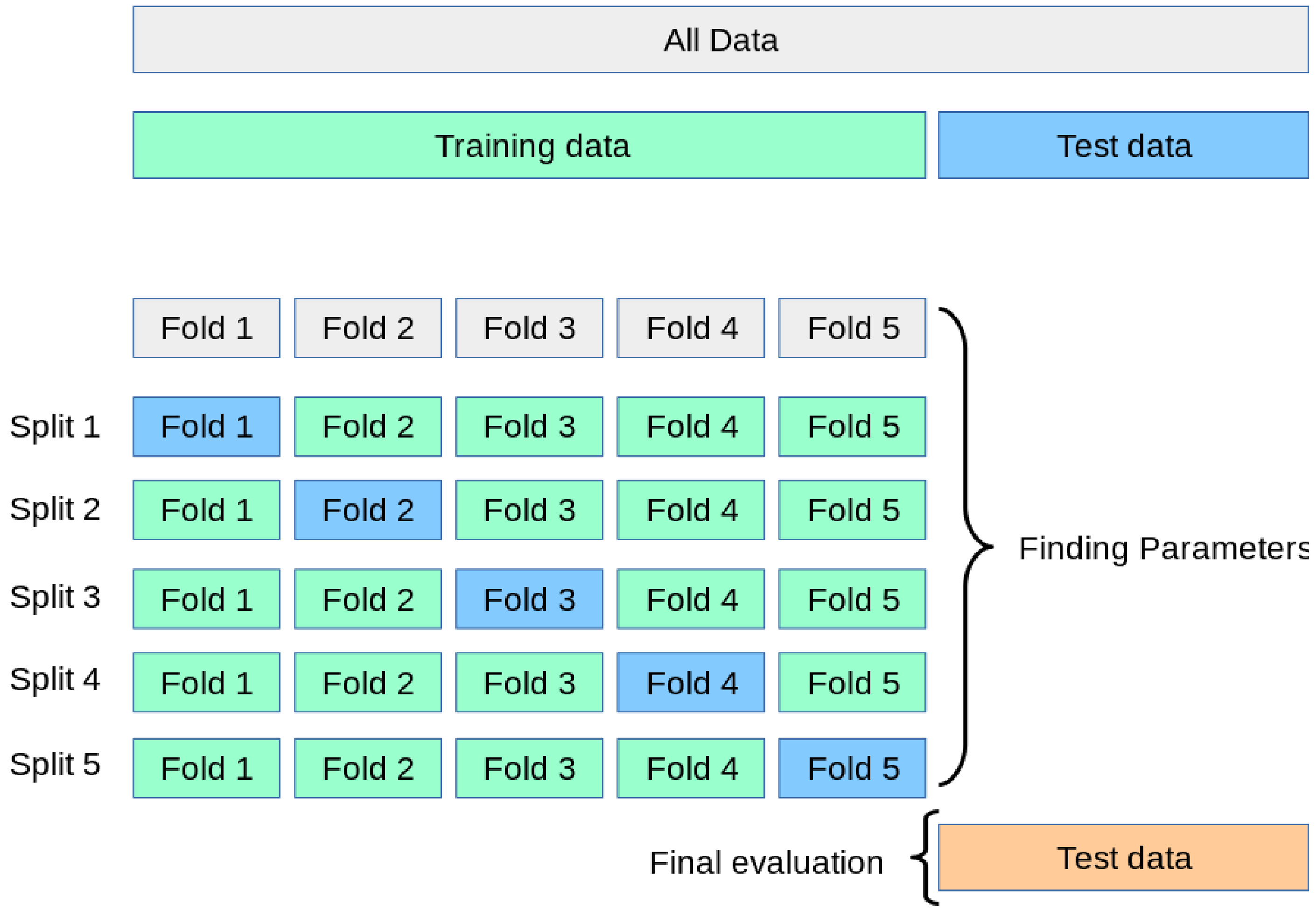


k-Fold Cross-Validation

- Using ***k-fold cross-validation*** for hyper-parameter tuning is common when the size of the training data is small
  - It also 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 (for reporting purposes)
  5. Once the best hyper-parameters are determined, evaluate the model on the test data

k-Fold Cross-Validation

- Illustration of a 5-fold cross-validation



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

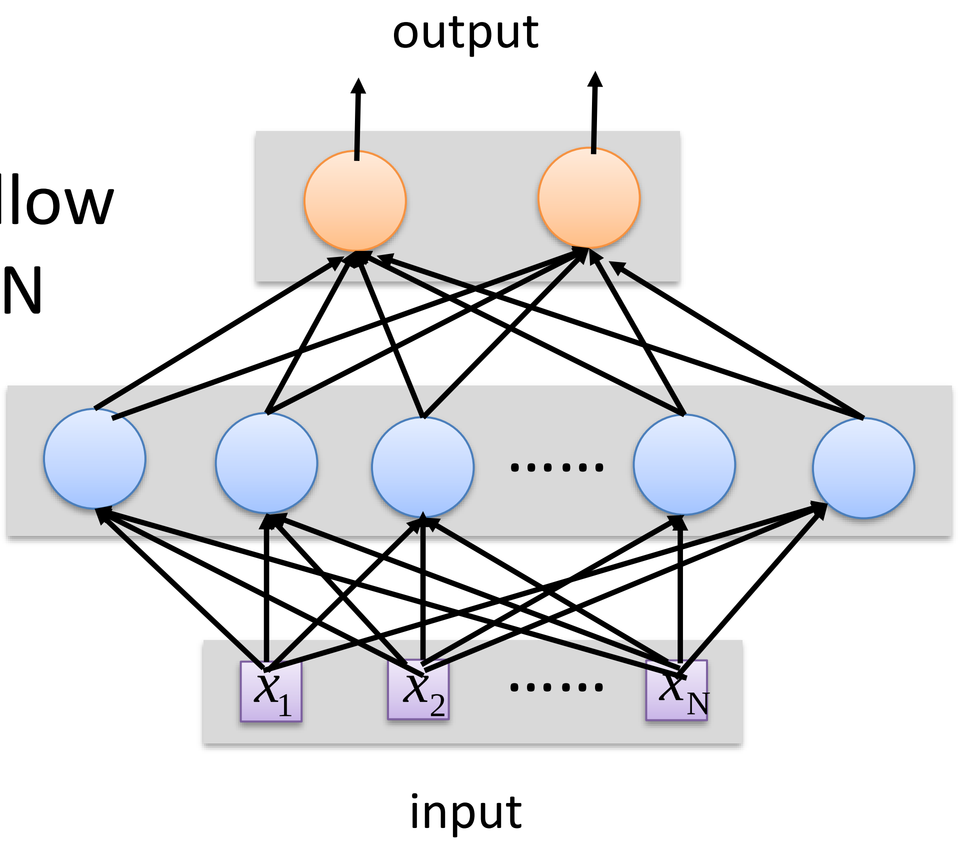
Ensemble Learning

- ***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 **agg**regating)
    - 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

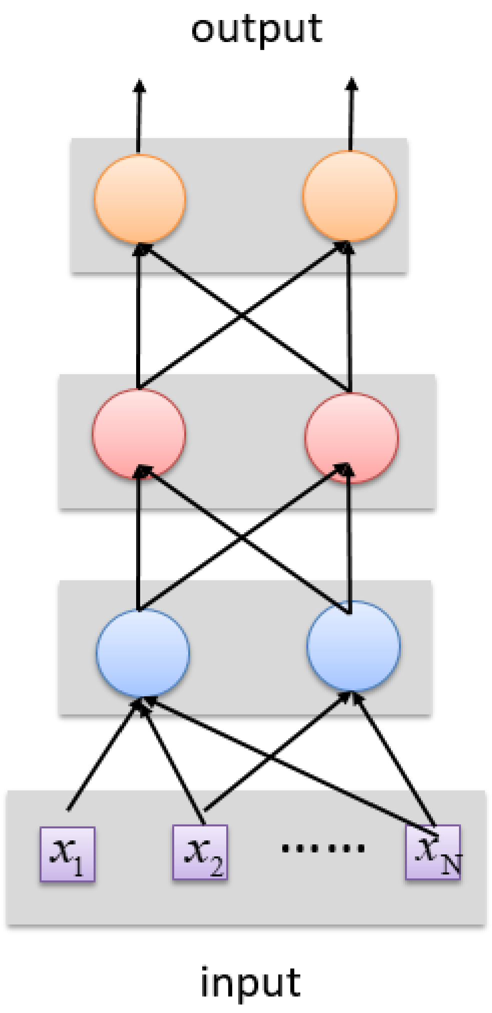
Deep vs Shallow Networks

- 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 NN



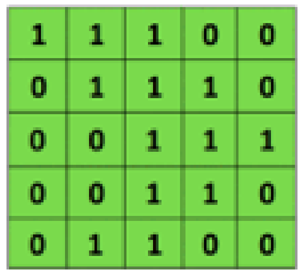
Deep NN



Convolutional Neural Networks (CNNs)

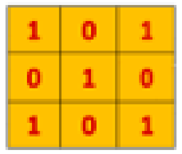
Convolutional Neural Networks

- *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



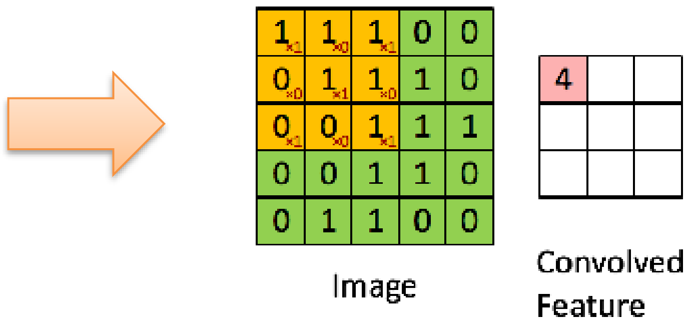
1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input matrix



1	0	1
0	1	0
1	0	1

Convolutional 3x3 filter



$1_{\times1}$ $1_{\times0}$ $1_{\times1}$ 0 0
$0_{\times0}$ $1_{\times1}$ $1_{\times0}$ 1 0
$0_{\times1}$ $0_{\times0}$ $1_{\times1}$ 1 1
0 0 1 1 0
0 1 1 0 0
Image
4
Convolved Feature

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

Convolutional Neural Networks (CNNs)

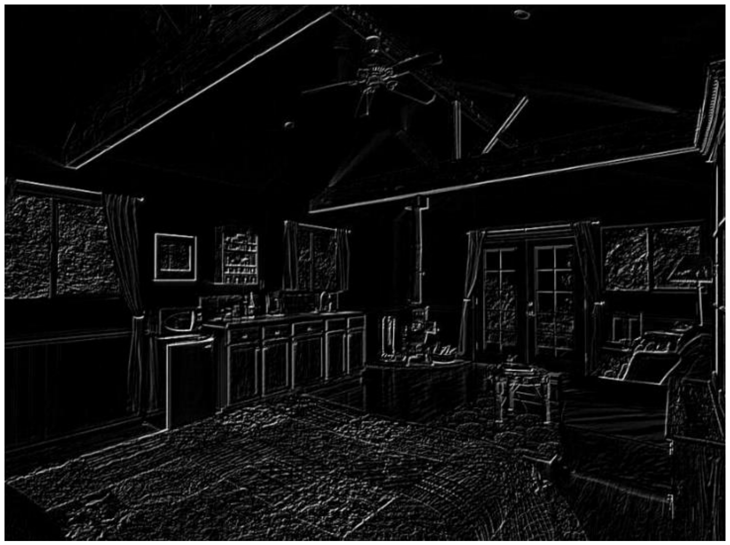
Convolutional Neural Networks

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

Filter

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

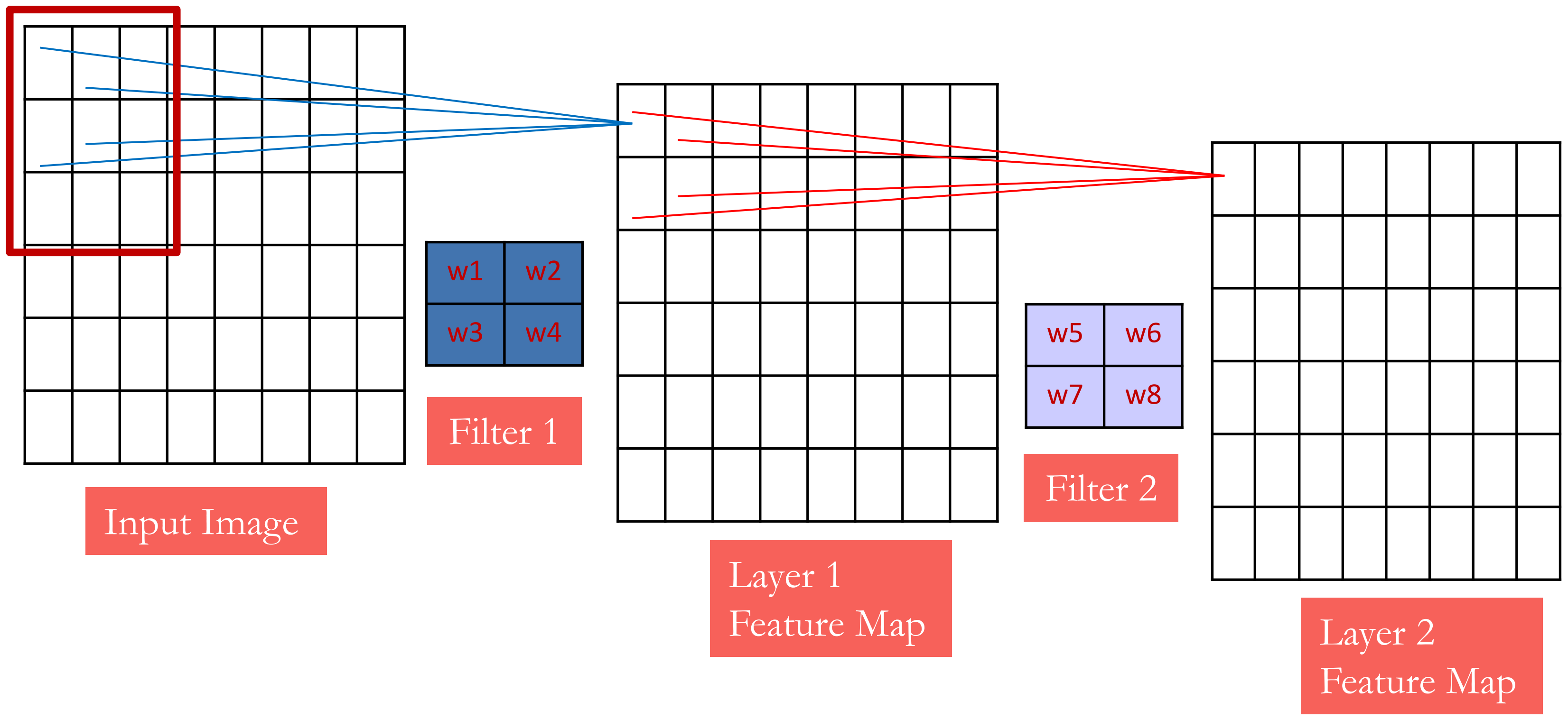
Input Image



Convoluted Image

Convolutional Neural Networks (CNNs)

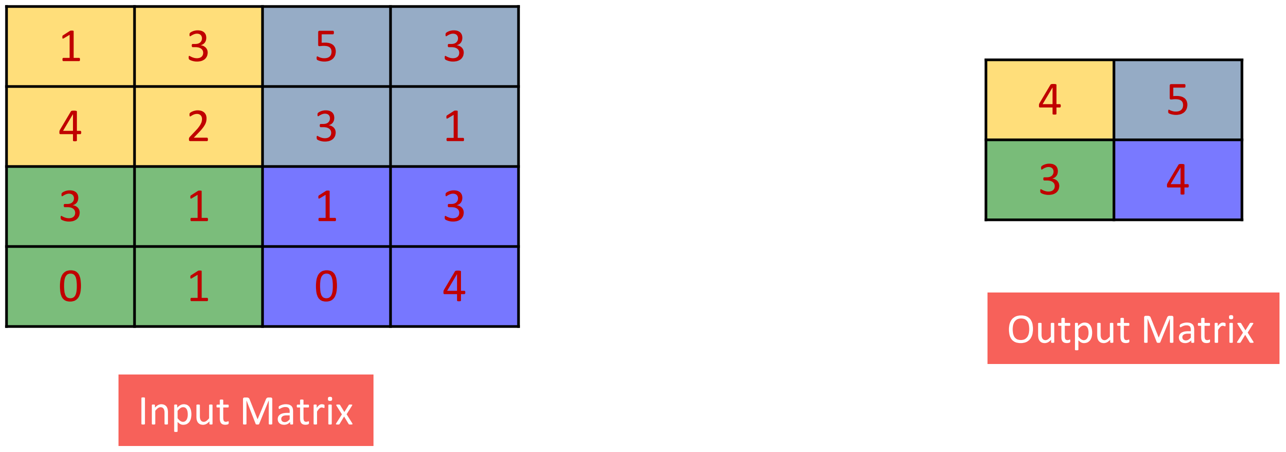
- In CNNs, hidden units in a layer are only connected to a small region of the layer before it (called local receptive field)
  - The depth of each feature map corresponds to the number of convolutional filters used at each layer



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

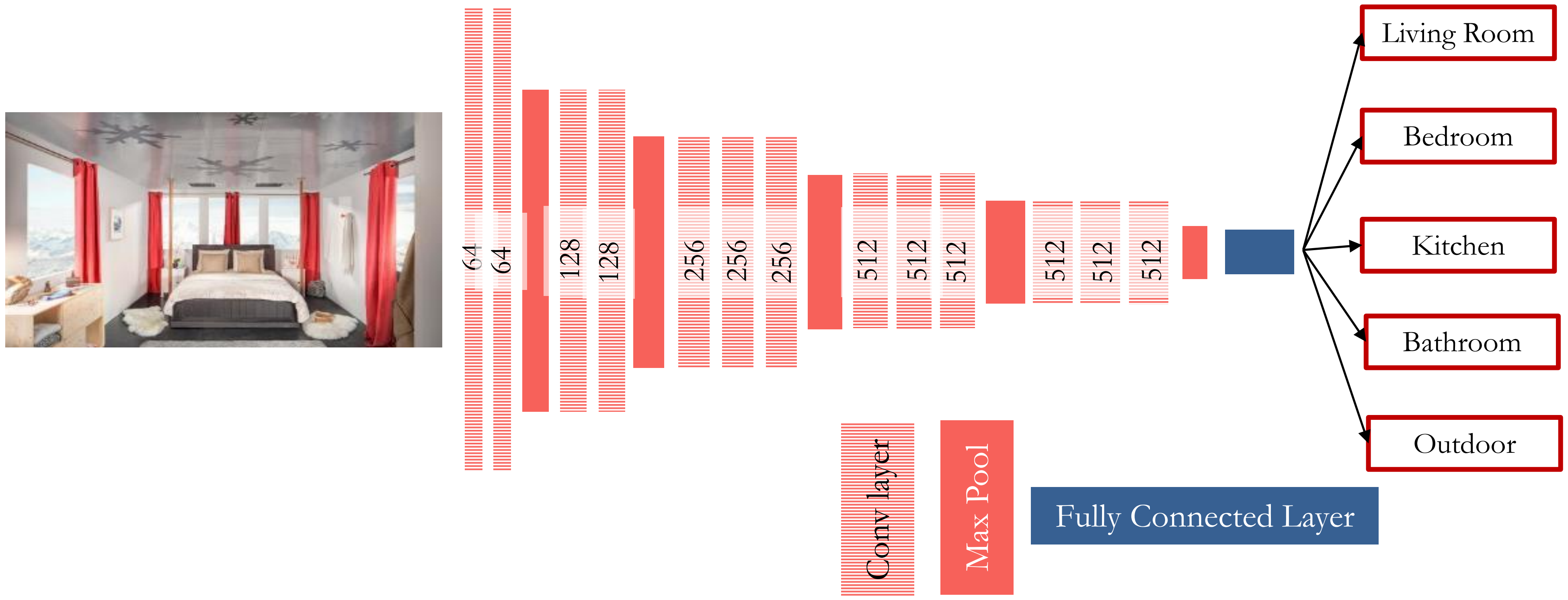
4	5
3	4

Output Matrix

Convolutional Neural Networks (CNNs)

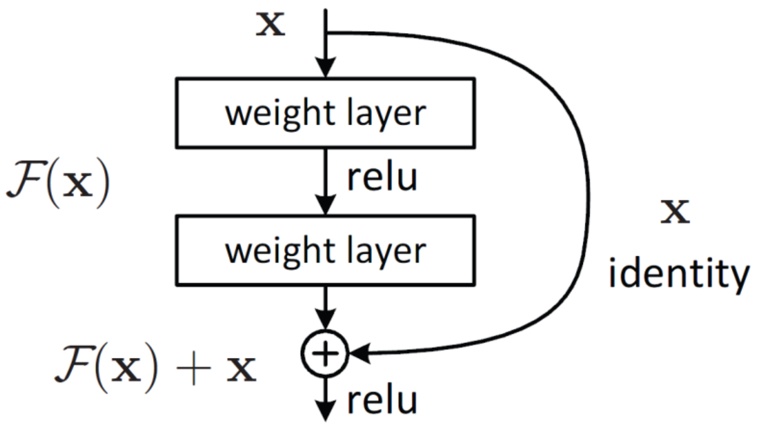
Convolutional Neural Networks

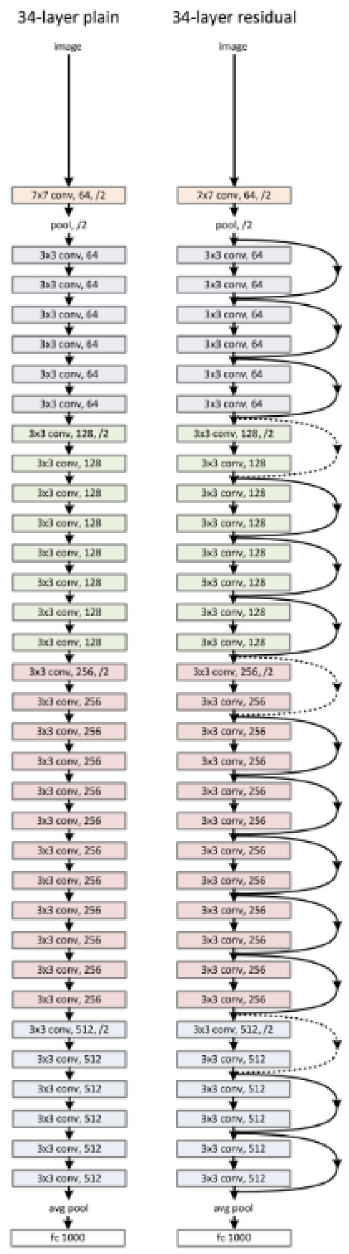
- Feature extraction architecture
  - After 2 convolutional layers, a max-pooling layer reduces the size of the feature maps (typically by 2)
  - A fully convolutional and a softmax layers are added last to perform classification



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 ResNet 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





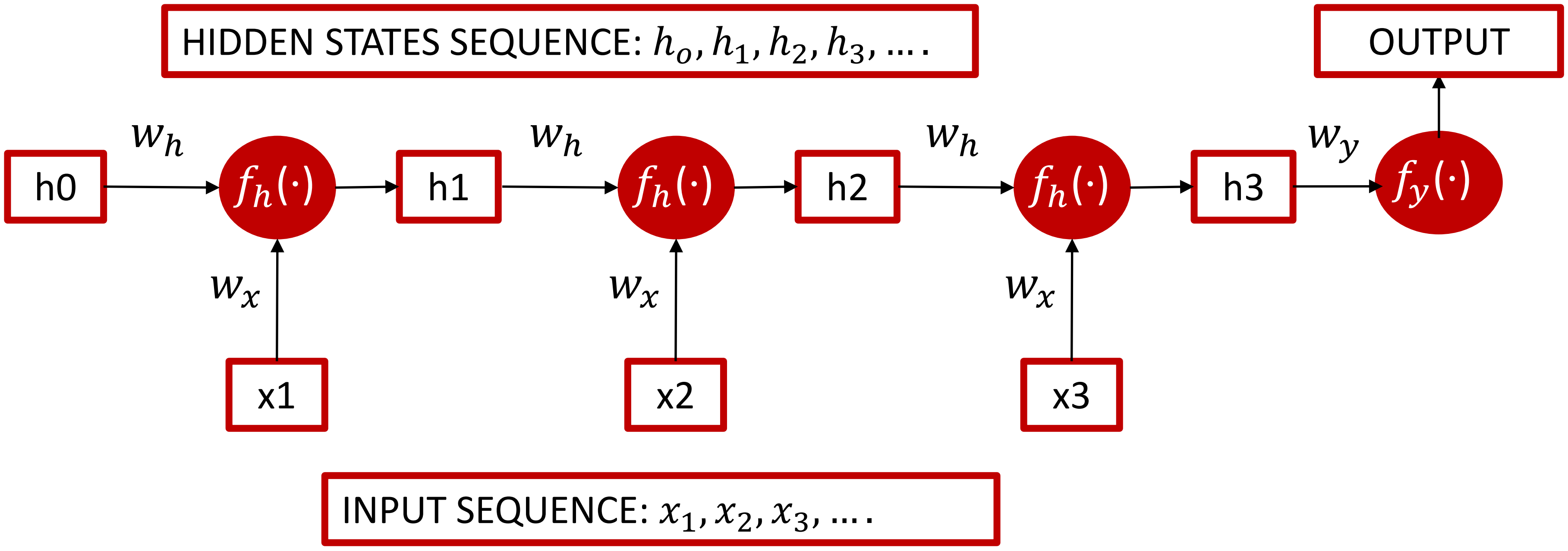
Recurrent Neural Networks (RNNs)

- ***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
- RNN variants:
  - ***Basic (Vanilla) RNN networks***
    - Are sensitive to the vanishing gradient problem
  - ***Long Short-Term Memory (LSTM)*** networks
    - 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
  - ***Gated Recurrent Networks (GRU)***
    - Similar to LSTM, less commonly used than LSTM

Recurrent Neural Networks (RNNs)

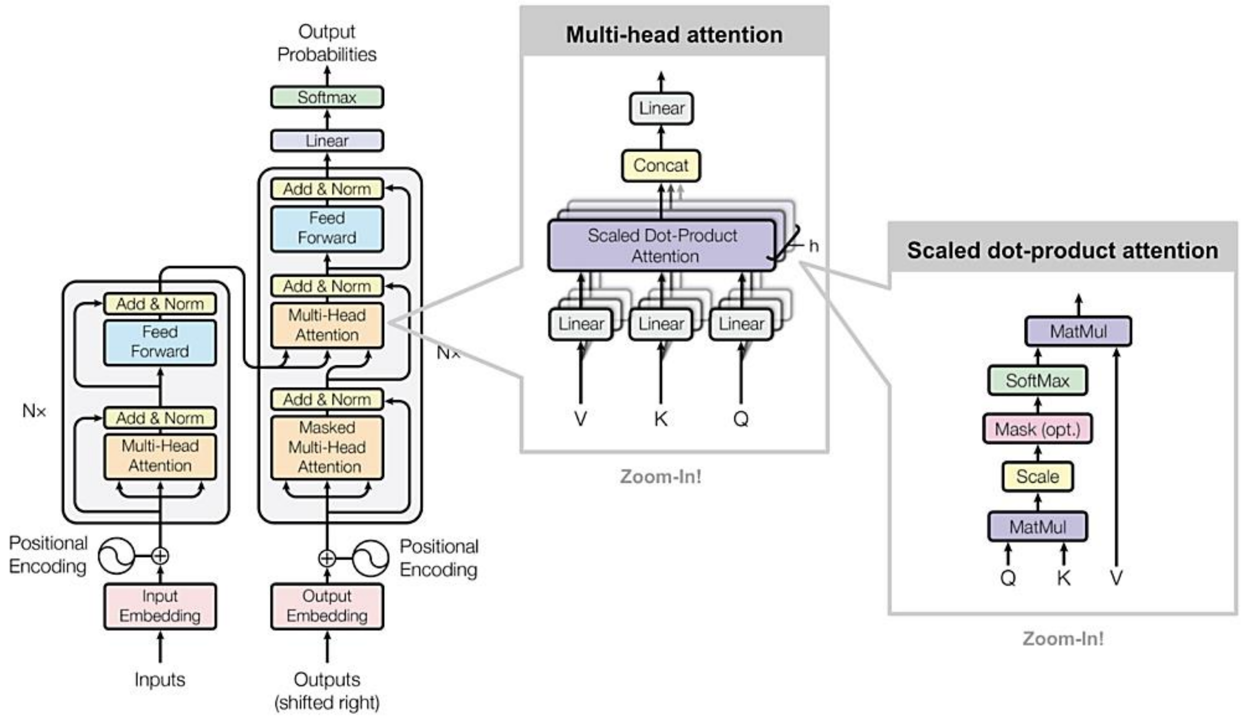
Recurrent Neural Networks

- RNN use same set of weights w_h and w_x across all time steps
  - A sequence of hidden states $\{h_o, h_o h_2, h_3, \dots\}$ is learned, which represents the memory of the network
  - The hidden state at step t, $h(t)$, is calculated based on the previous hidden state $h(t-1)$ and the input at the current step $x(t)$, i.e., $h(t) = f_h\big(w_h * h(t-1) + w_x * x(t)\big)$
  - The function $f_h(\cdot)$ is a nonlinear activation function, e.g., ReLU or tanh
- RNN shown rolled over time



Transformer Networks

- *Transformer networks* have been initially designed for processing test data in Large Language Models, such as GPT-3
  - Later, they have been used for image tasks, and tabular data processing
- The main block of transformers is the self-attention mechanism, which uses scaled dot-product attention to force the model to attend to portions of the data
  - Several self-attention modules are combined into a multi-head attention layer



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)