Introduction to Deep Learning Models

Tanujit Chakraborty

@ Sorbonne

Webpage: https://www.ctanujit.org

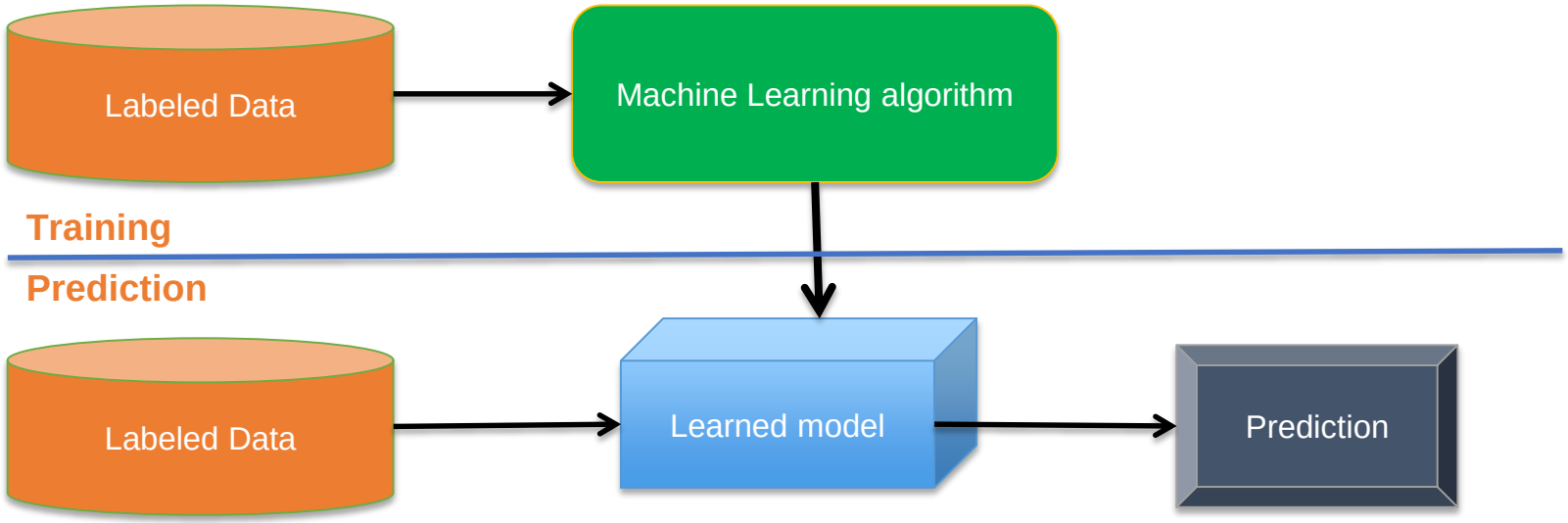
Lecture Outline

- 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
  - LSTM
  - Transformers
  - GAN

* Most of the slides are adapted from DL and D2L Book and several online resources and slides that are acknowledged at the end of the presentation.

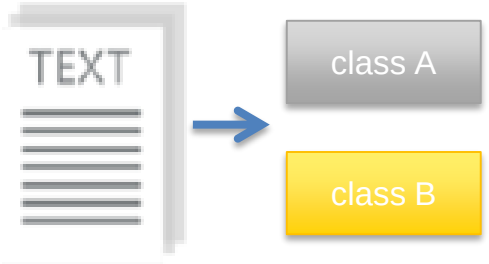
Machine Learning Basics

- ***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

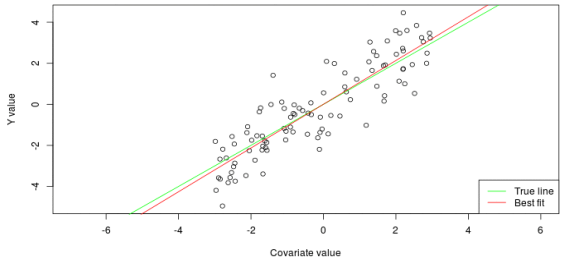


Machine Learning Types

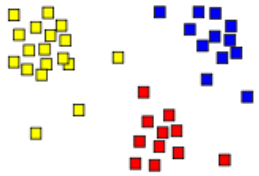
- *Supervised*: learning with labeled data
  - Example: email classification, image classification
  - Example: regression for predicting real-valued outputs
- *Unsupervised*: discover patterns in unlabeled data
  - Example: cluster similar data points
- *Reinforcement learning*: learn to act based on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



Classification



Regression

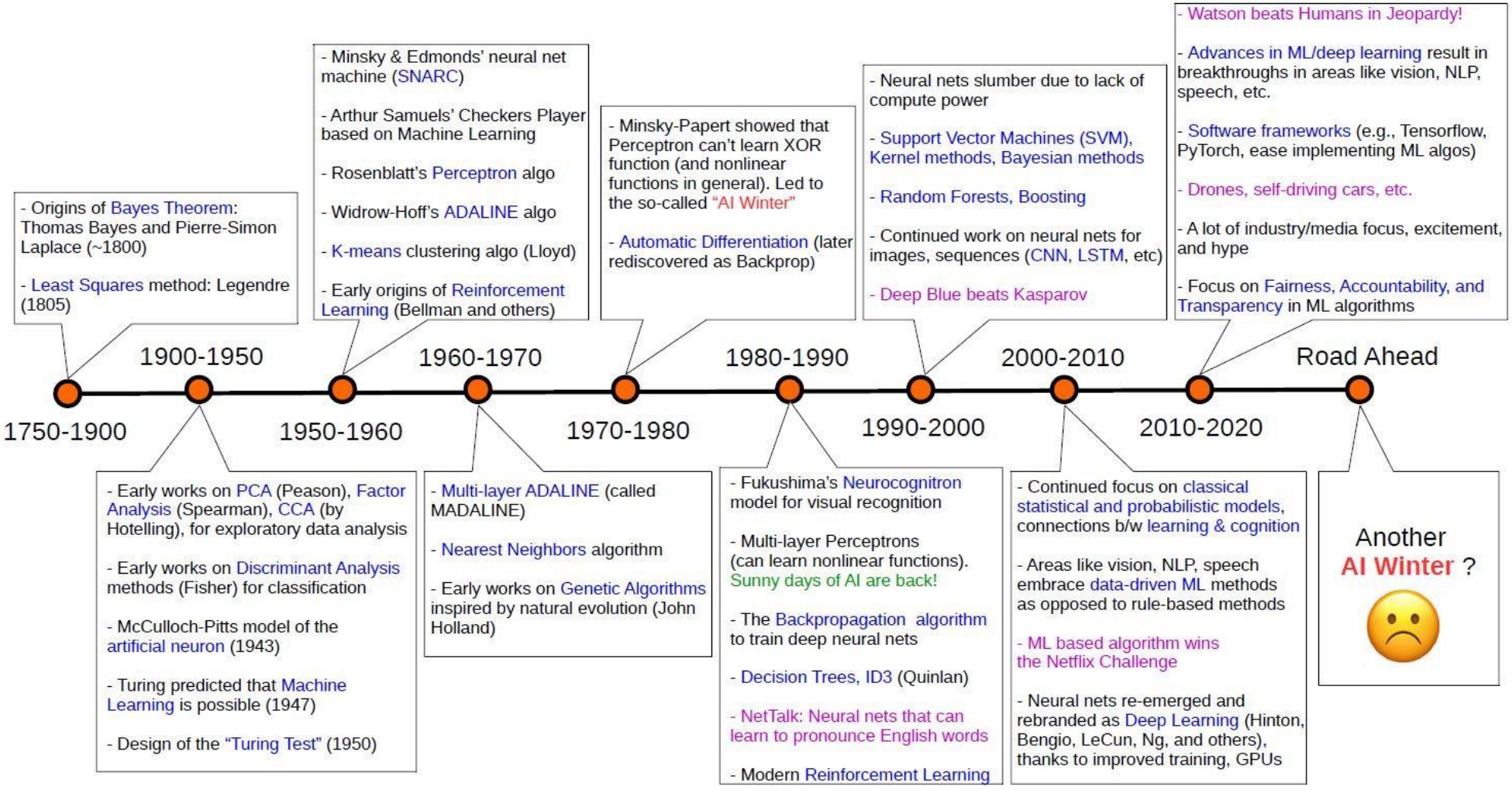

Clustering

Supervised and Unsupervised 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

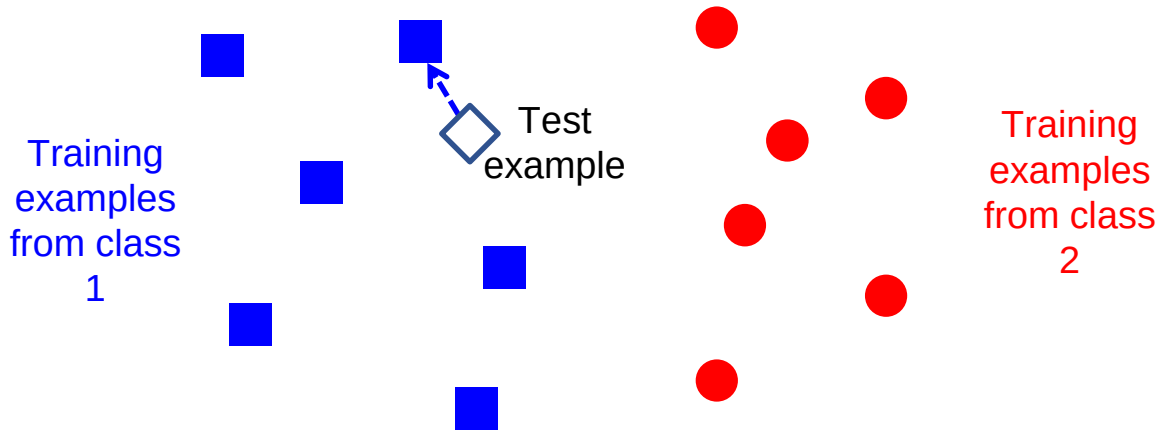
- *Unsupervised learning* categories and techniques
  - **Clustering**
    - *k*-means clustering
    - Mean-shift clustering
    - Spectral clustering
  - **Density estimation**
    - Gaussian mixture model (GMM)
    - Graphical models
  - **Dimensionality reduction**
    - Principal component analysis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

Machine Learning Timeline



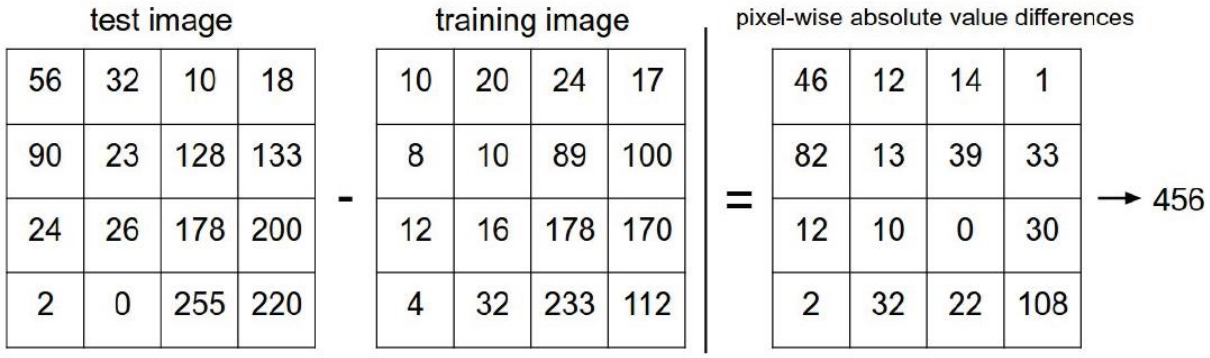
Nearest Neighbor Classifier

- *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)
  - It does not require learning a set of weights



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 CIFAR-10: 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



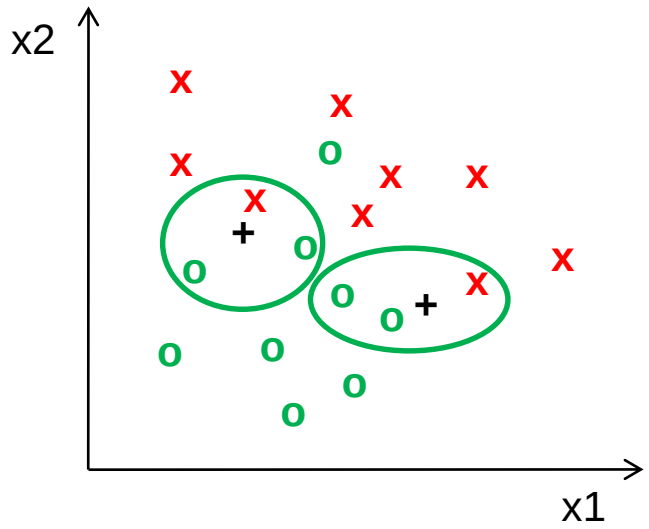
ℓ_1 norm
(Manhattan distance)

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

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

k-Nearest Neighbors 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 (based on the distance to the 3 closest points)



Linear Classifier

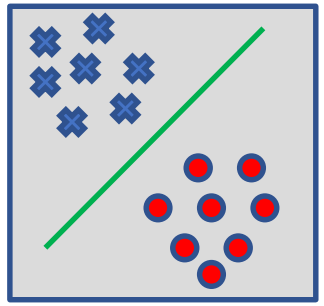
- ***Linear classifier***
  - Find a linear function f of the inputs x_i that separates the classes

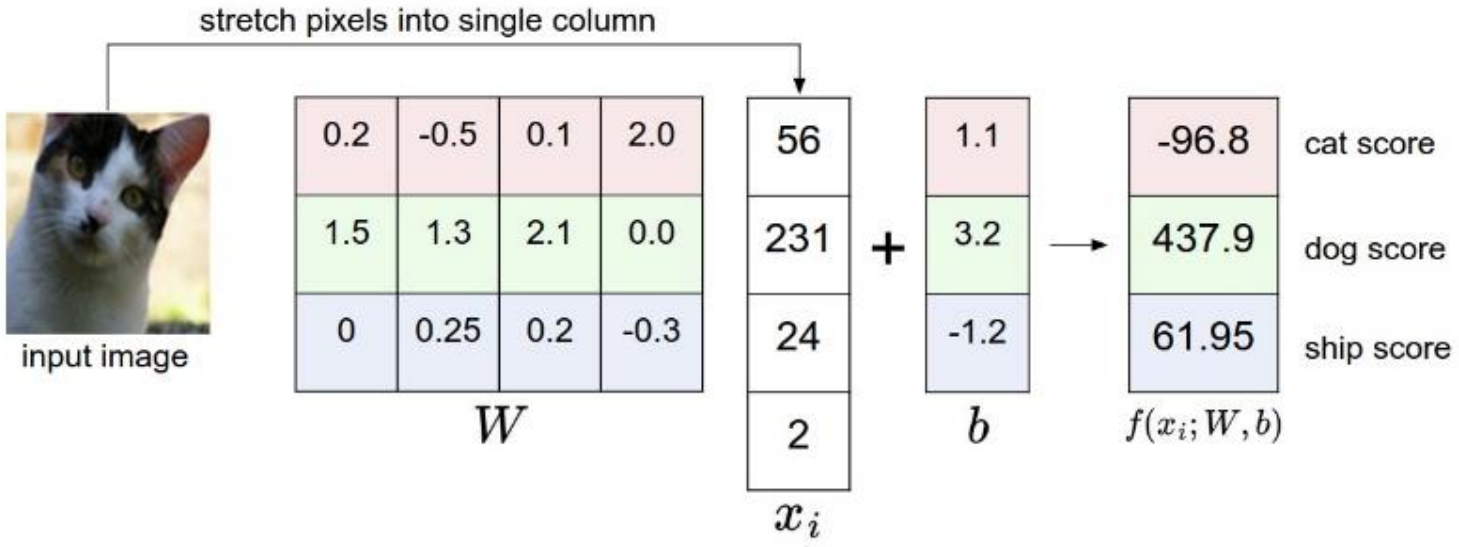
$$f(x_i, W, b) = Wx_i + b$$

  - 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
  - Several methods have been used to find the optimal set of parameters of a linear classifier
    - A common method of choice is the ***Perceptron*** algorithm, where the parameters are updated until a minimal error is reached (single layer, does not use backpropagation)
  - Linear classifier is a simple approach, but it is a building block of advanced classification algorithms, such as SVM and neural networks
    - Earlier multi-layer neural networks were referred to as multi-layer perceptrons (MLPs)

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





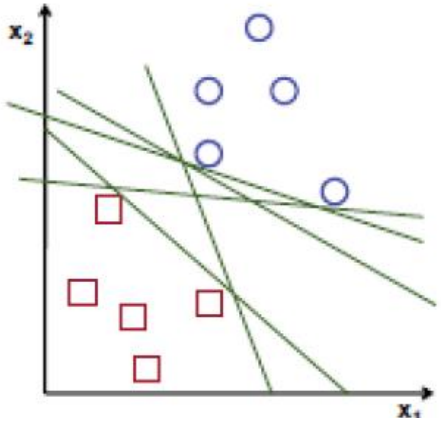
Picture from: https://cs231n.github.io/classification/

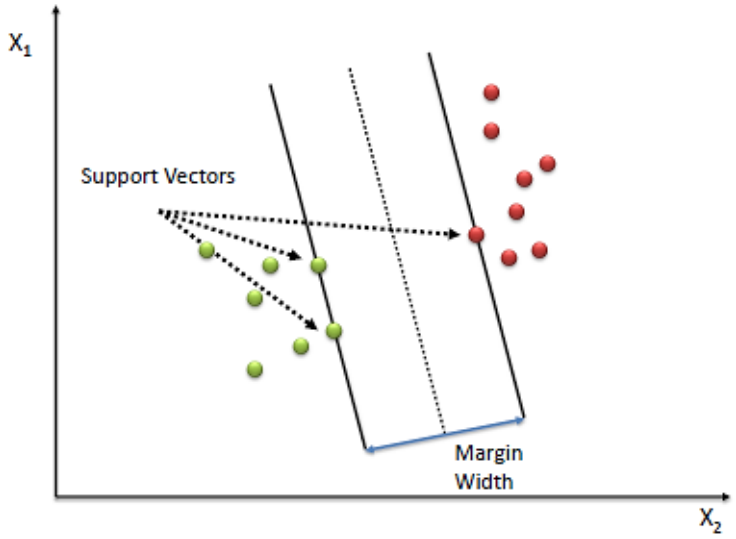
Support Vector Machines

- *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 the 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:

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

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

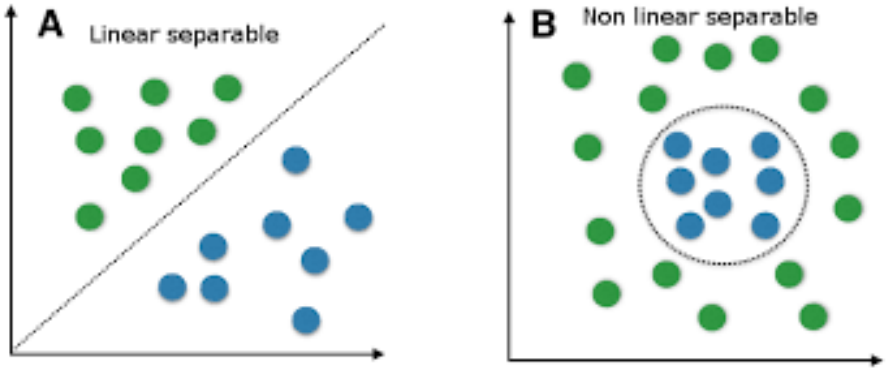




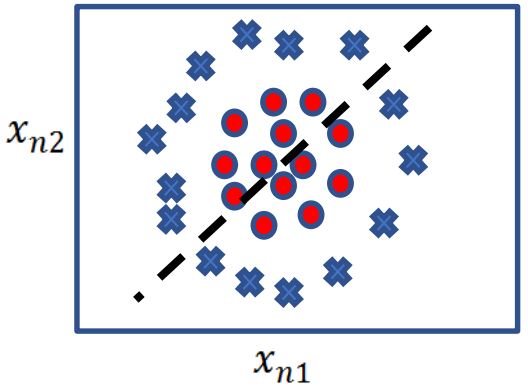
Linear vs Non-linear Techniques

- 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

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

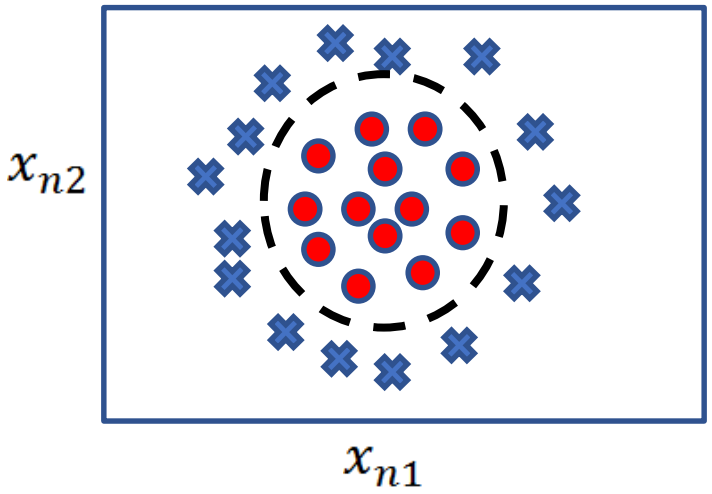


- For other tasks, linear classifiers may have difficulties to produce adequate decision boundaries



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



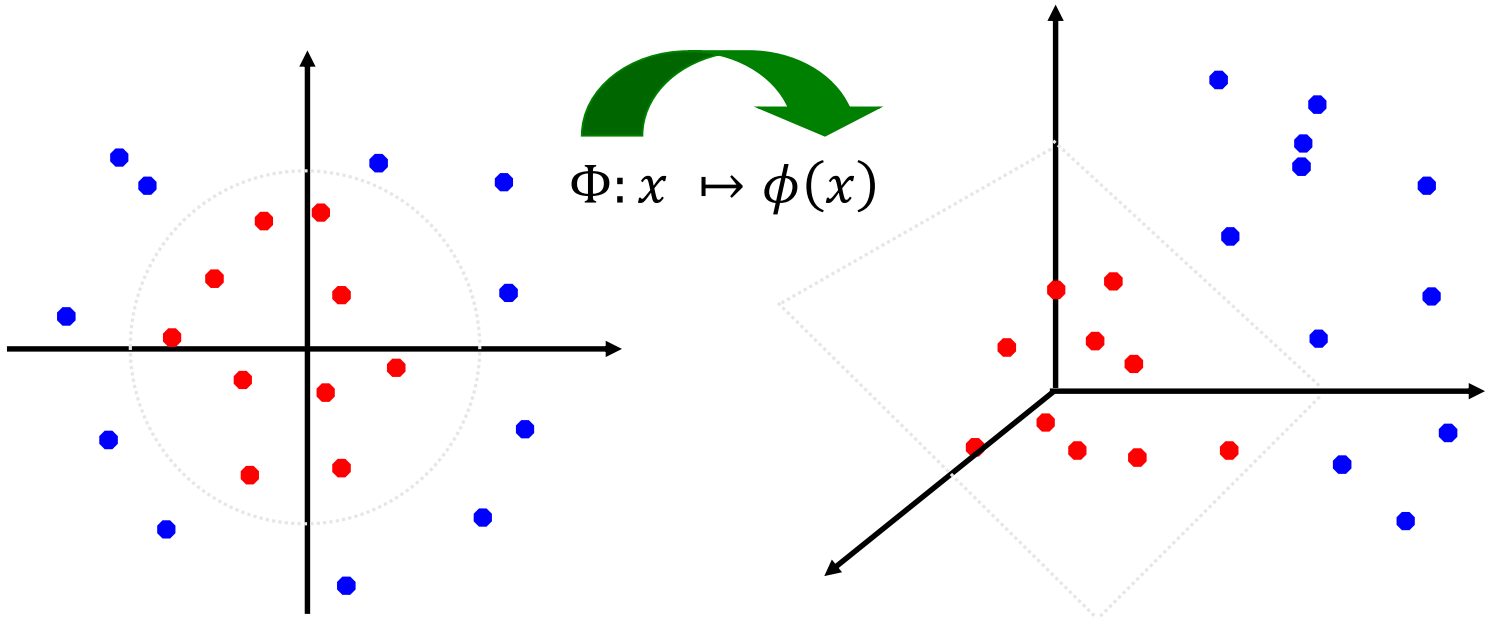
Inputs: $x_i = [x_{n1} \quad x_{n2}]$

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

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

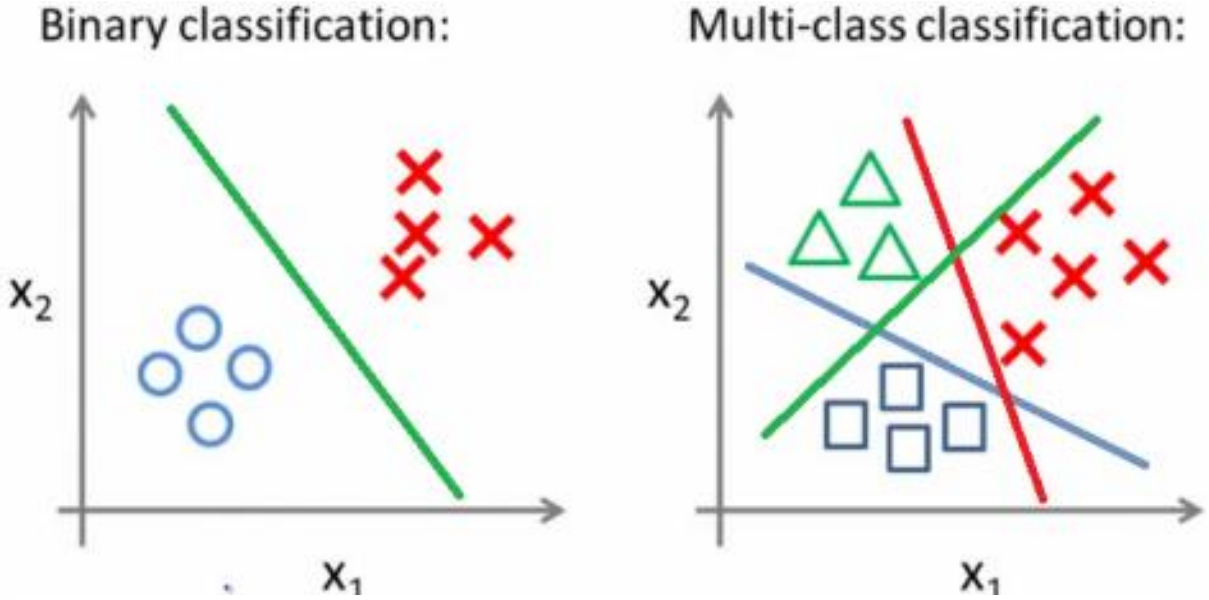
Non-linear Support Vector Machines

- ***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

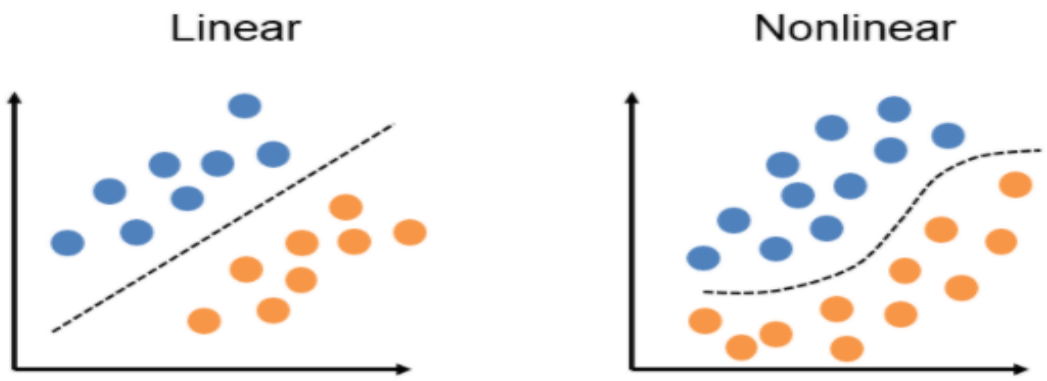


Binary vs Multi-class Classification

- 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***



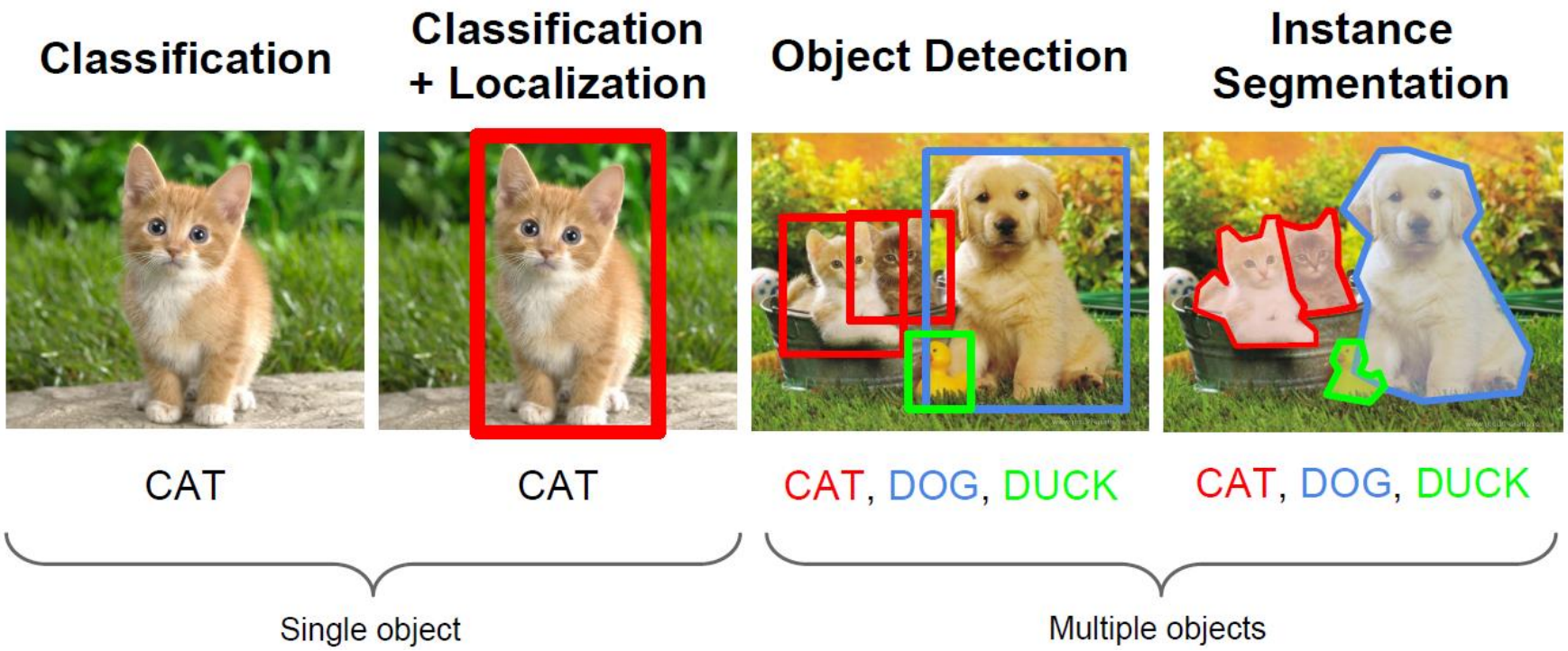
- Both the binary and multi-class classification problems can be linearly or non-linearly separated



  - Figure: linearly and non-linearly separated data for binary classification problem

Computer Vision Tasks

- Computer vision has been the primary area of interest for ML
- The tasks include: classification, localization, object detection, instance segmentation



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 ML model parameters from limited data

- In summary, ***No-Free-Lunch Theorem*** states:
  - **No single classifier works the best for all possible problems**
  - Since we need to make assumptions to generalize

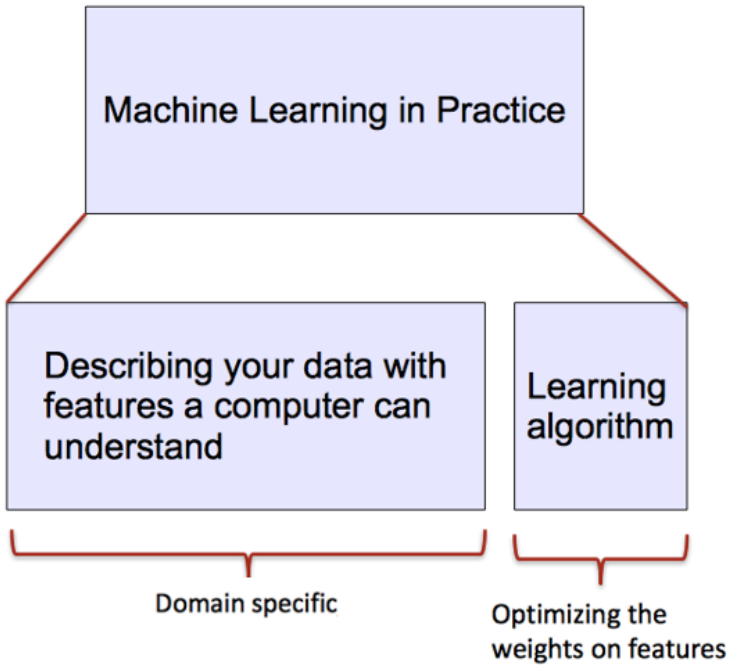
Why is DL Useful?

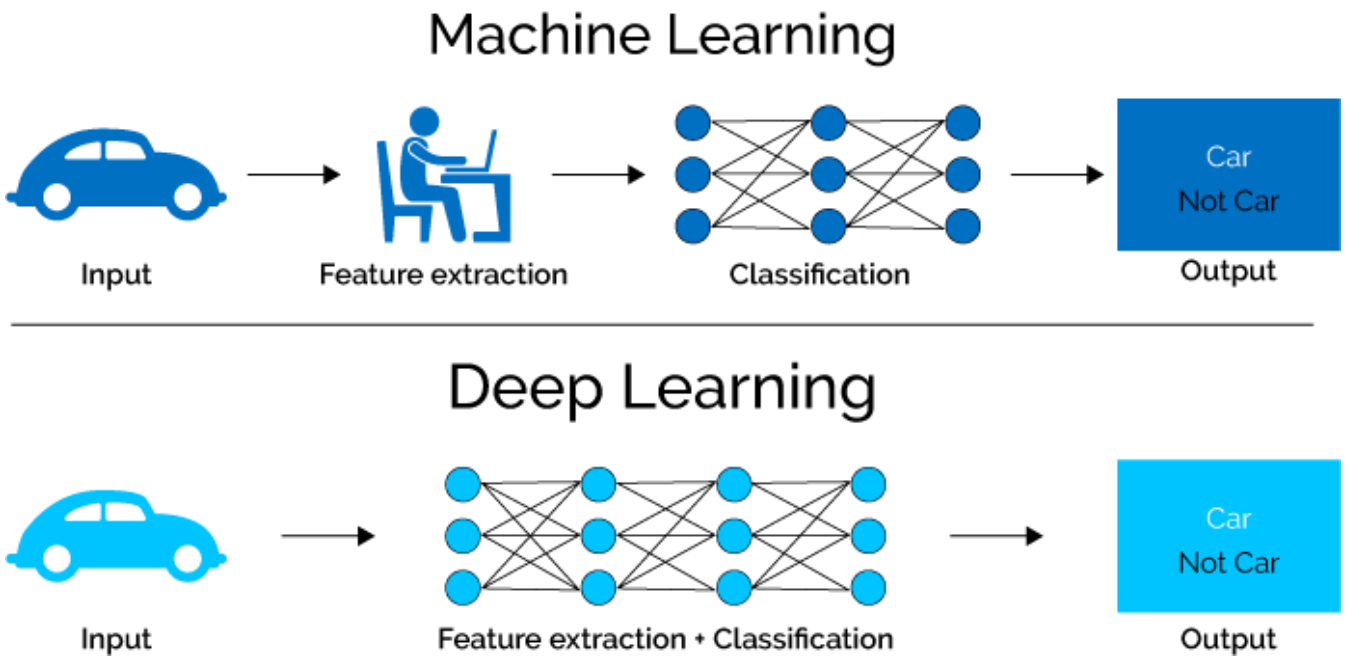
- DL provides a flexible, learnable framework for representing visual, text, linguistic information
  - Can learn in supervised and unsupervised manner
- DL represents an effective end-to-end learning system
- Requires large amounts of training data
- Since about 2010, DL has outperformed other ML techniques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

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

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

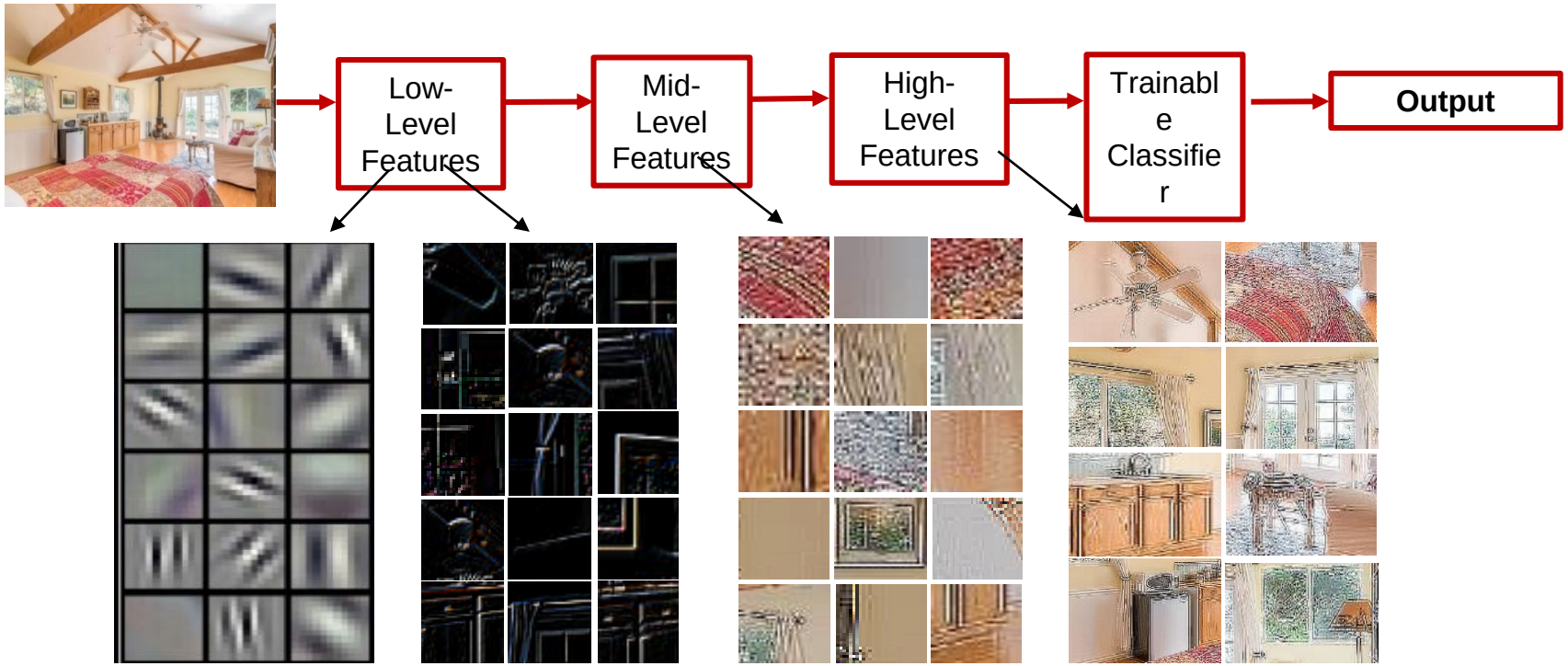




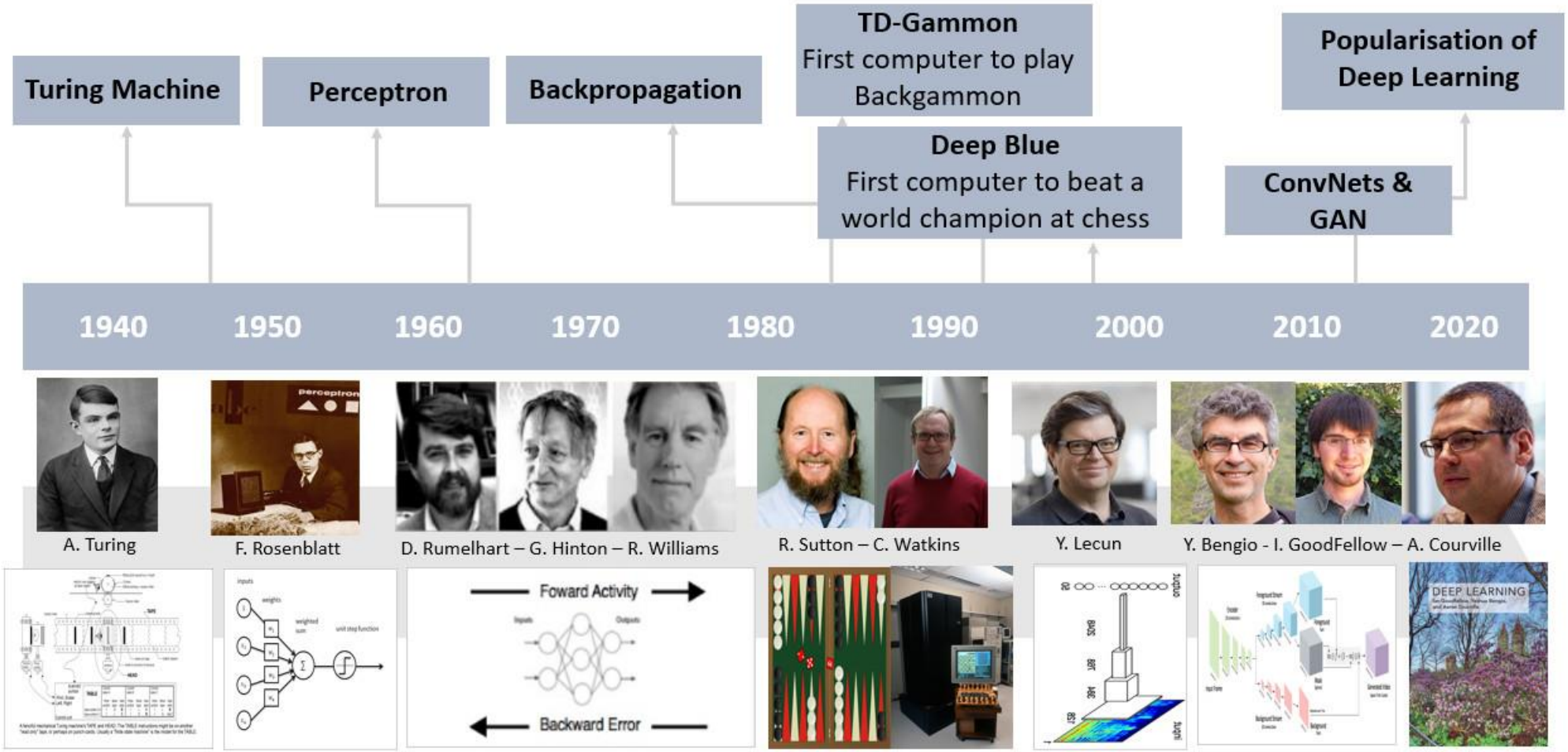
Picture from: https://www.xenonstack.com/blog/static/public/uploads/media/machine-learning-vs-deep-learning.png

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



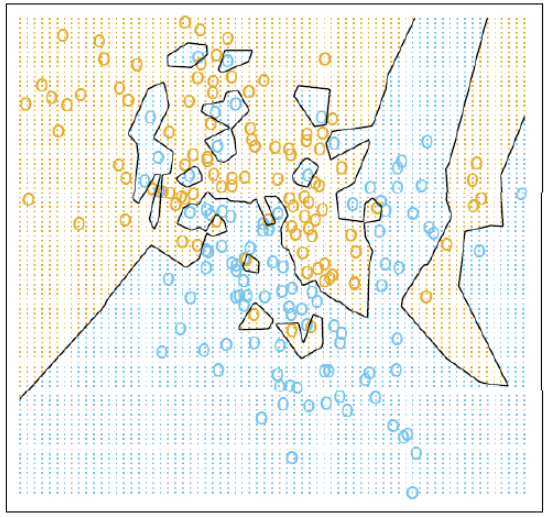
Deep Learning Timeline



Representational Power

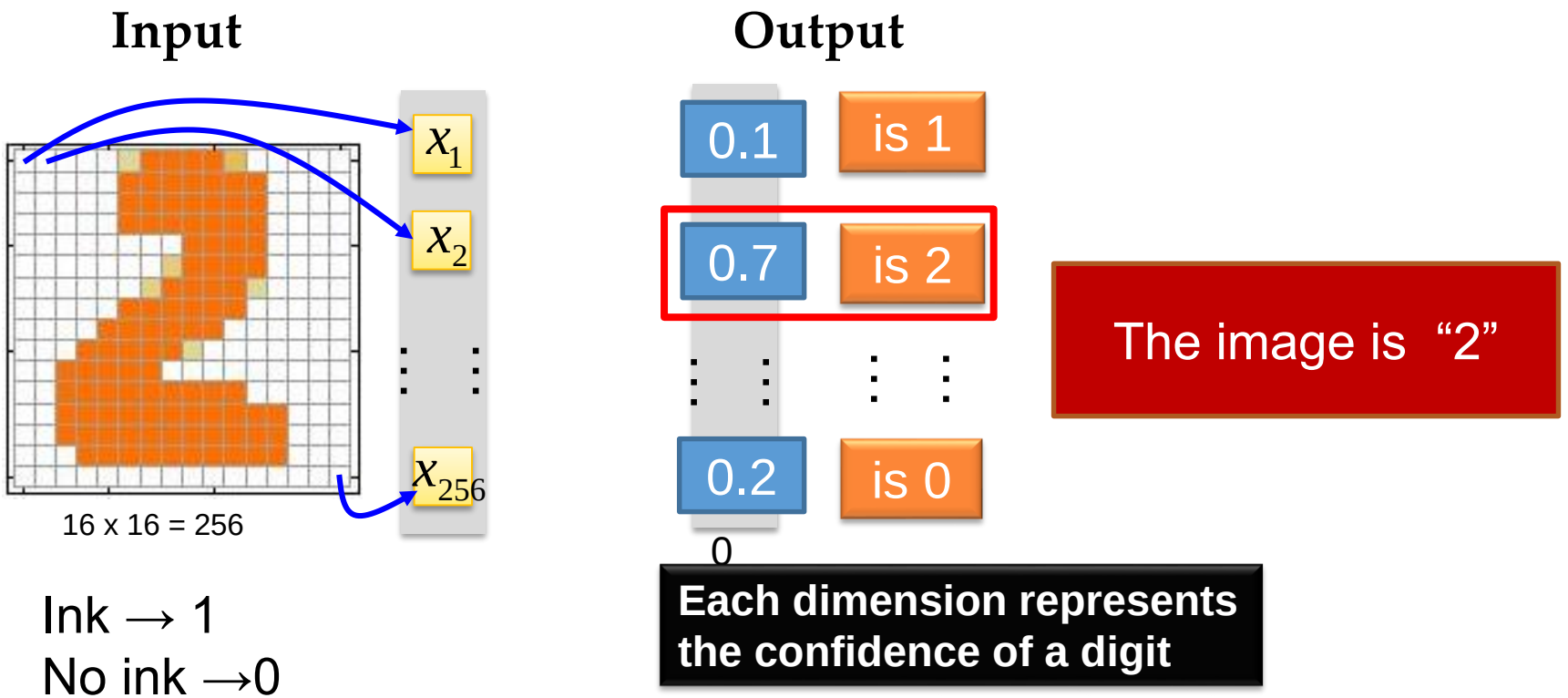
- NNs with at least one hidden layer are universal approximators
  - Given any continuous function $h(x)$ and some $\epsilon > 0$, there exists a NN with one hidden layer (and with a reasonable choice of non-linearity) described with the function $f(x)$, such that $\forall x, |h(x) - f(x)| < \epsilon$
  - I.e., NN can approximate any arbitrary complex continuous function

- NNs use nonlinear mapping of the inputs x 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



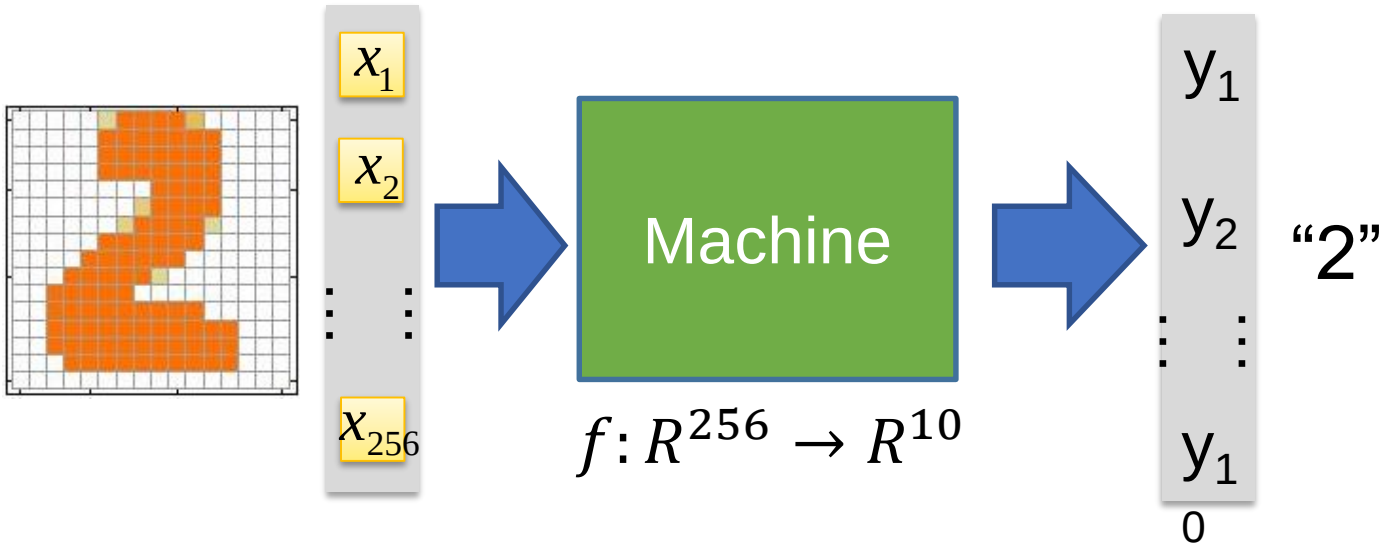
Introduction to Neural Networks

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



Introduction to Neural Networks

- Handwritten digit recognition



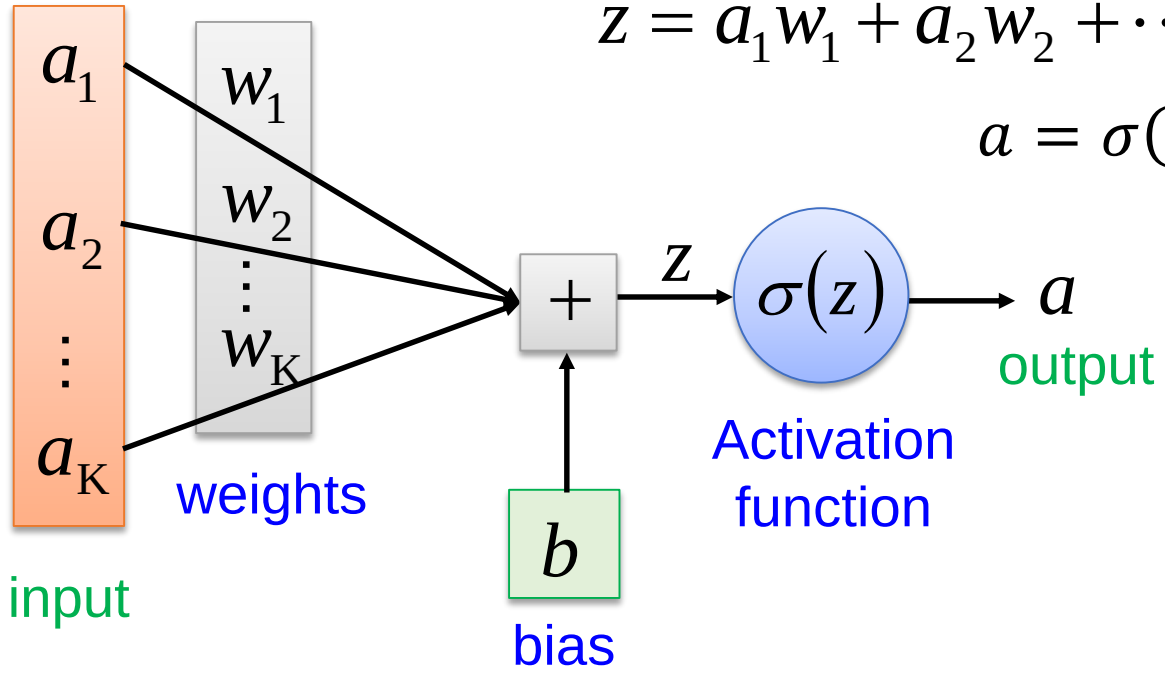
The function f is represented by a neural network

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 $f: R^K \rightarrow R$

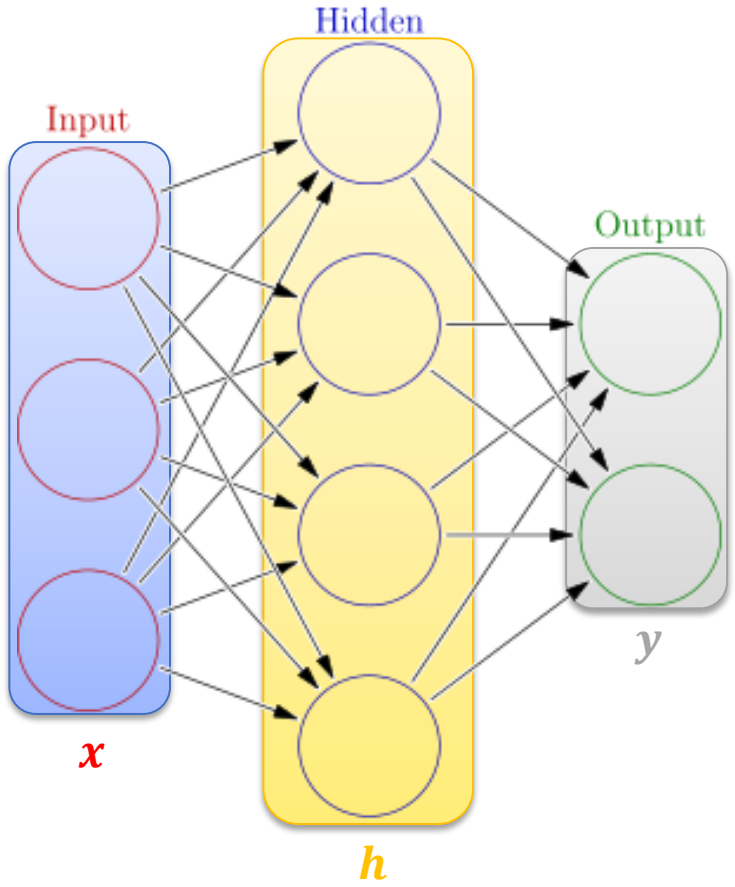
$$z = a_1 w_1 + a_2 w_2 + \cdots + a_K w_K + b$$

$$a = \sigma(z)$$



Elements of Neural Networks

- A NN with one hidden layer and one output layer



Weights Biases

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

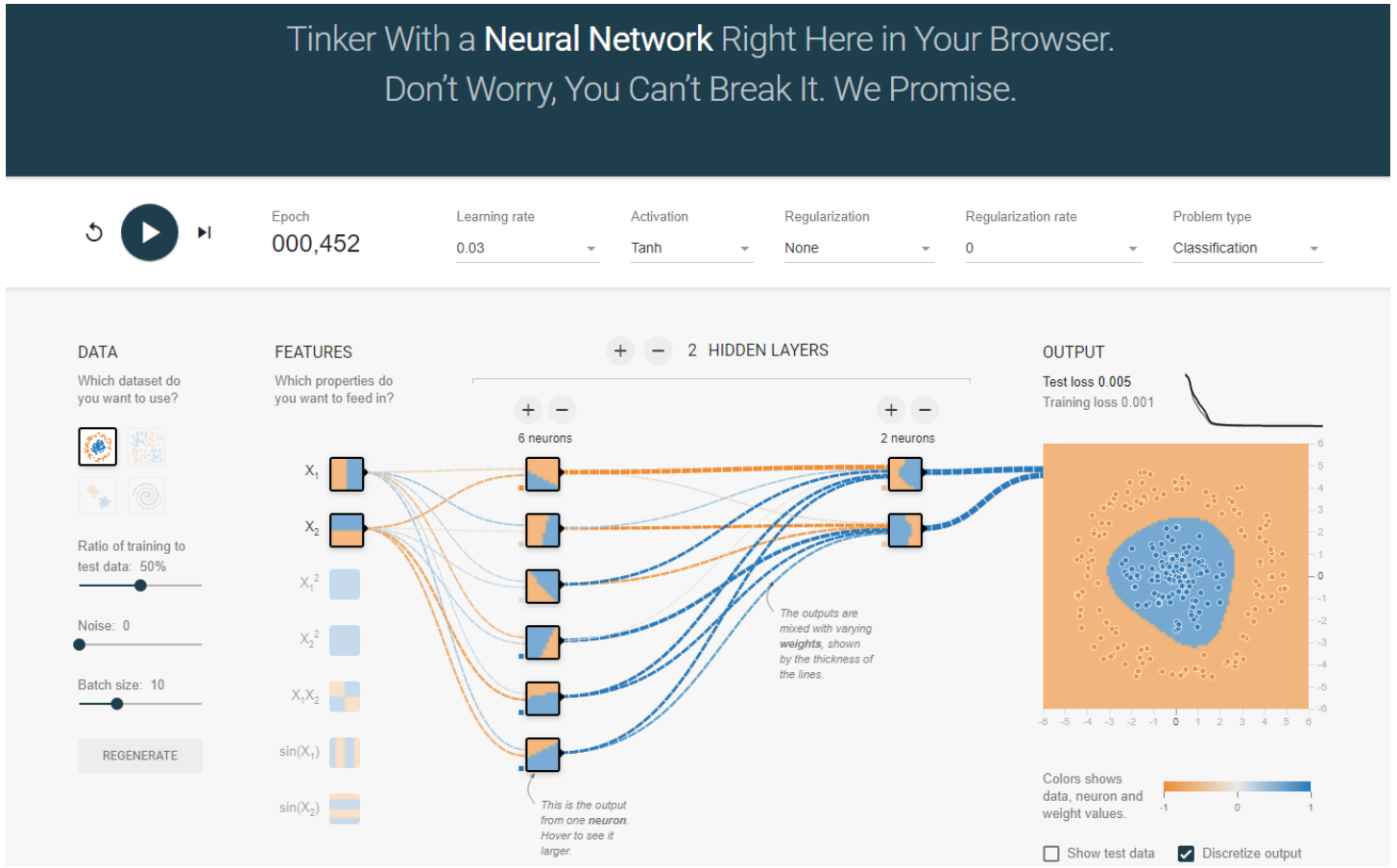
$$output\ layer\ y = \sigma(W_2 h + b_2)$$

Activation functions

4 + 2 = 6 neurons (not counting inputs)
[3 × 4] + [4 × 2] = 20 weights
4 + 2 = 6 biases
26 learnable parameters

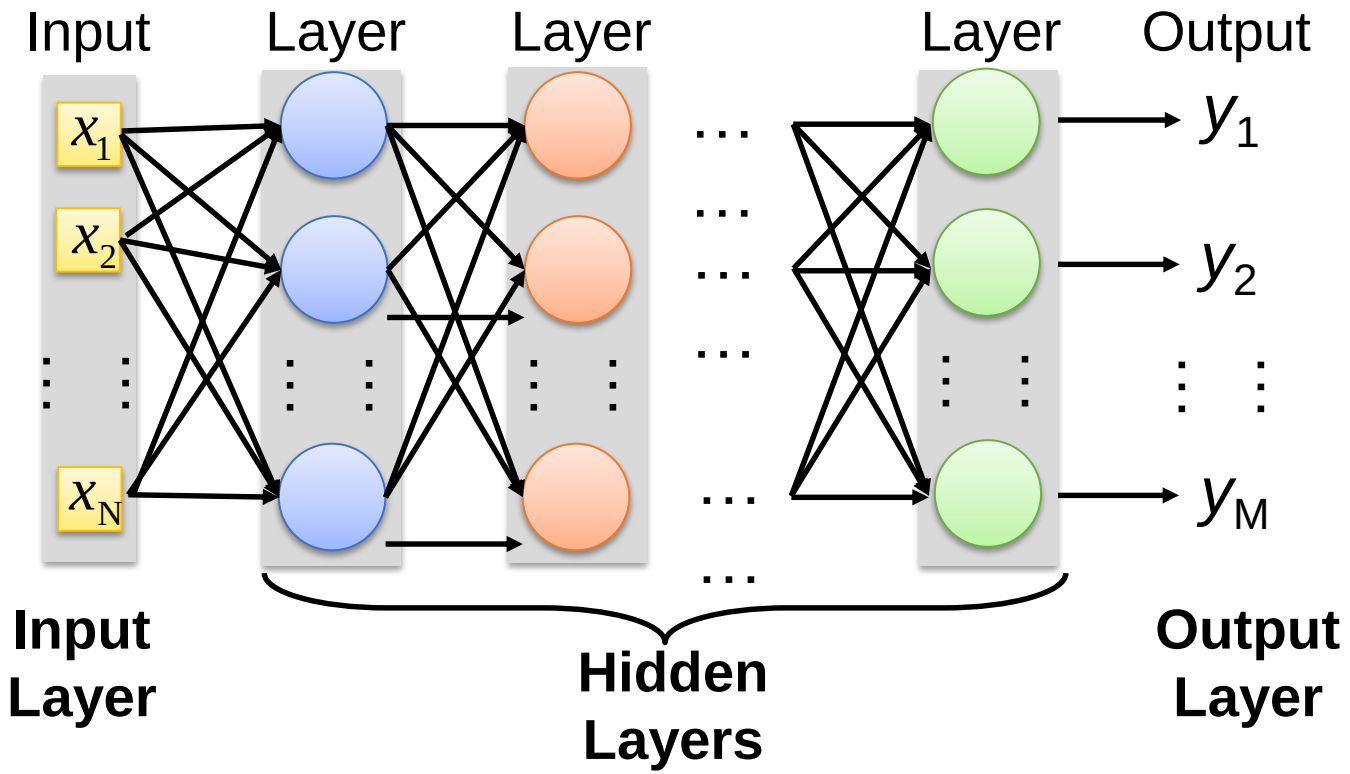
Elements of Neural Networks

- A neural network playground [link]



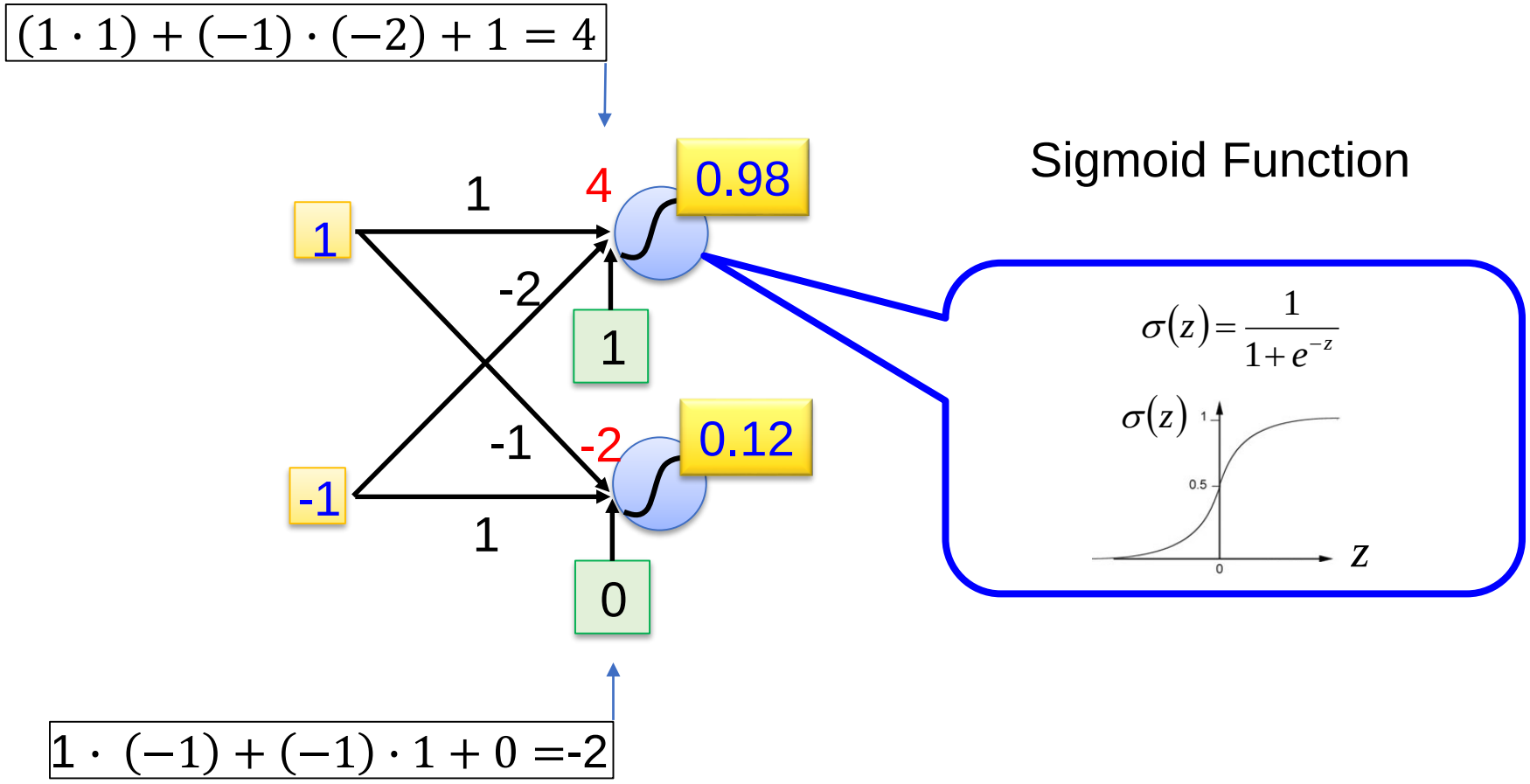
Elements of Neural Networks

- 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



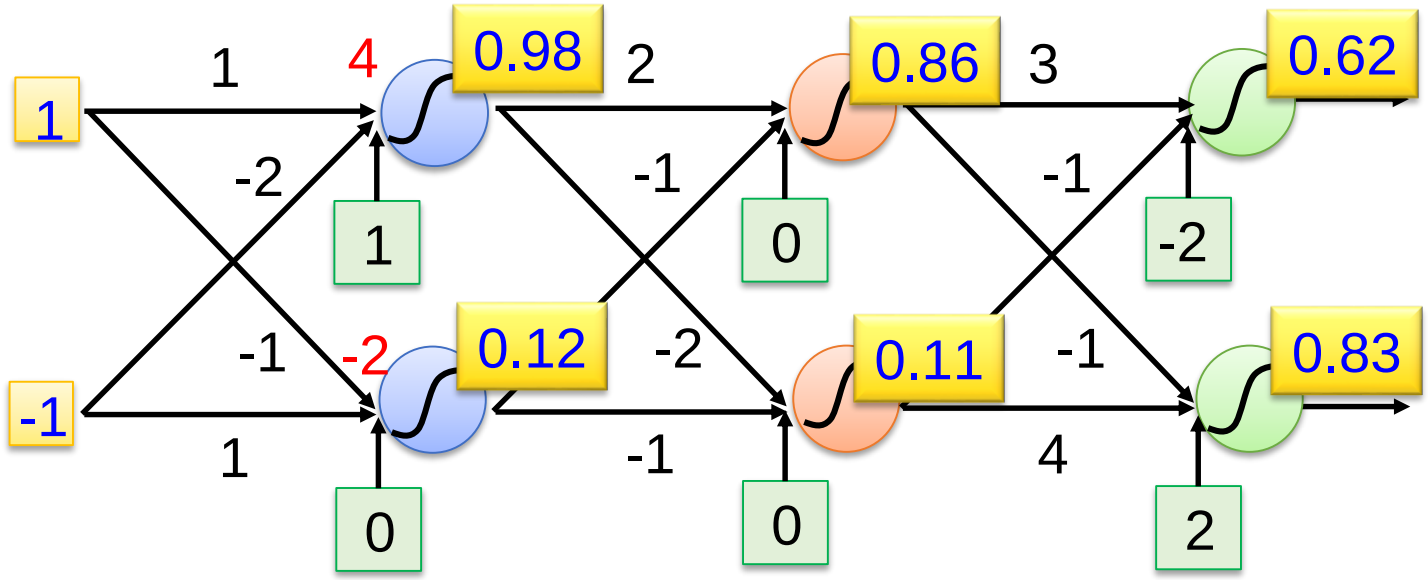
Elements of Neural Networks

- A simple network, toy example



Elements of Neural Networks

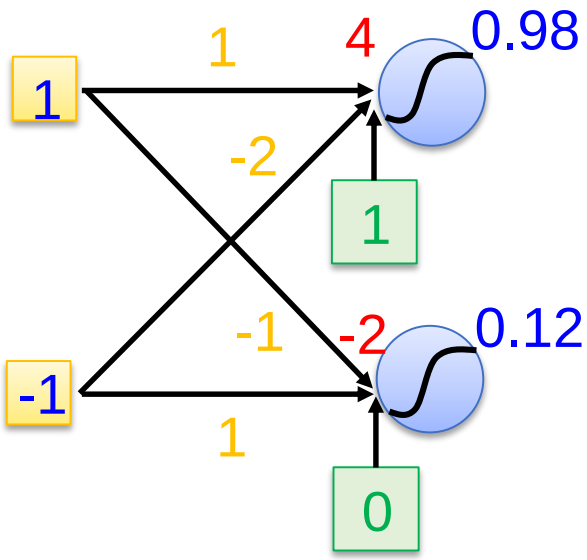
- A simple network, toy example (cont'd)
  - For an input vector $[1 \quad -1]^T$, the output is $[0.62 \quad 0.83]^T$



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

Matrix Operation

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

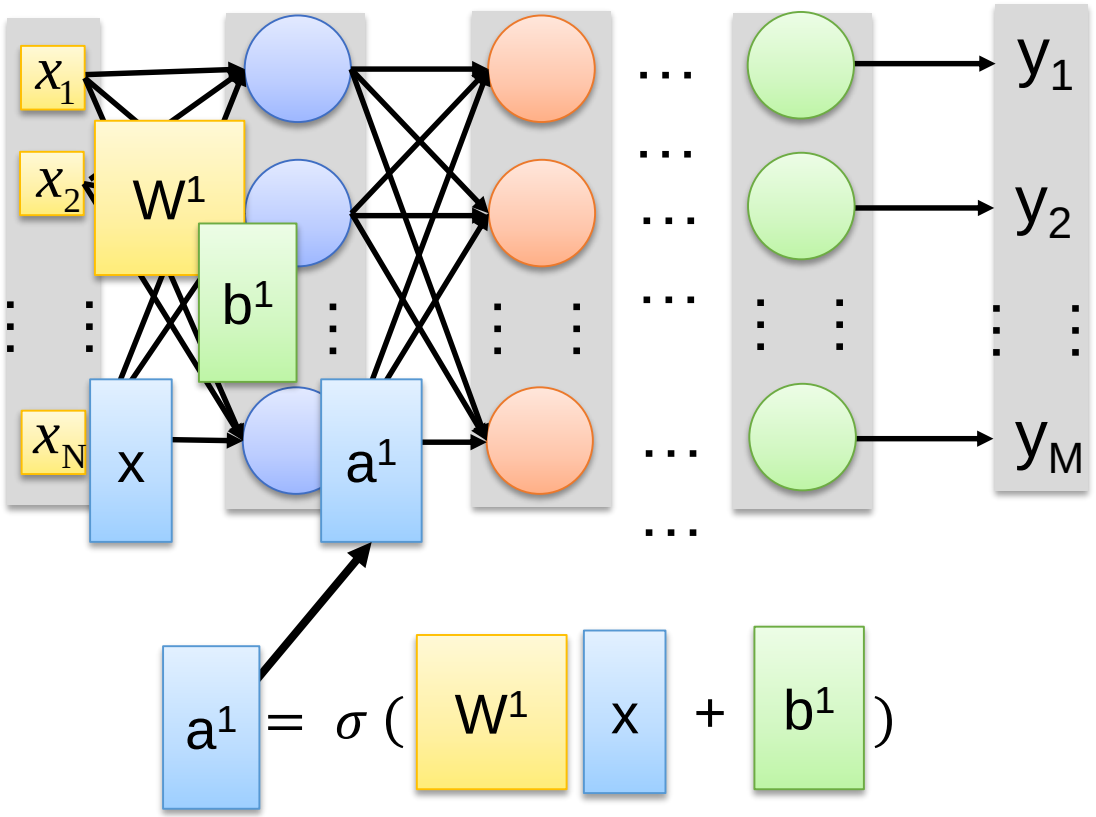


$$\sigma\left(\begin{bmatrix}1 & -2\\ -1 & 1\end{bmatrix}\begin{bmatrix}1\\ -1\end{bmatrix} + \begin{bmatrix}1\\ 0\end{bmatrix}\right) = \begin{bmatrix}0.98\\ 0.12\end{bmatrix}$$

$$\begin{bmatrix}4\\ -2\end{bmatrix}$$

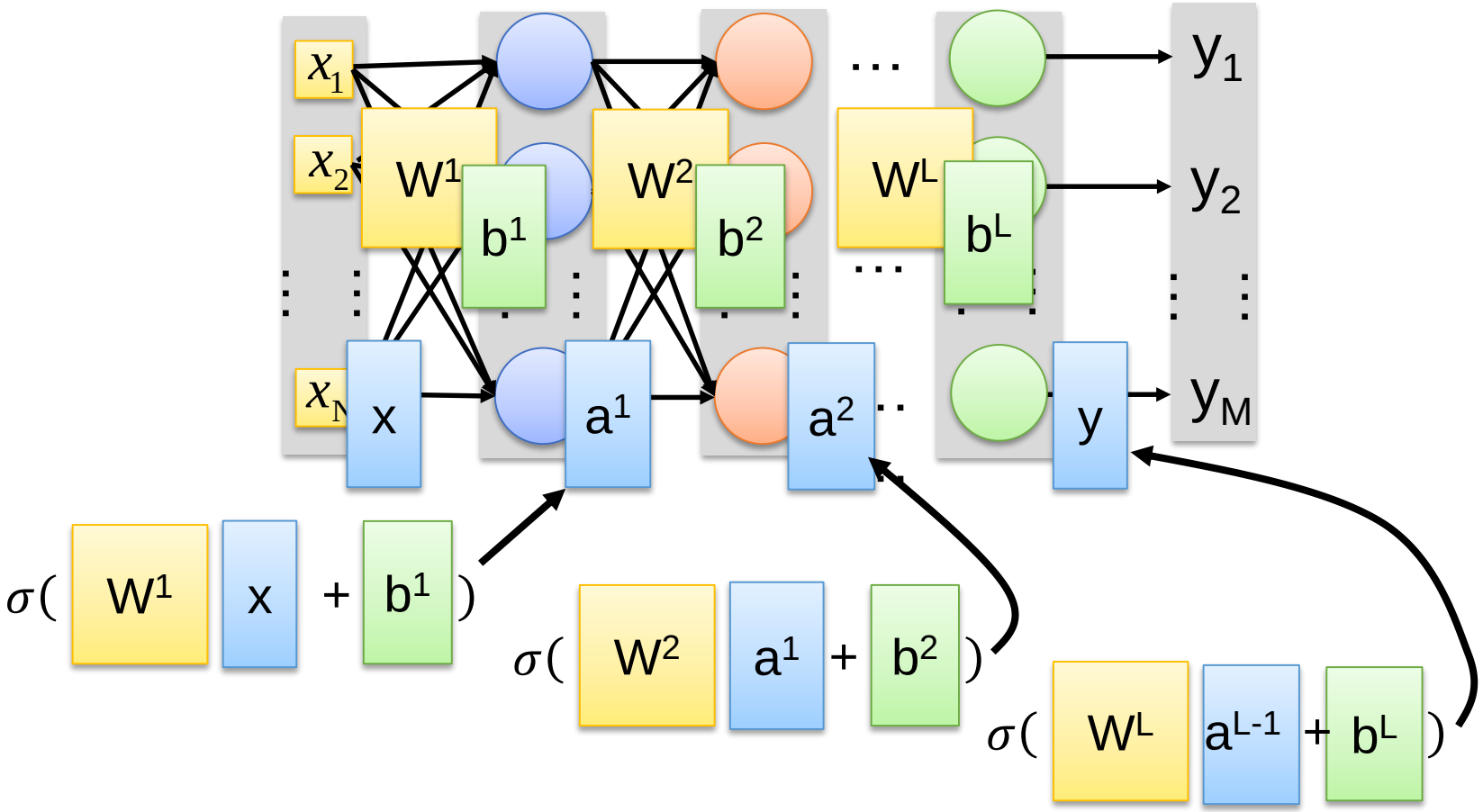
Matrix Operation

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



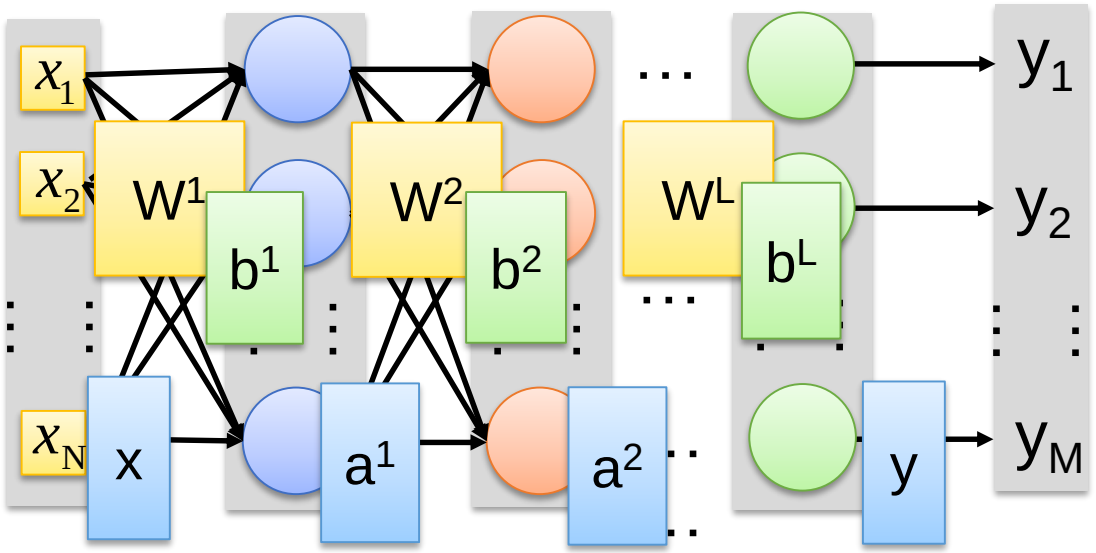
Matrix Operation

- Multilayer NN, matrix calculations for all layers



Matrix Operation

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

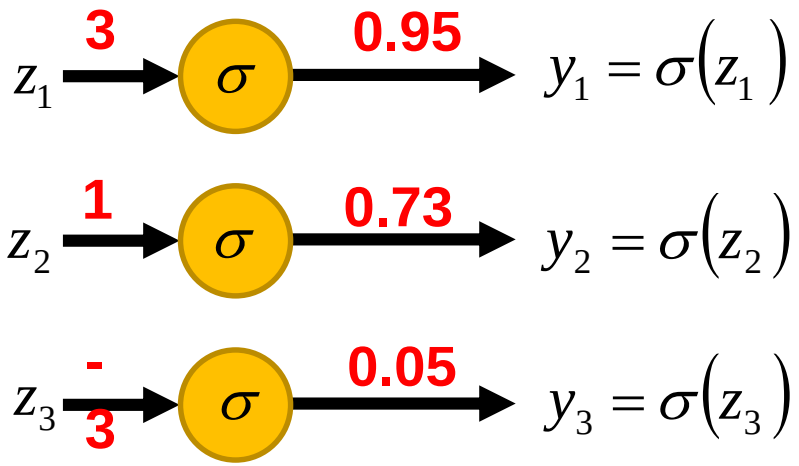


$$y = f(x) = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \cdots + b^L)$$

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

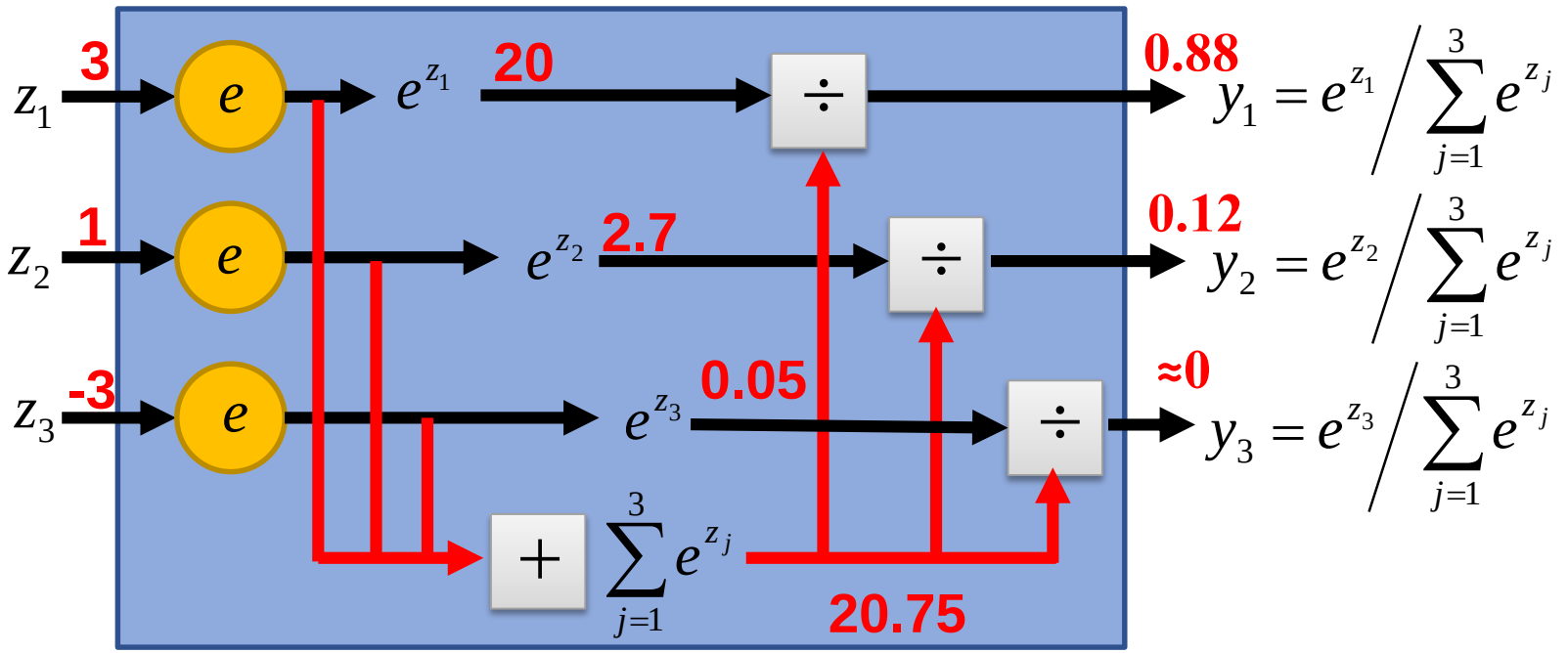


Softmax Layer

- 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***

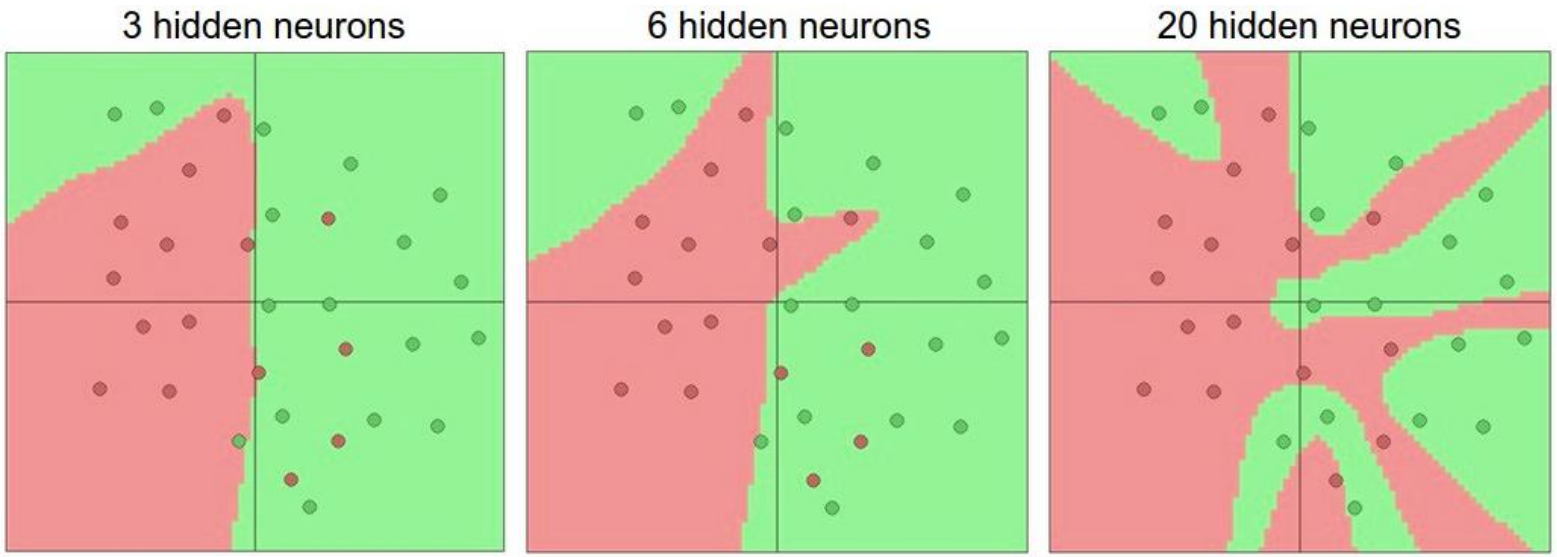
Probability:
- $0 < y_i < 1$
- $\sum_i y_i = 1$

A Softmax Layer



Activation Functions

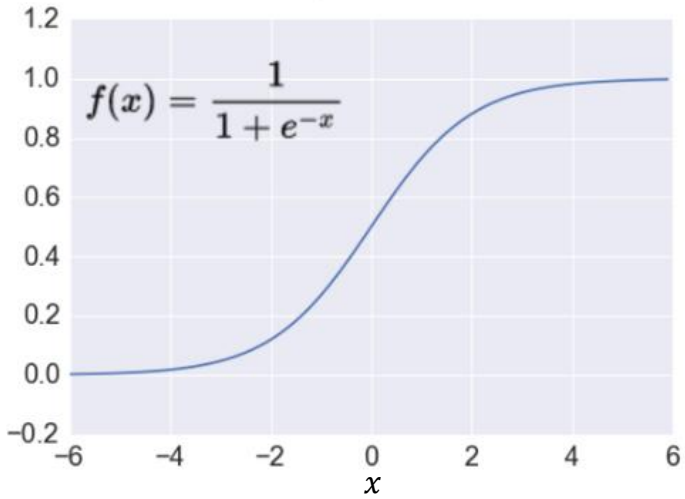
- Non-linear activations are needed to learn complex (non-linear) data representations
  - Otherwise, NNs would be just a linear function (such as $W_1 W_2 x = Wx$)
  - NNs with large number of layers (and neurons) can approximate more complex functions
    - Figure: more neurons improve representation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



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

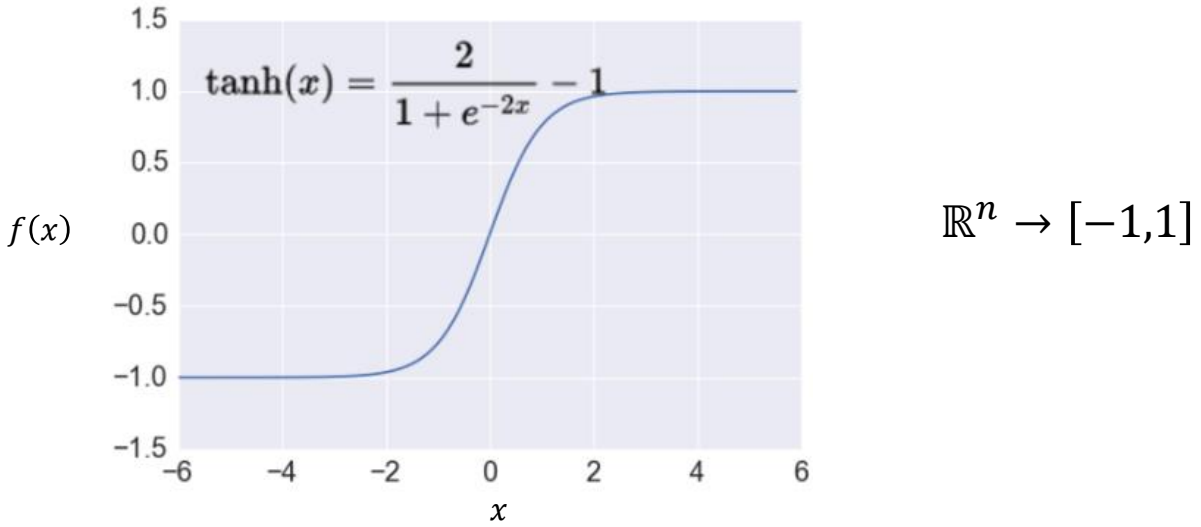
$f(x)$



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

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

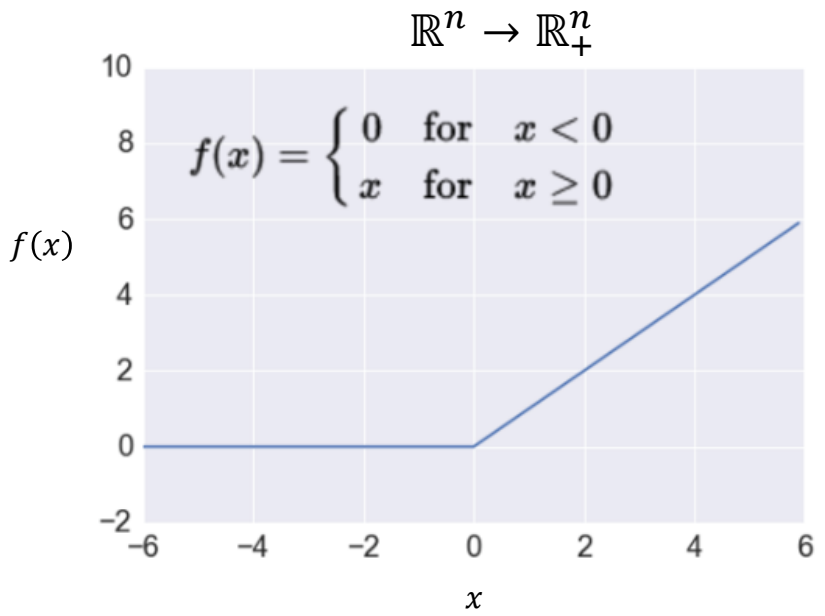


$\mathbb{R}^n \rightarrow [-1,1]$

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

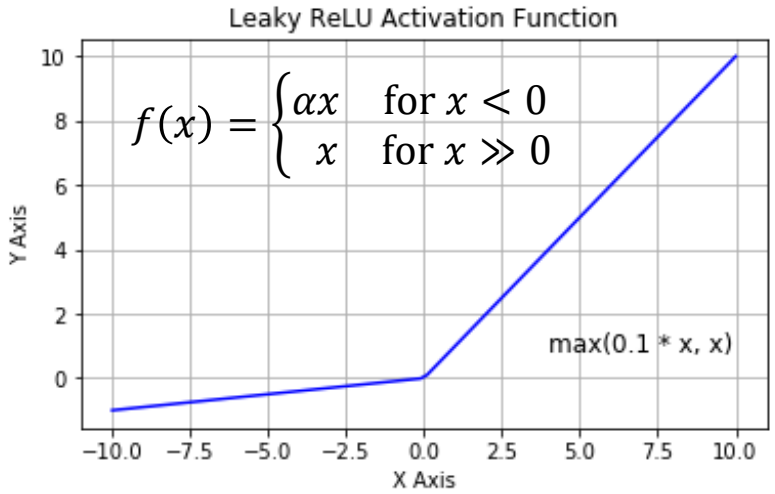


Activation: Leaky ReLU

- 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

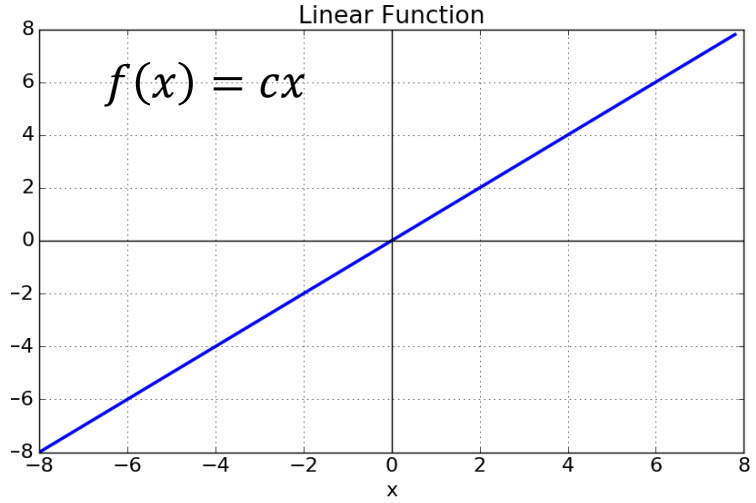


Activation: Linear Function

- ***Linear function*** means that the output signal is proportional to the input signal to the neuron

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

  - 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)

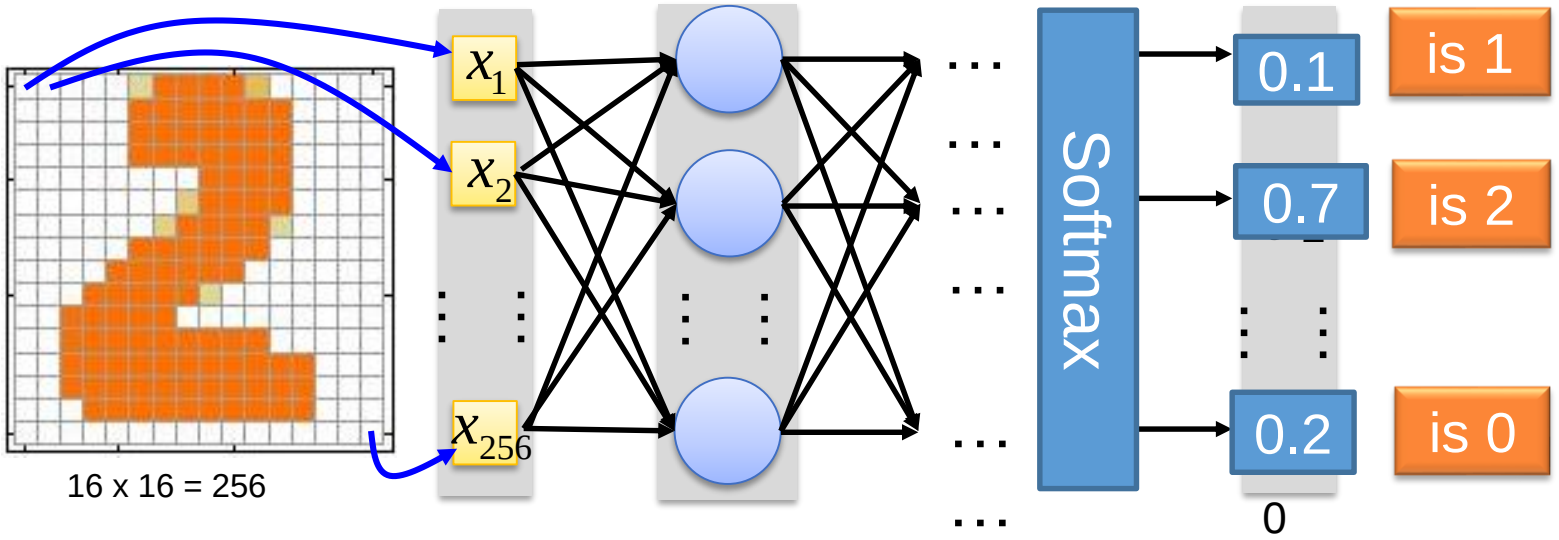


Training NNs

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

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

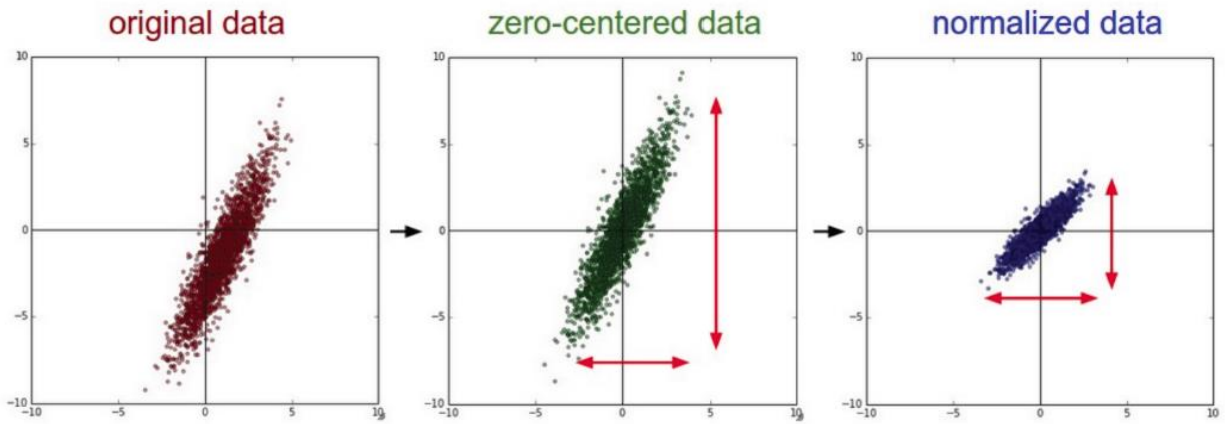
  - Often, the model parameters θ are referred to as weights
- Training a model to learn a set of parameters θ that are optimal (according to a criterion) is one of the greatest challenges in ML



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Training NNs

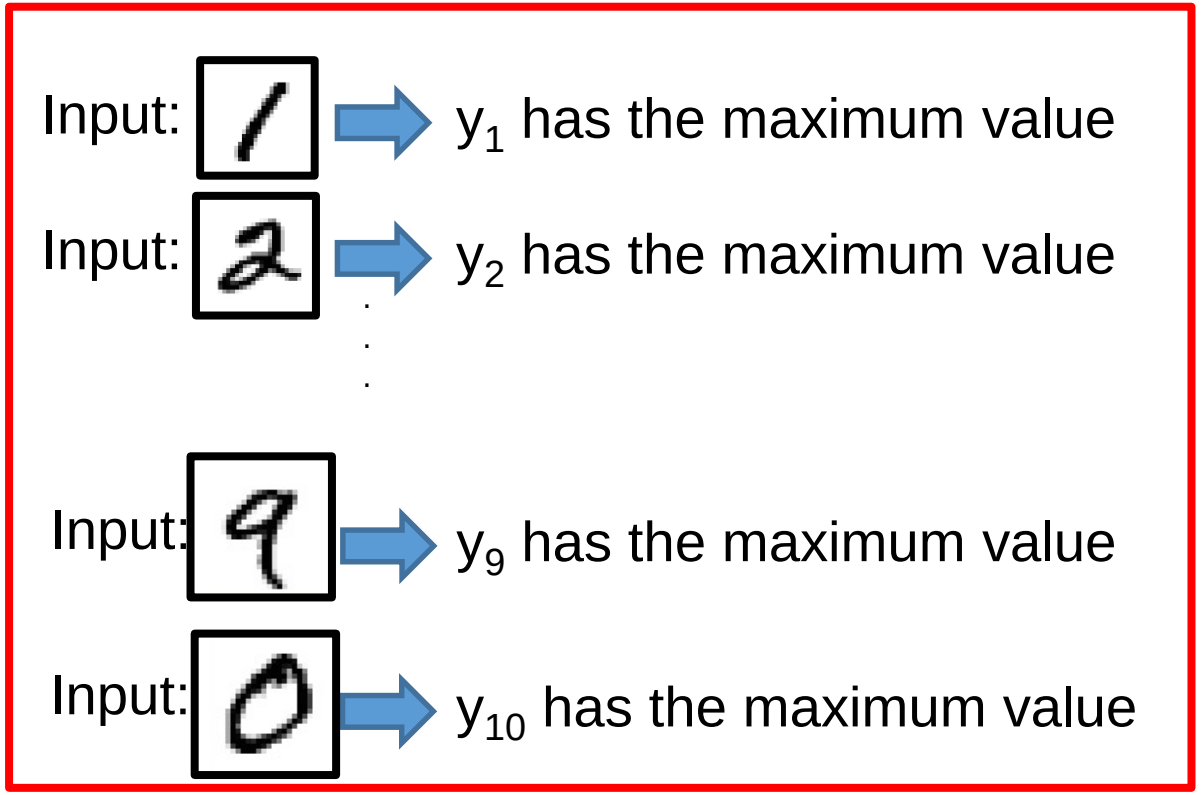
- *Data preprocessing* – helps convergence during training
  - Mean subtraction, to obtain zero-centered data
    - Subtract the mean for each individual data dimension (feature)
  - Standardization
    - In zero-centered data, divide each feature by its standard deviation
      - To obtain standard 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/

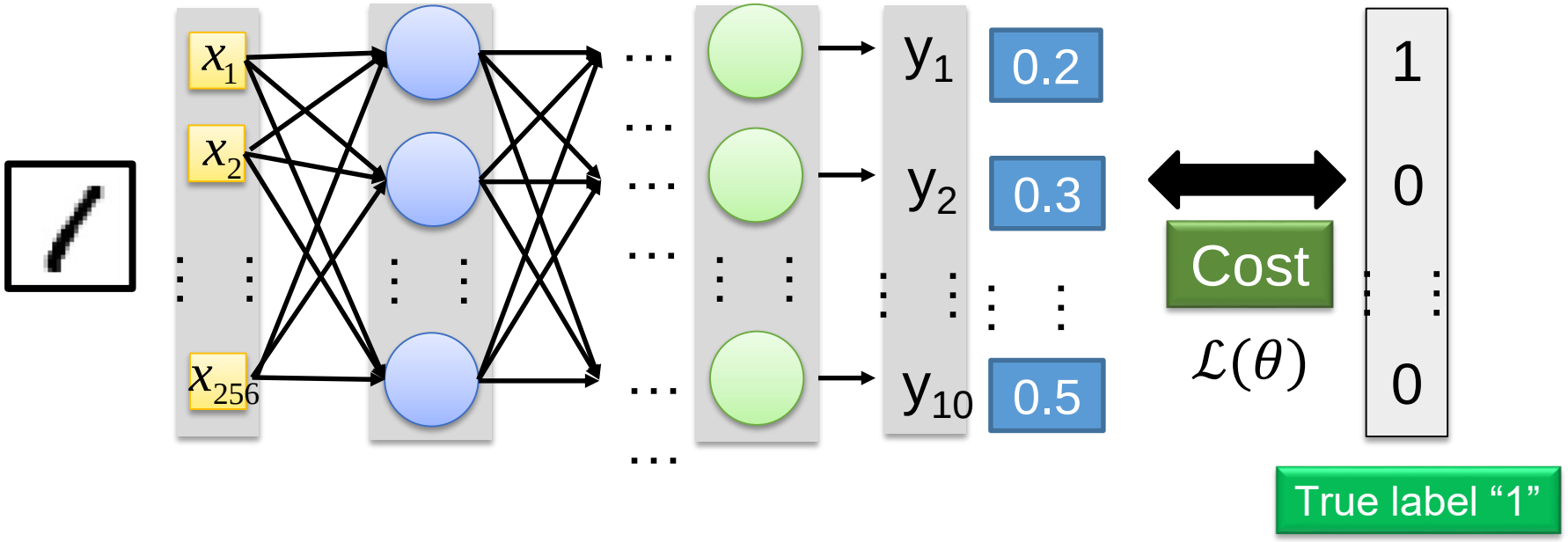
Training NNs

- 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



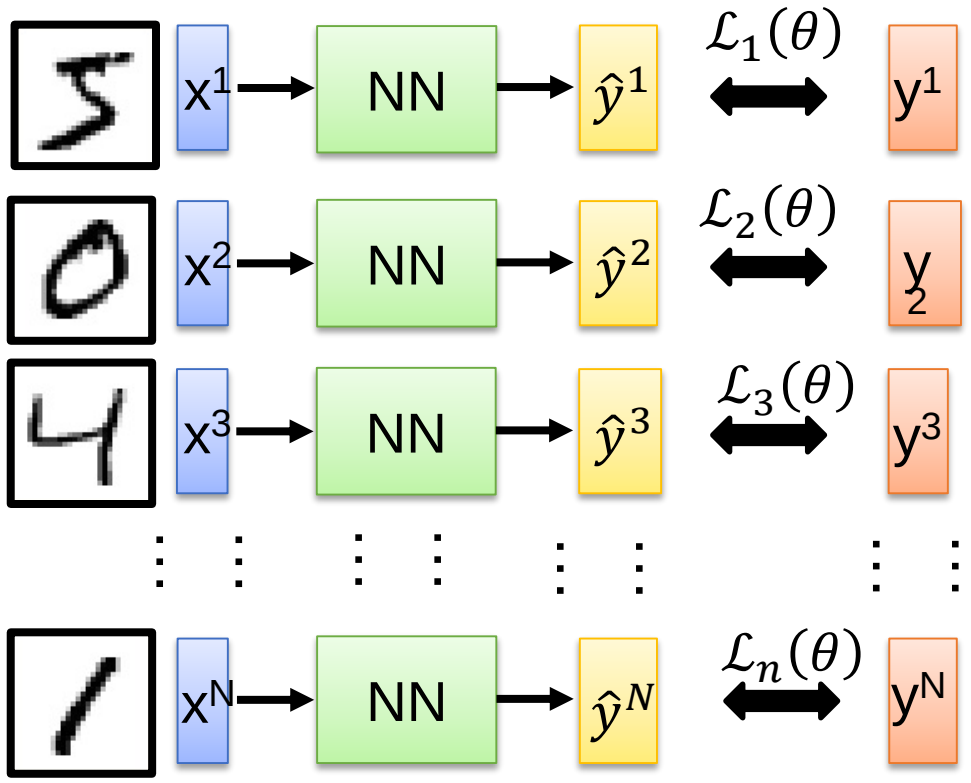
Training NNs

- 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

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



Loss Functions

- ***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

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[y_k^{(i)} \log \hat{y}_k^{(i)} + \left(1 - y_k^{(i)}\right) \log\left(1 - \hat{y}_k^{(i)}\right)\right]$$

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

Loss Functions

- *Regression tasks*

Training examples

Pairs of N inputs x_i and ground-truth output values y_i

Output Layer

Linear (Identity) or Sigmoid Activation

Loss function

Mean Squared Error

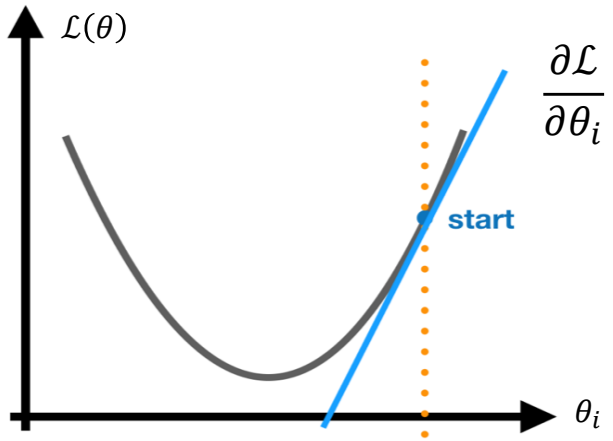
$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

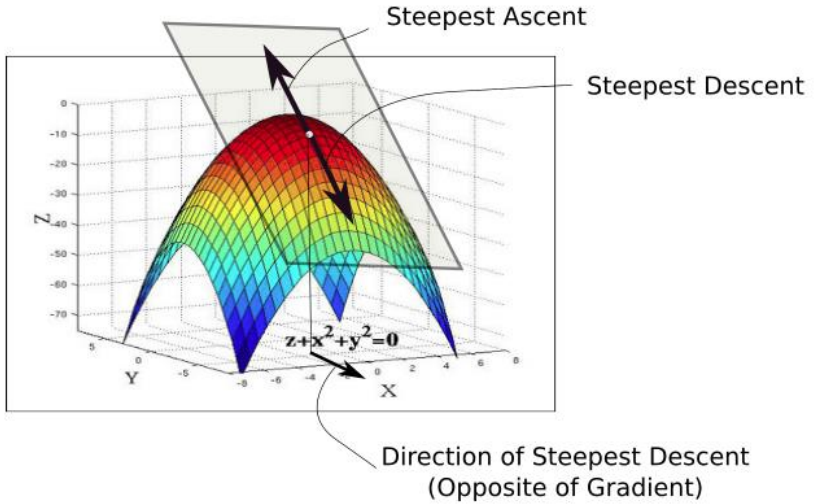
Mean Absolute Error

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

Training NNs

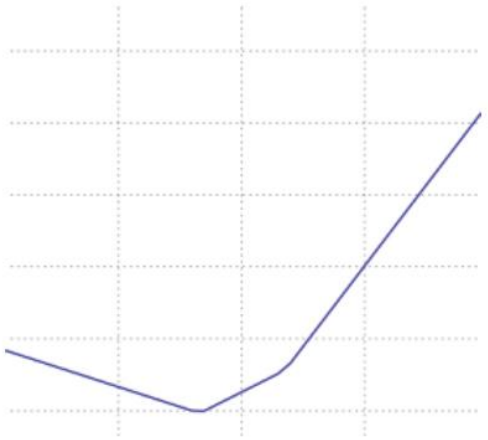
- Optimizing the loss function $\mathcal{L}(\theta)$
  - Almost all DL models these days are trained with a variant of the ***gradient descent*** (GD) algorithm
  - GD applies iterative refinement of the network parameters θ
  - GD uses the opposite direction of the gradient of the loss with respect 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



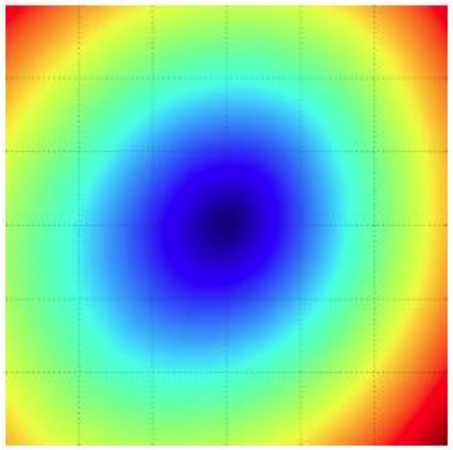


Training NNs

- 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)

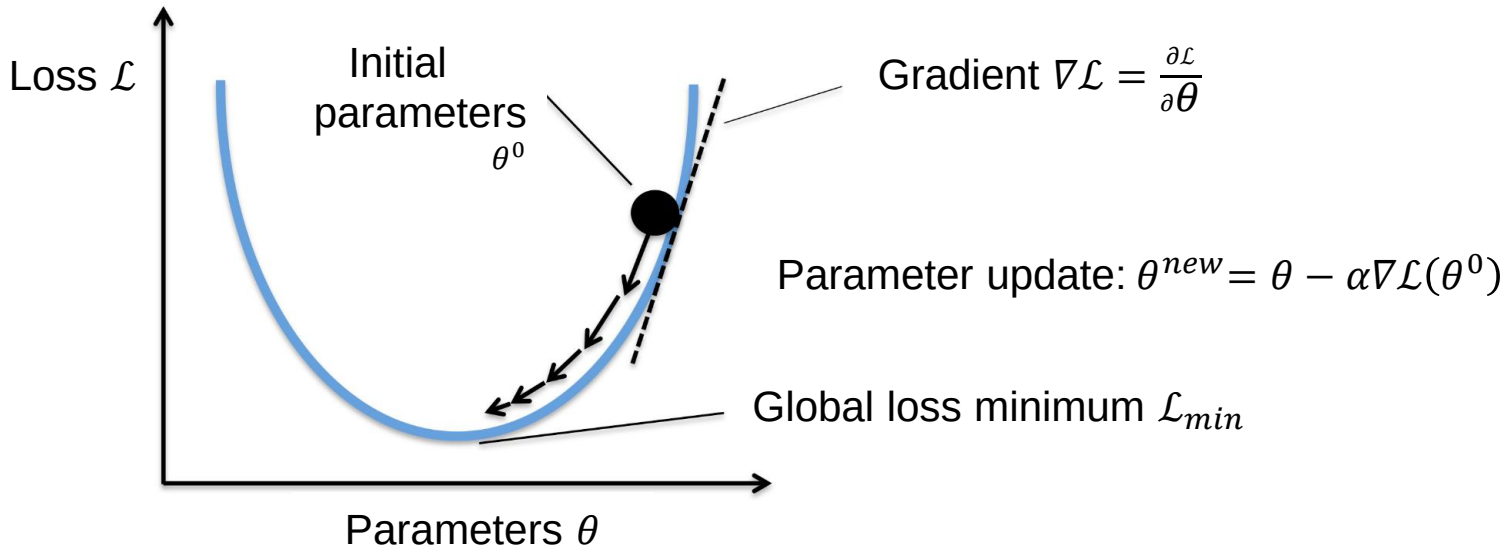


2D loss (blue = low loss, red = high loss)

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

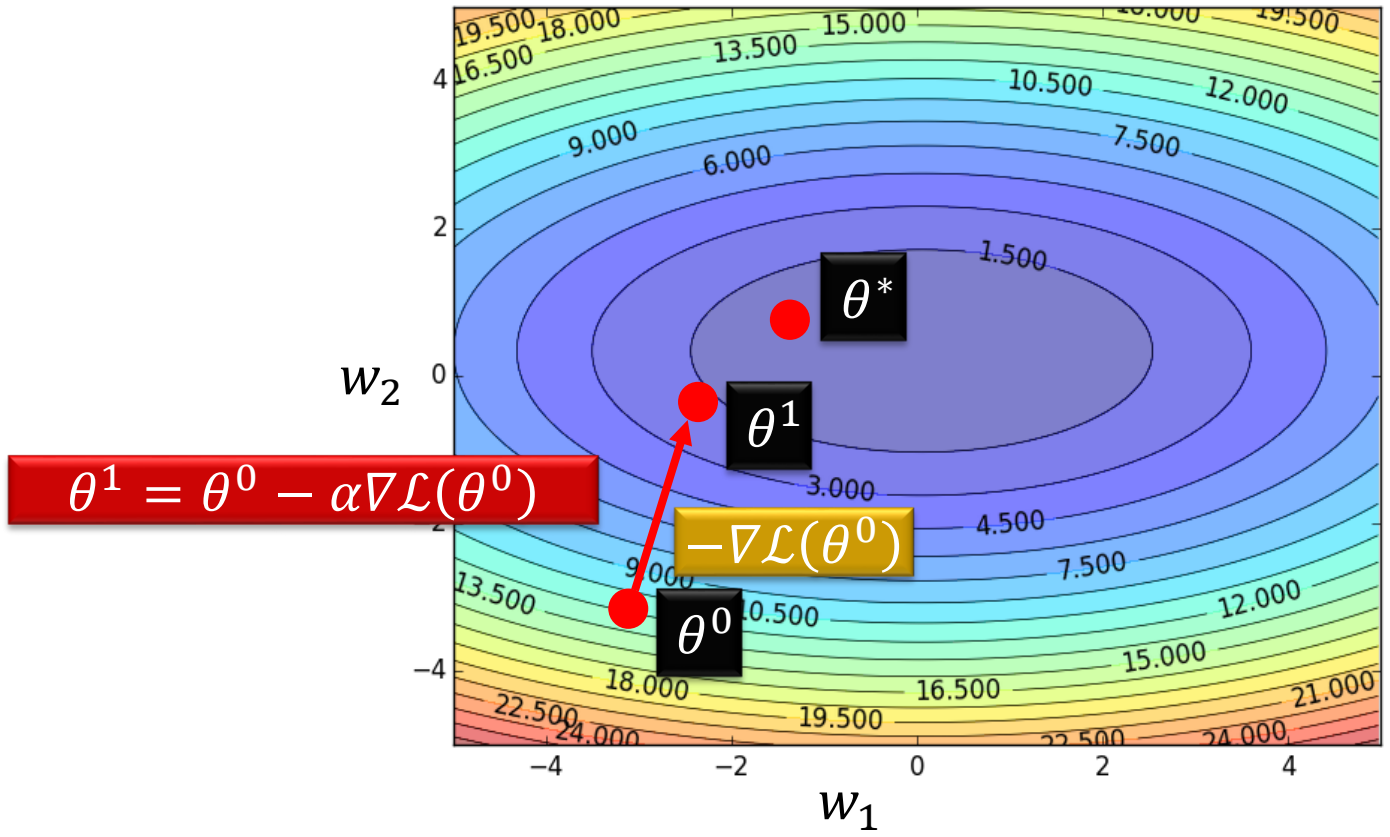
Gradient Descent Algorithm

- Steps in the ***gradient descent algorithm***:
  1. Randomly initialize the model parameters, θ^0
  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)



Gradient Descent Algorithm

- 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)

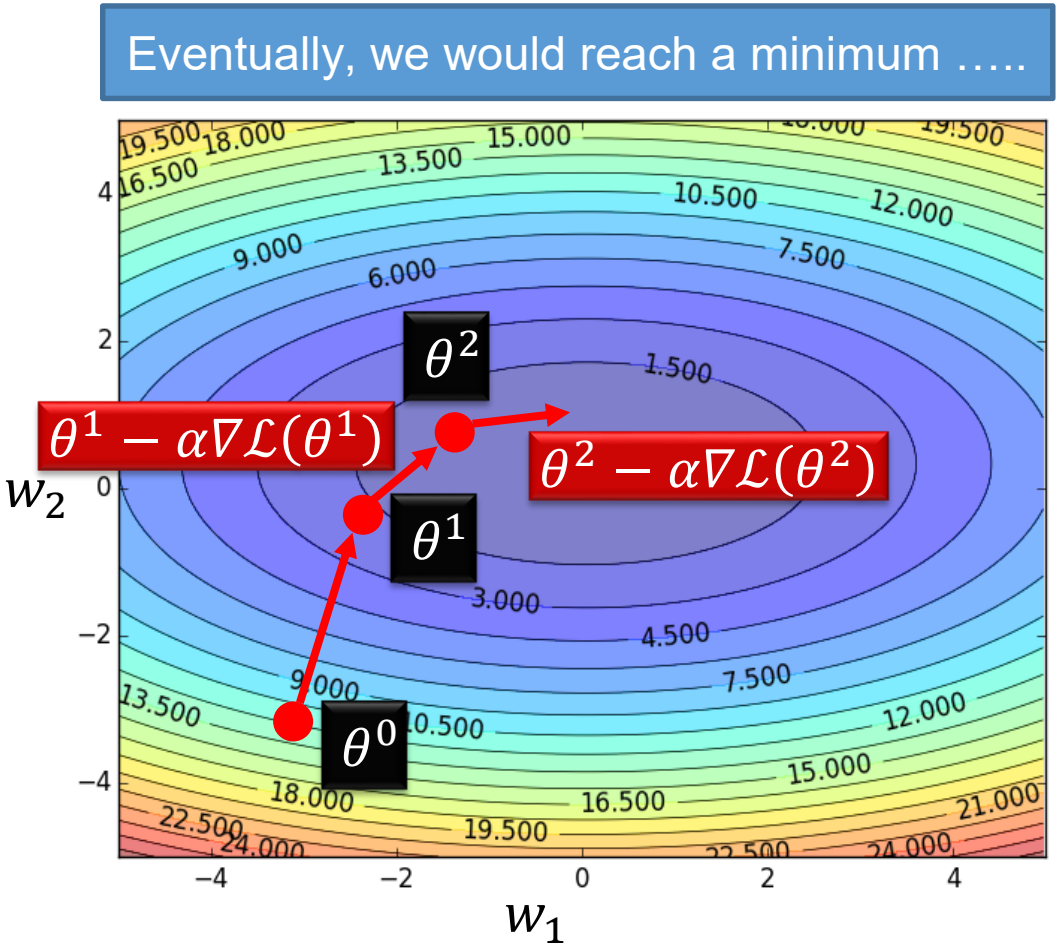


1. Randomly pick a starting point θ^0
2. Compute the gradient at θ^0, $\nabla\mathcal{L}(\theta^0)$
3. Times the learning rate η, and update θ, $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$
4. Go to step 2, repeat

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

Gradient Descent Algorithm

- Example (contd.)



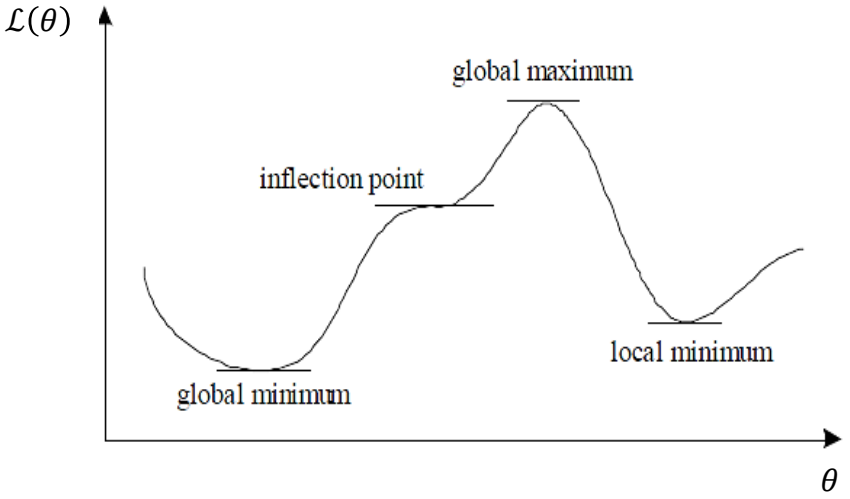
2. Compute the gradient at θ^{old}, $\nabla\mathcal{L}(\theta^{old})$

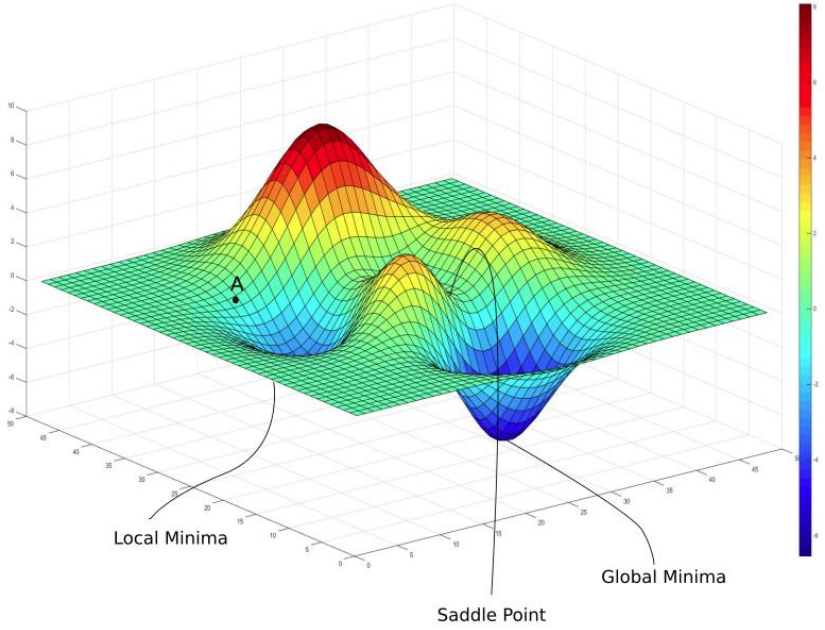
3. Times the learning rate η, and update θ,
$\theta^{new} = \theta^{old} - \alpha\nabla\mathcal{L}(\theta^{old})$

4. Go to step 2, repeat

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

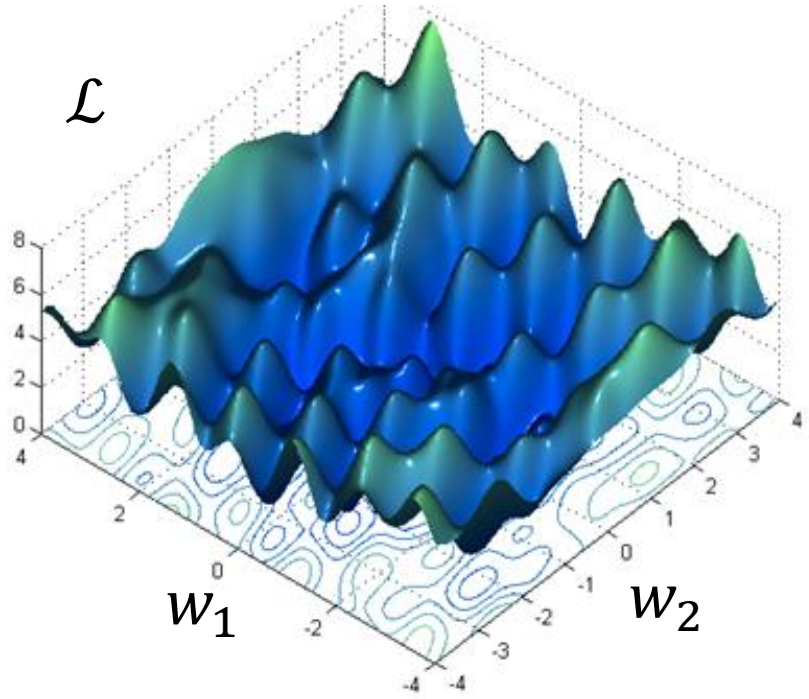




Picture from: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

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 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

- 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

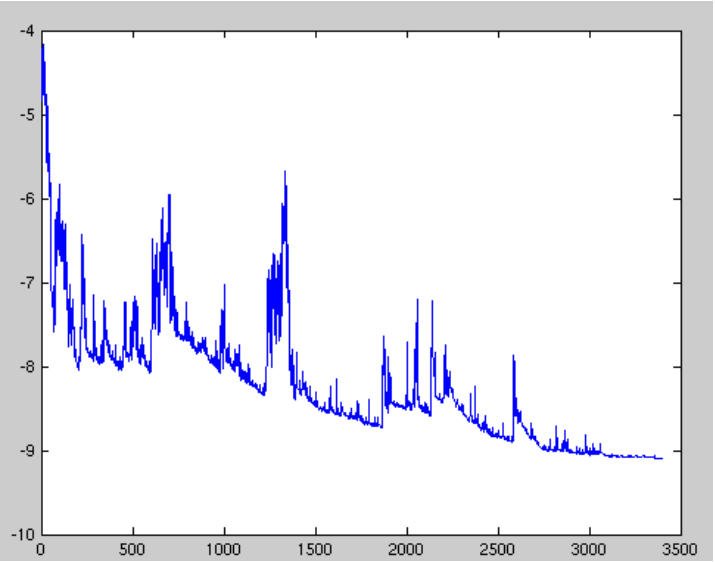
Mini-batch Gradient Descent

- 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

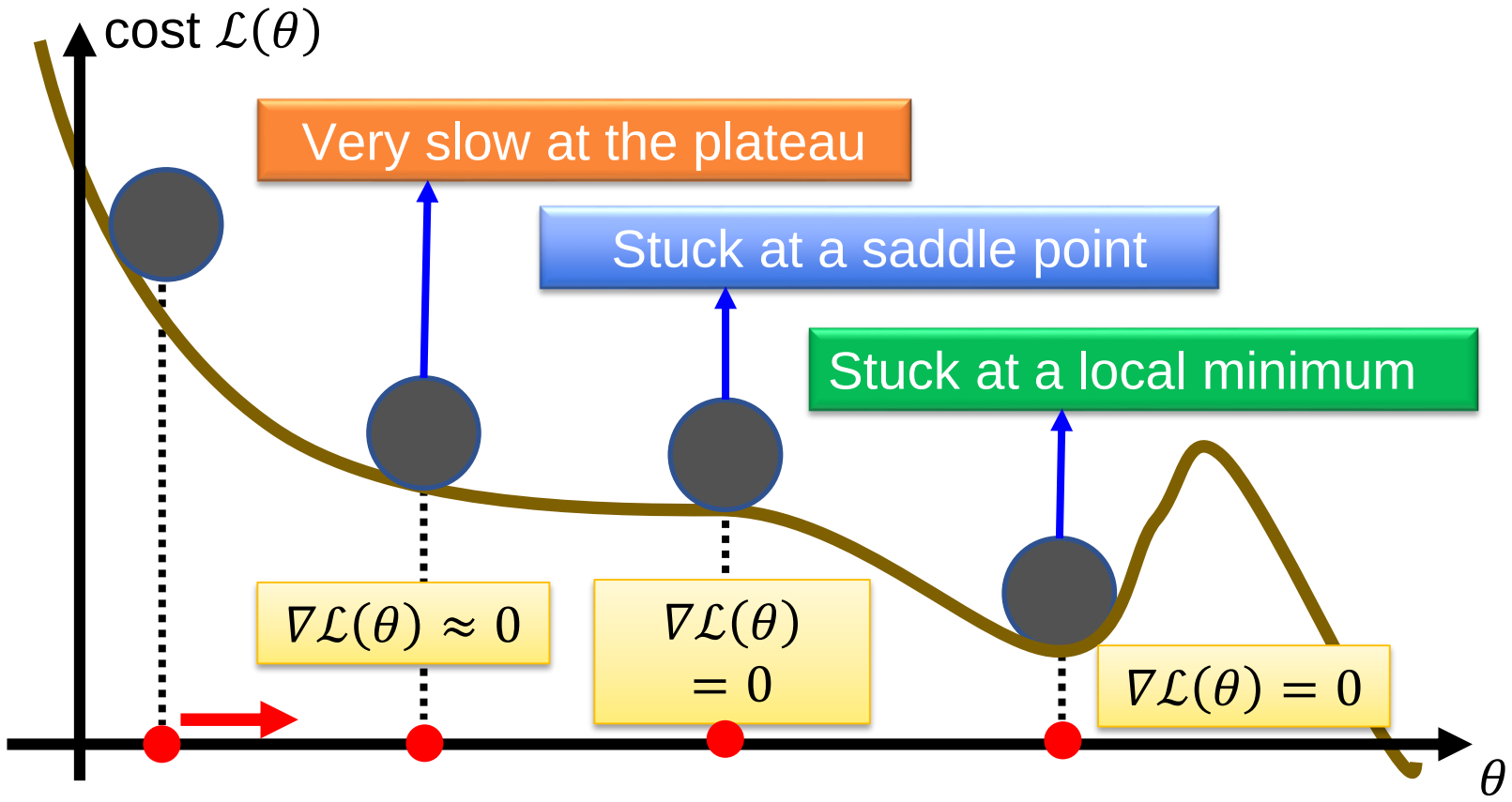
Stochastic Gradient Descent

- ***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)



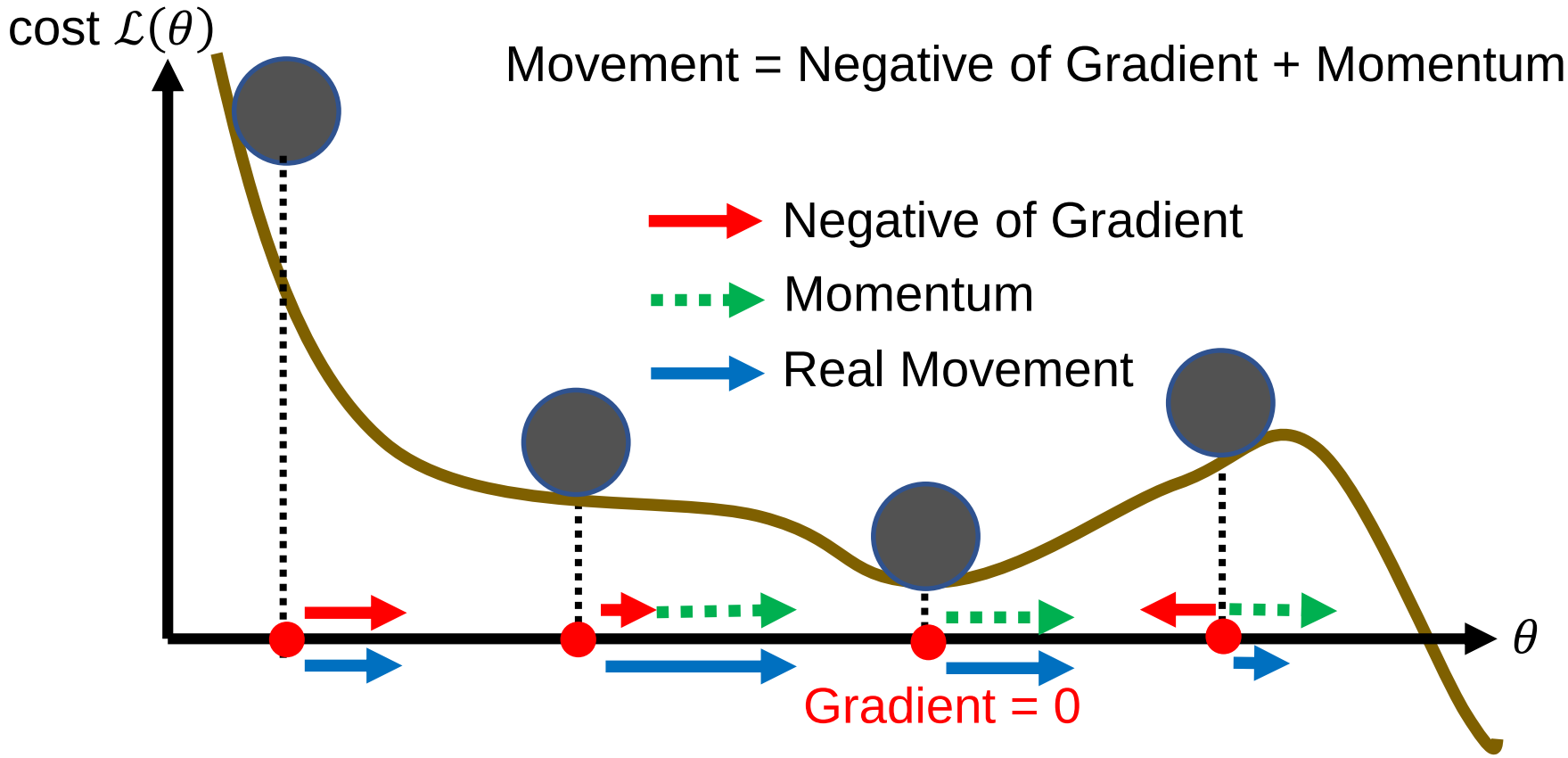
Problems with Gradient Descent

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



Gradient Descent with Momentum

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



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.,

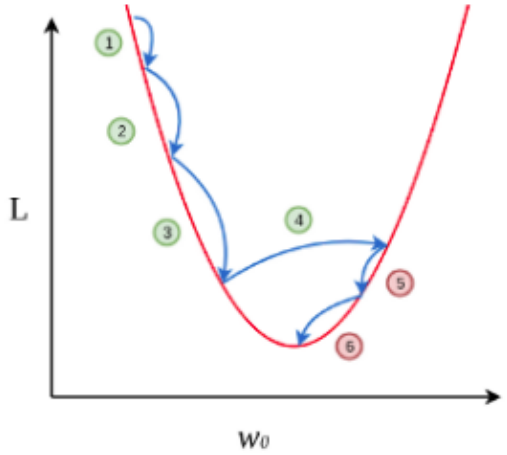
$$\begin{aligned} V^t &= \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta\left(\beta V^{t-2} + \alpha \nabla \mathcal{L}(\theta^{t-2})\right) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta^2 V^{t-2} + \beta\alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1})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

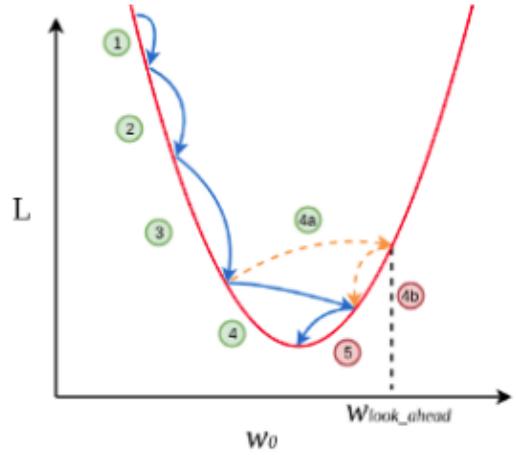
  - This term is analogous to a momentum of a heavy ball rolling down the hill
- The parameter β is referred to as a coefficient of momentum
  - A typical value of the parameter β is 0.9
- This method updates the parameters θ in the direction of the weighted average of the past gradients

Nesterov Accelerated Momentum

- ***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

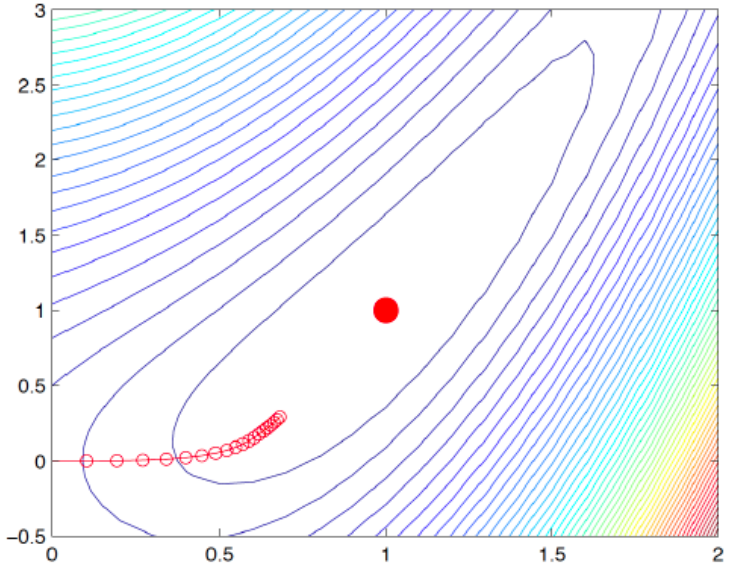
Adam

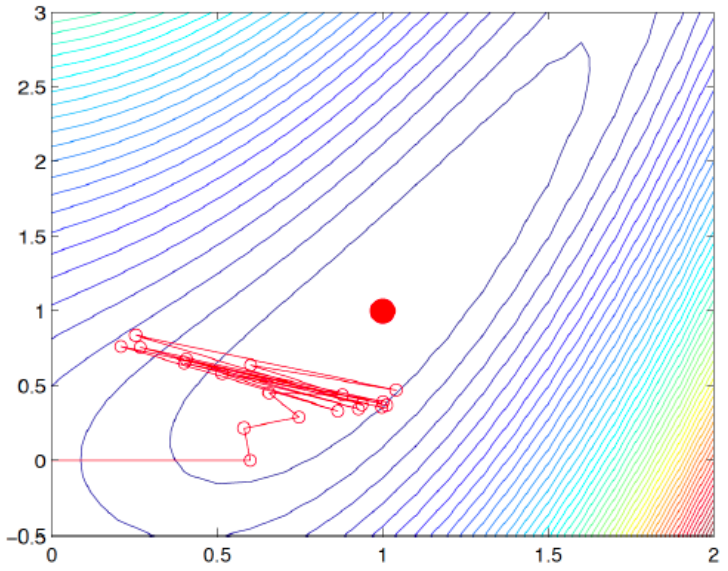
- ***Adaptive Moment Estimation (Adam)***
  - Adam combines insights from the momentum optimizers that accumulate the values of past gradients, and it also introduces new terms based on the second moment of the gradient
    - Similar to GD with momentum, Adam computes a **weighted average of past gradients** (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: $\theta^t = \theta^{t-1} - \alpha \frac{\hat{V}^t}{\sqrt{\hat{U}^t}+\epsilon}$
    - Where: $\hat{V}^t = \frac{V^t}{1-\beta_1}$ and $\hat{U}^t = \frac{U^t}{1-\beta_2}$
    - 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

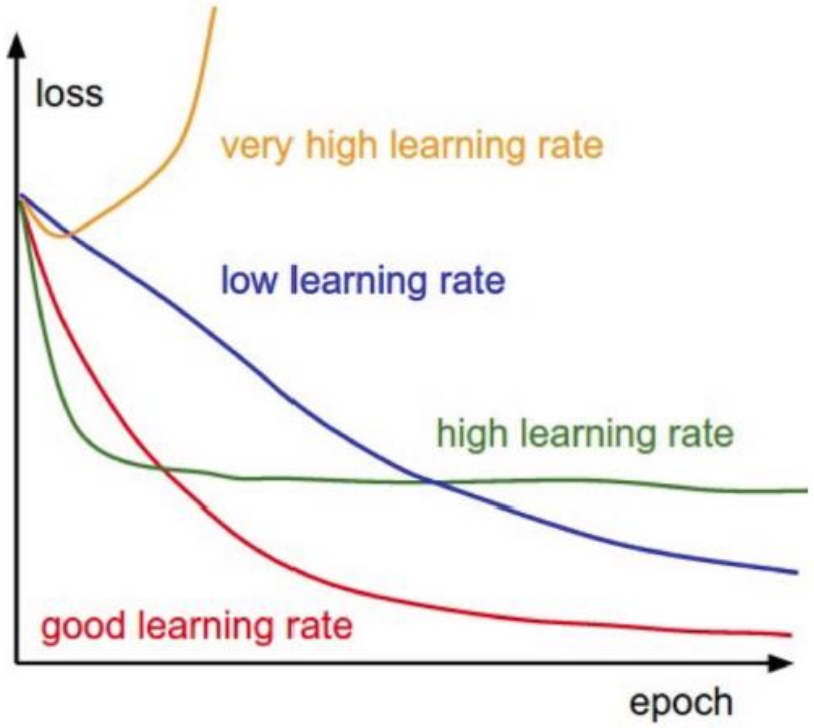




LR too large

Learning Rate

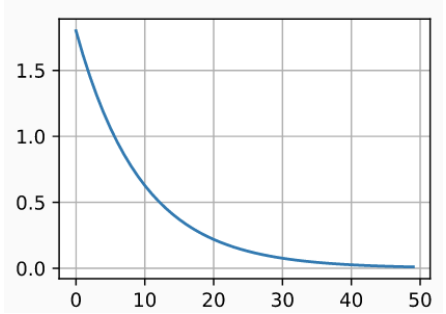
- Training loss for different learning rates
  - High learning rate: the loss increases or plateaus too quickly
  - Low learning rate: the loss decreases too slowly (takes many epochs to reach a solution)



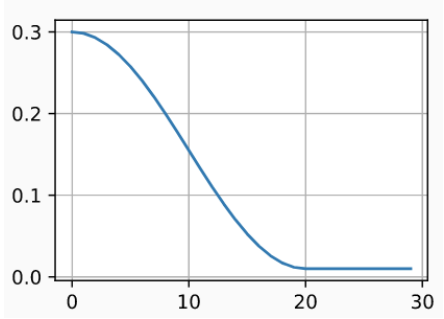
Learning Rate Scheduling

- ***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

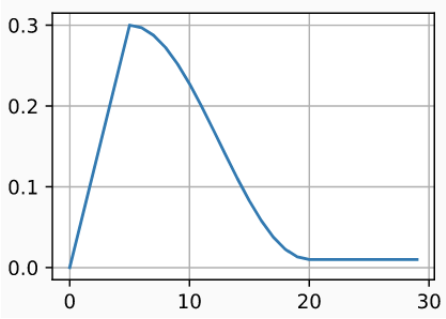
Exponential decay



Cosine decay

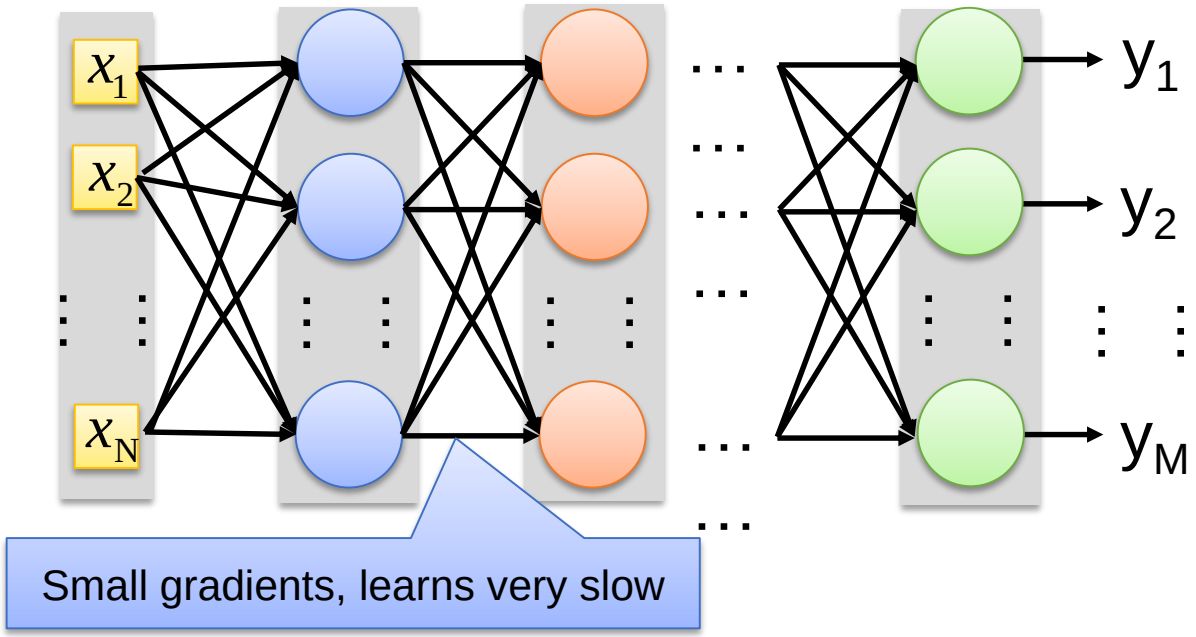


Warmup



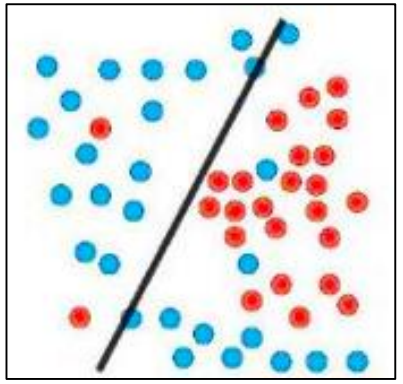
Vanishing Gradient Problem

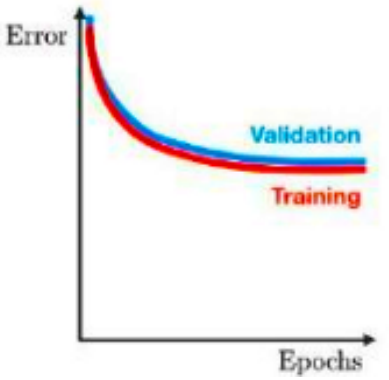
- In some cases, during training, the gradients can become either very small (vanishing gradients) of very large (exploding gradients)
  - They result in very small or very large update of the parameters
  - Solutions: change learning rate, ReLU activations, regularization, LSTM units in RNNs



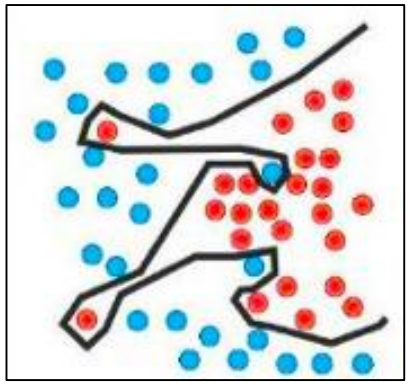
Generalization

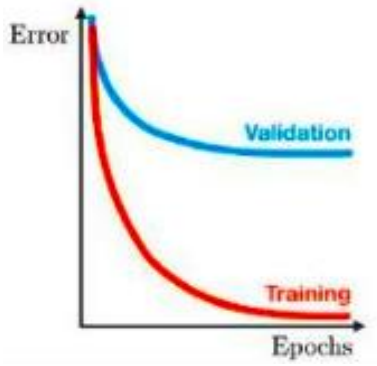
- *Underfitting*
  - The model is too "simple" to represent all the relevant class characteristics
  - E.g., model with too few parameters
  - Produces high error on the training set and high error on the validation set





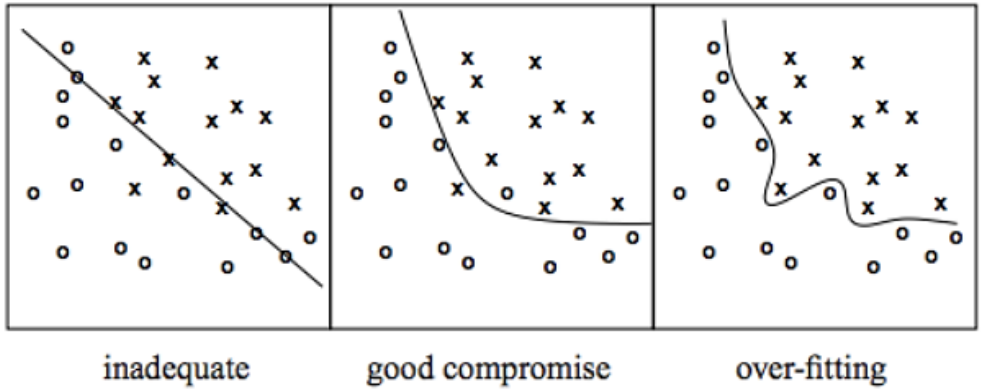
- *Overfitting*
  - The model is too "complex" and fits irrelevant characteristics (noise) in the data
  - E.g., model with too many parameters
  - Produces low error on the training error and high error on the validation set



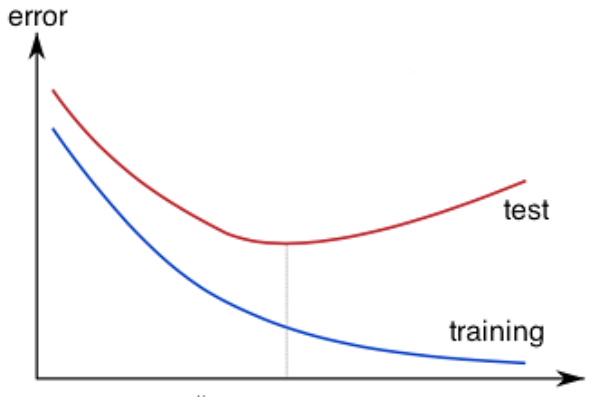


Overfitting

- 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)



Regularization: Weight Decay

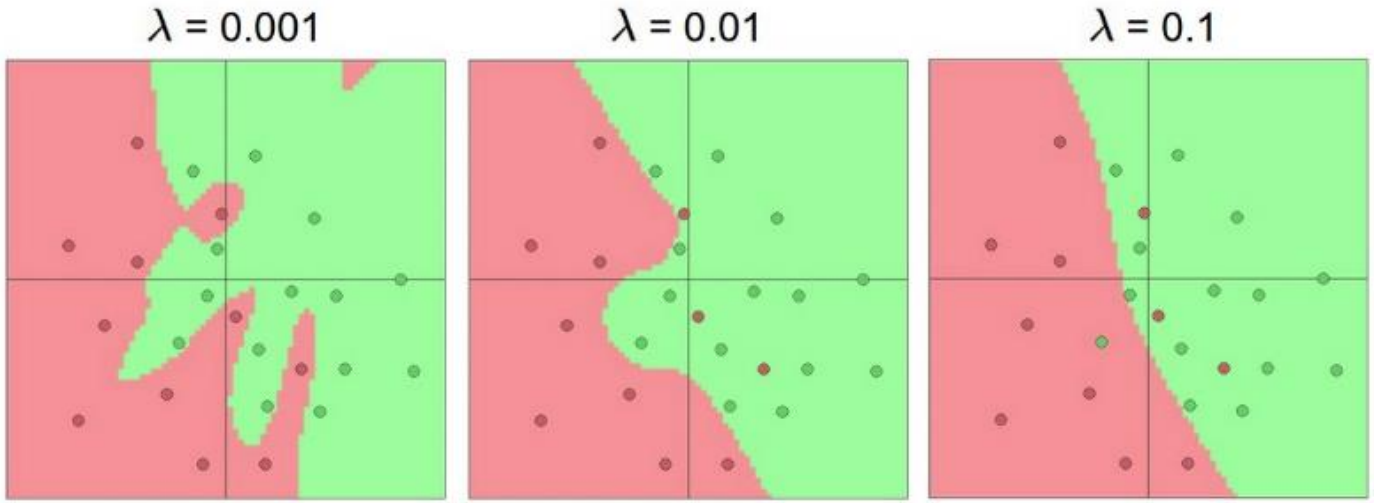
- *ℓ_2 weight decay*
  - A regularization term that penalizes large weights is added to the loss function

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

  - 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

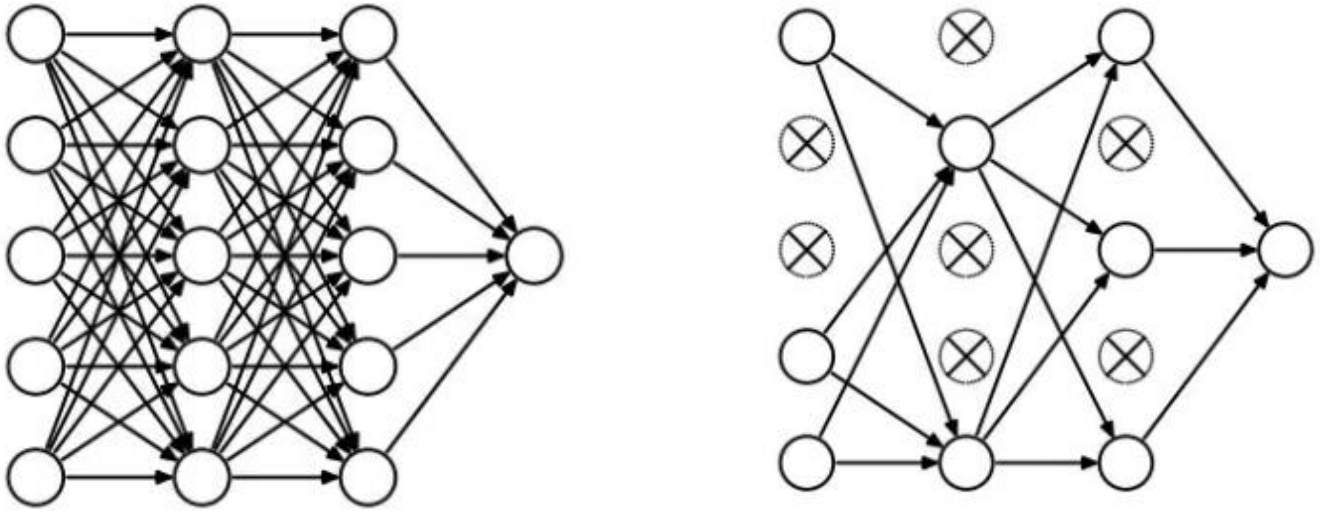
$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k |\theta_k|$$

  - 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

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

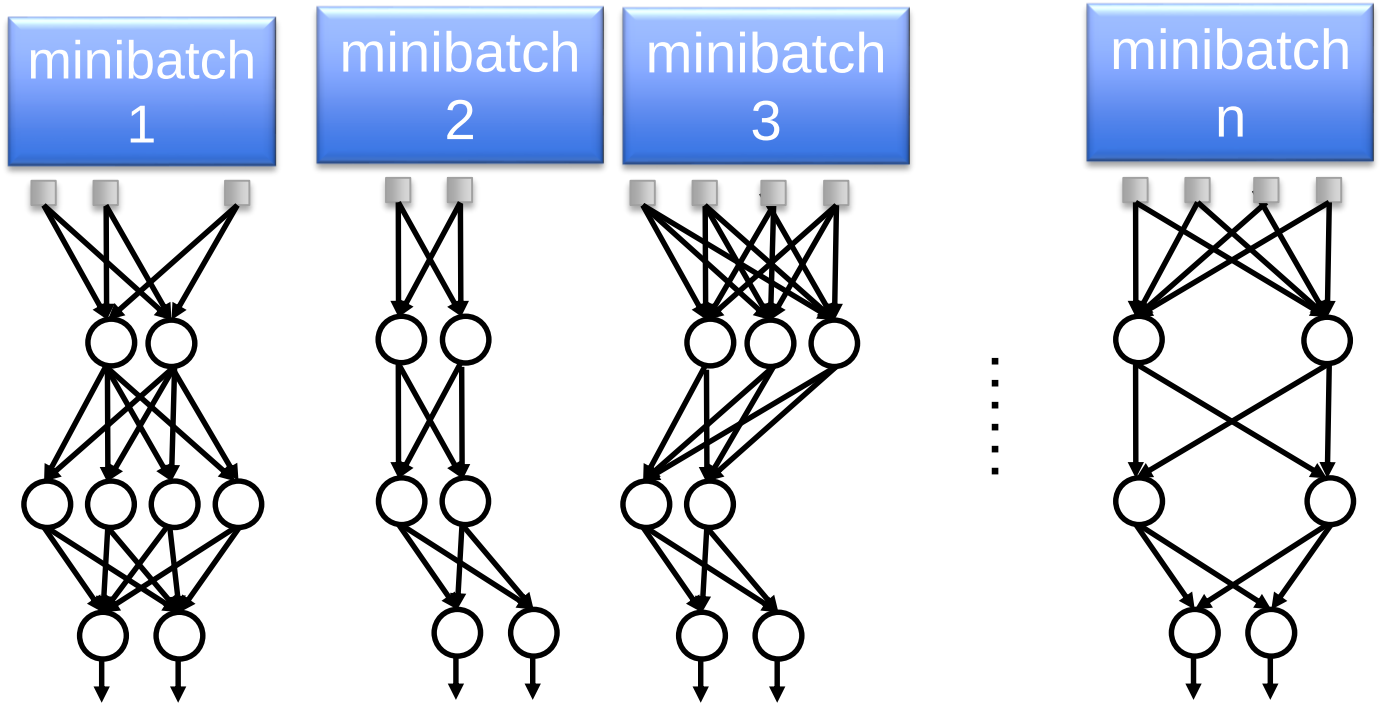
Regularization: Dropout

- ***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



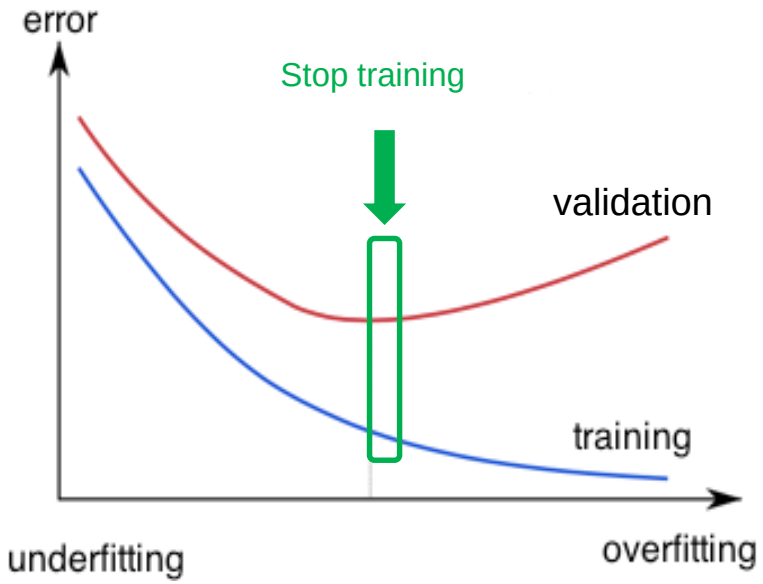
Regularization: Dropout

- 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 data
    - E.g., validation/train ratio of about 25% to 75% (often)
  - Stop when the validation accuracy (or loss) has not improved after n subsequent epochs
    - The parameter n is called patience



Batch Normalization

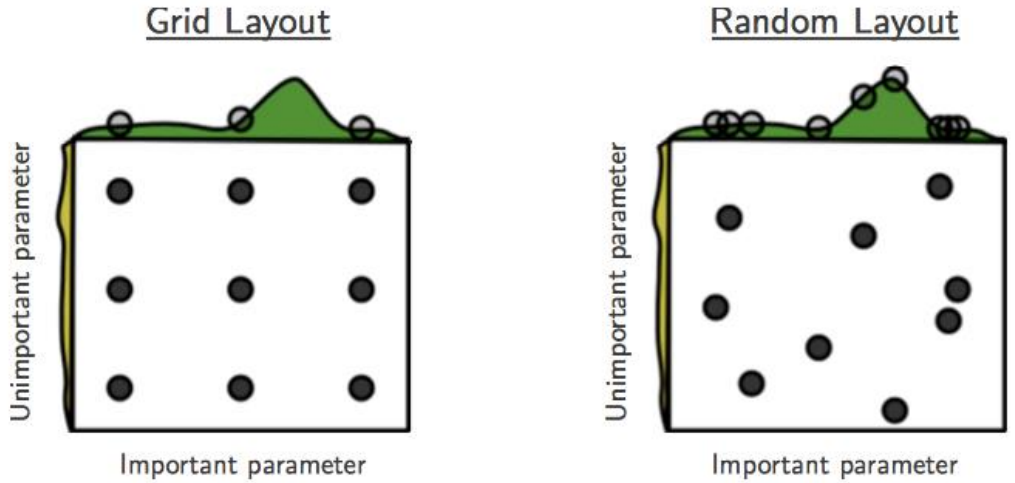
- ***Batch normalization layers*** act similar to the data preprocessing steps mentioned earlier
  - They calculate the mean μ and variance σ of a batch of input data, and normalize the data x to a zero mean and unit variance
  - i.e., $\hat{x} = \frac{x-\mu}{\sigma}$
- BatchNorm layers alleviate the problems of proper initialization of the parameters and hyper-parameters
  - Result in faster convergence training, allow larger learning rates
  - Reduce the internal covariate shift
- 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

- 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



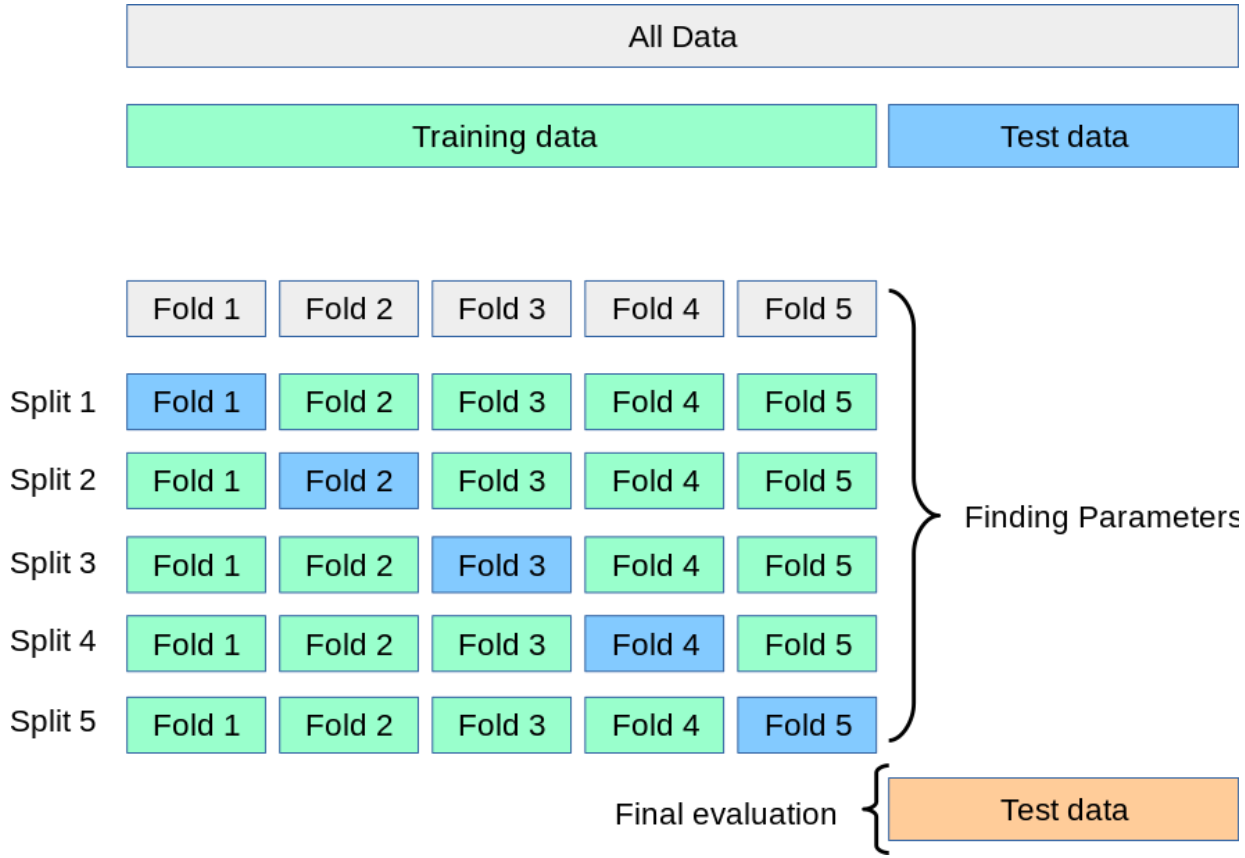
k-Fold Cross-Validation

- 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

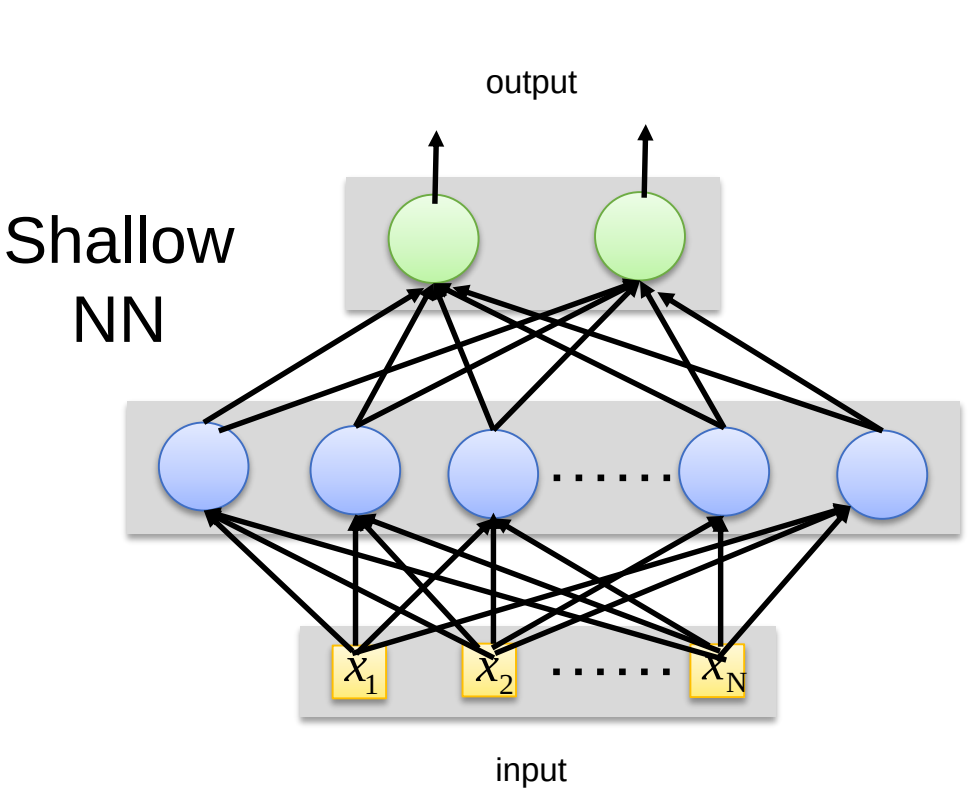


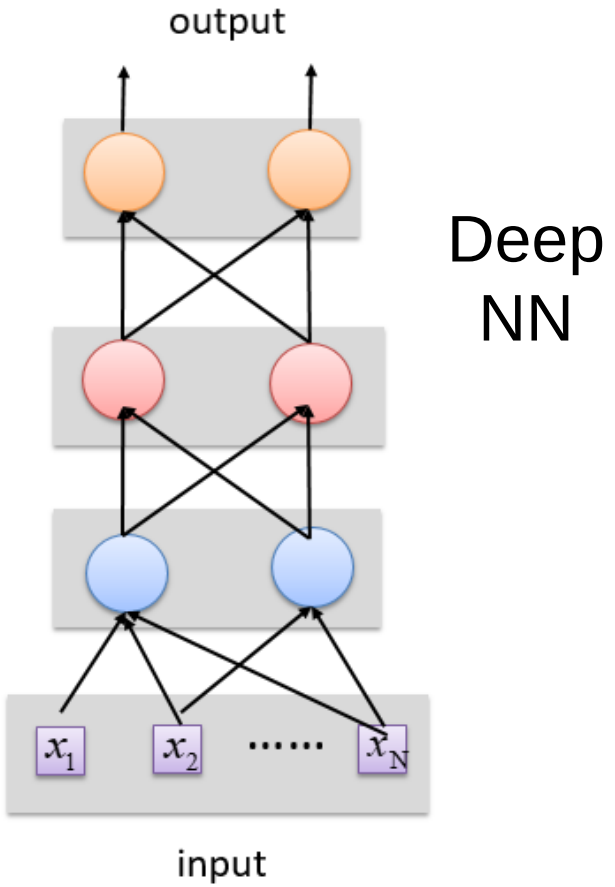
Ensemble Learning

- ***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

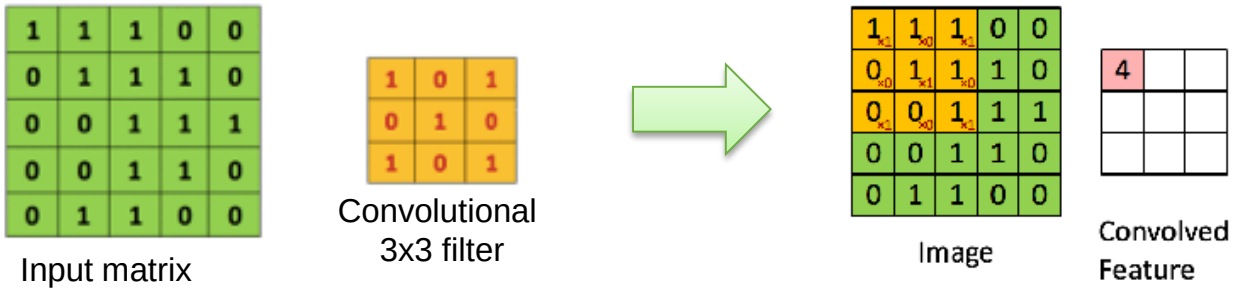
- Deeper networks perform better than shallow networks
  - But only up to some limit: after a certain number of layers, the performance of deeper networks plateaus





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



Convolutional Neural Networks (CNNs)

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

Filter 

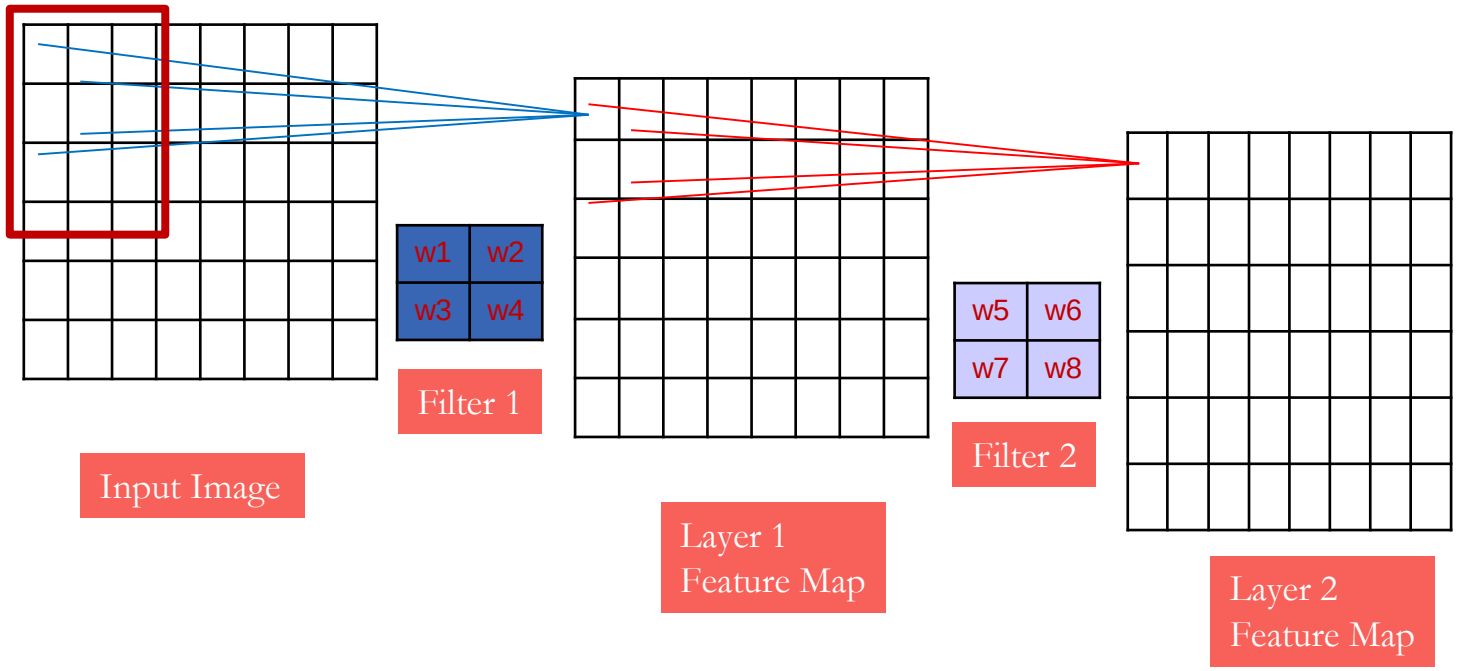
$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Input Image

Convoluted Image

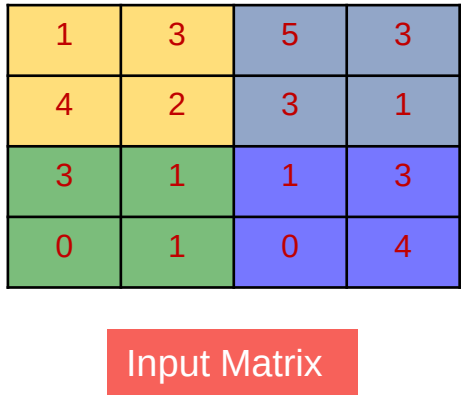
Convolutional Neural Networks (CNNs)

- 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

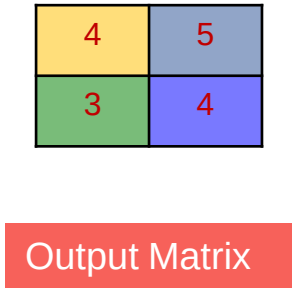


Convolutional Neural Networks (CNNs)

- ***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

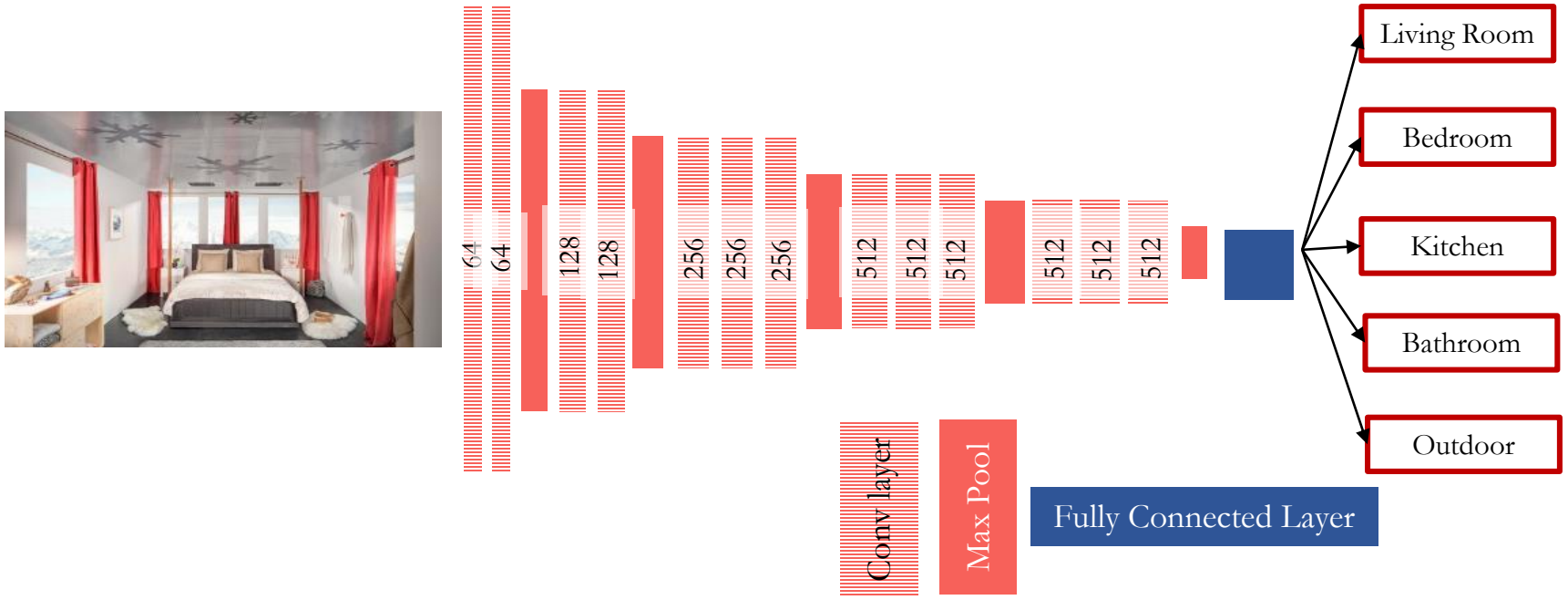


MaxPool with a 2×2 filter with stride of 2



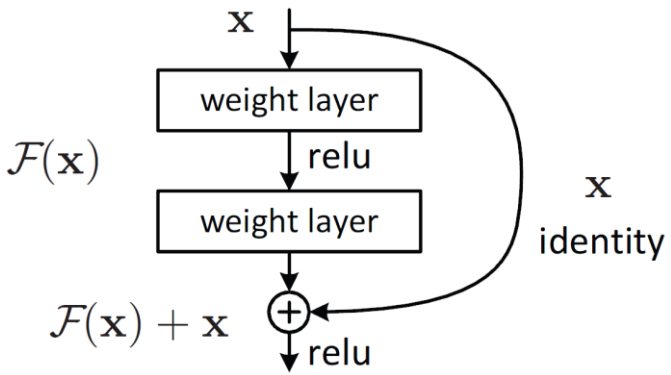
Convolutional Neural Networks (CNNs)

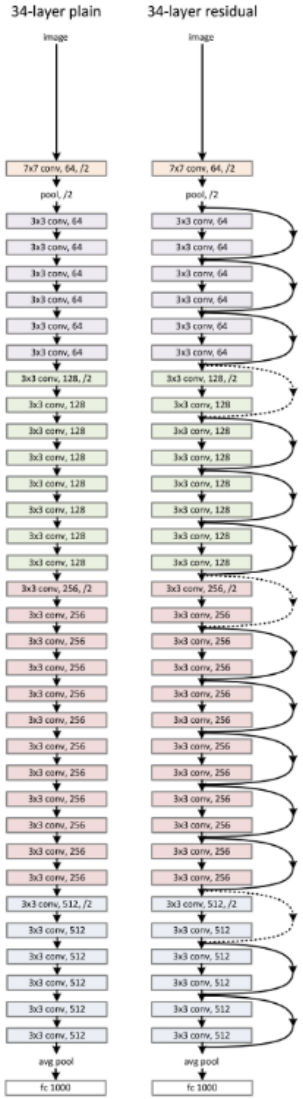
- Feature extraction architecture
  - After 2 convolutional layers, a max-pooling layer reduces the size of the feature maps (typically by 2)
  - A fully convolutional and a softmax layers are added last to perform classification



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



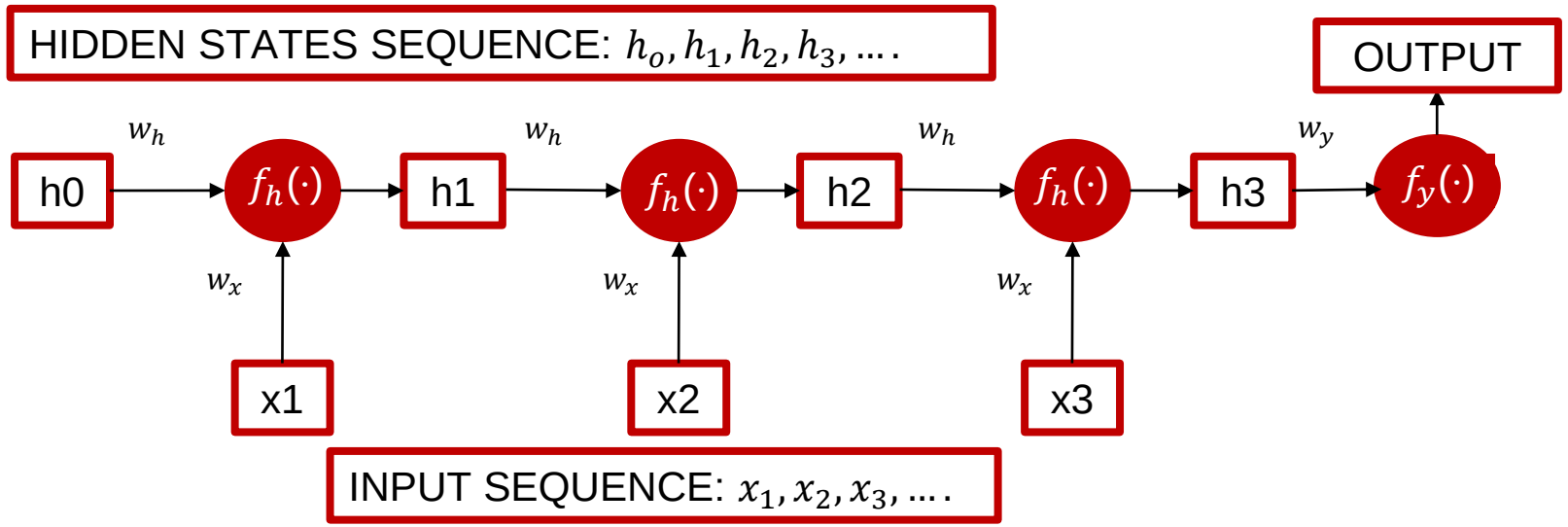


Recurrent Neural Networks (RNNs)

- ***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
  - 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 than CNNs
- 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

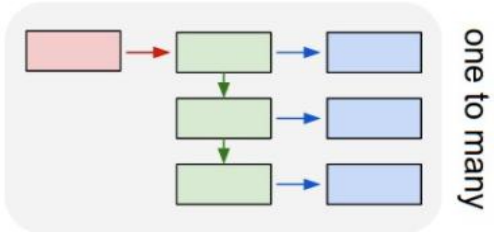
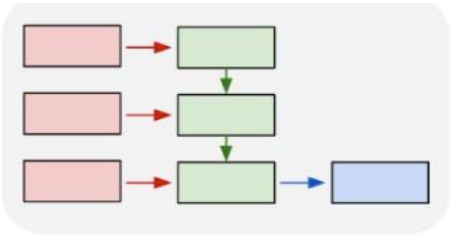
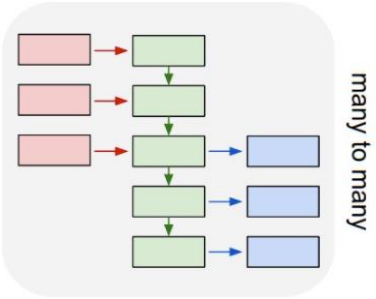
Recurrent Neural Networks (RNNs)

- RNN use same set of weights w_h and w_x across all time steps
  - A sequence of hidden states $\{h_o, h_o h_2, h_3, \ldots\}$ 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



Recurrent Neural Networks (RNNs)

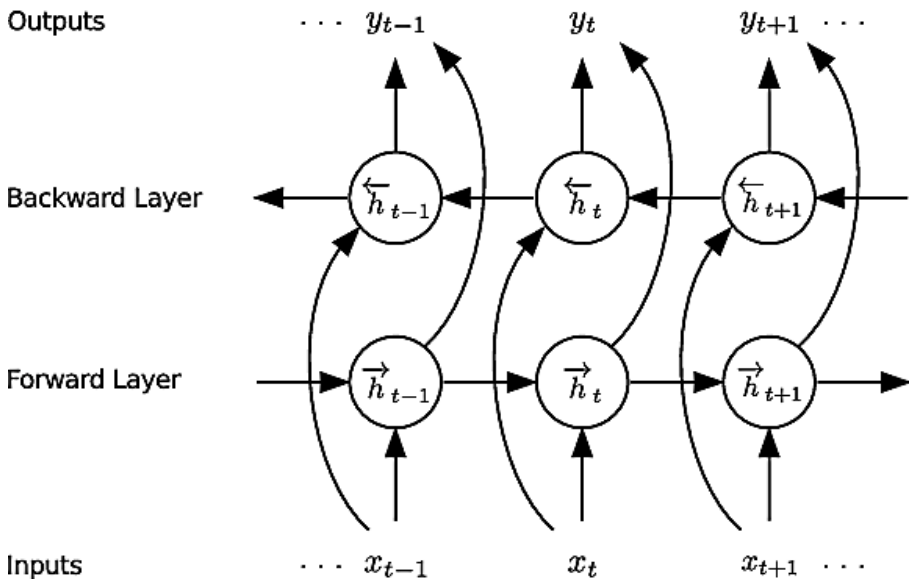
- RNNs can have one of many inputs and one of many outputs

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

Slide credit: Param Vir SIngh– Deep Learning

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



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

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

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

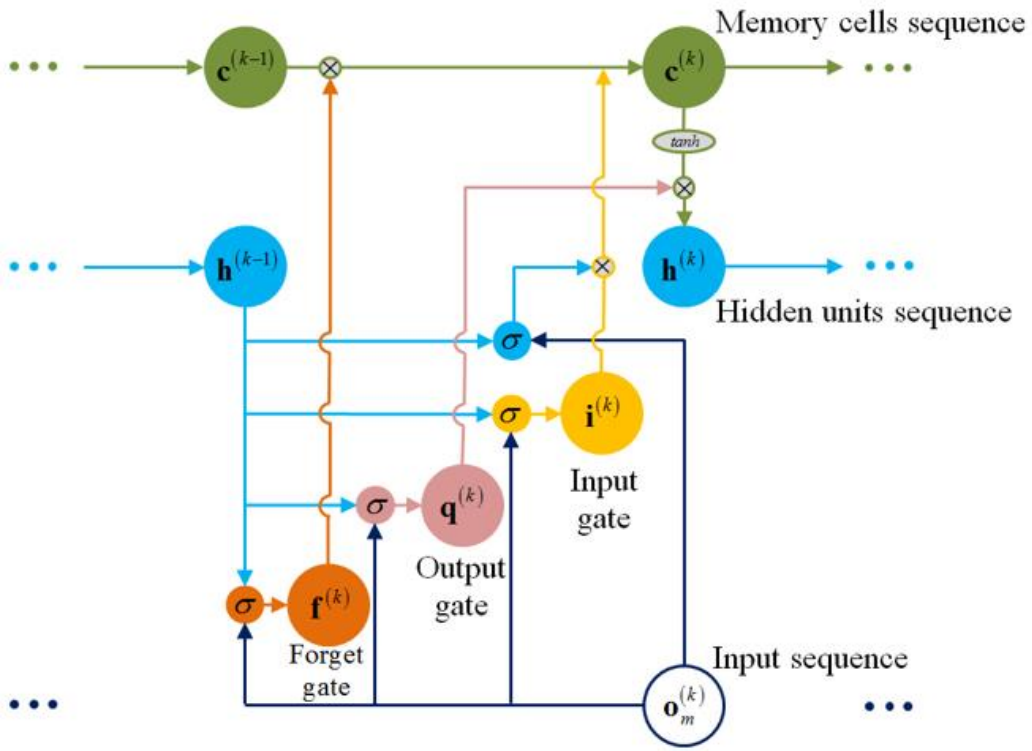
Outputs both past and future elements

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 data sequences



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

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

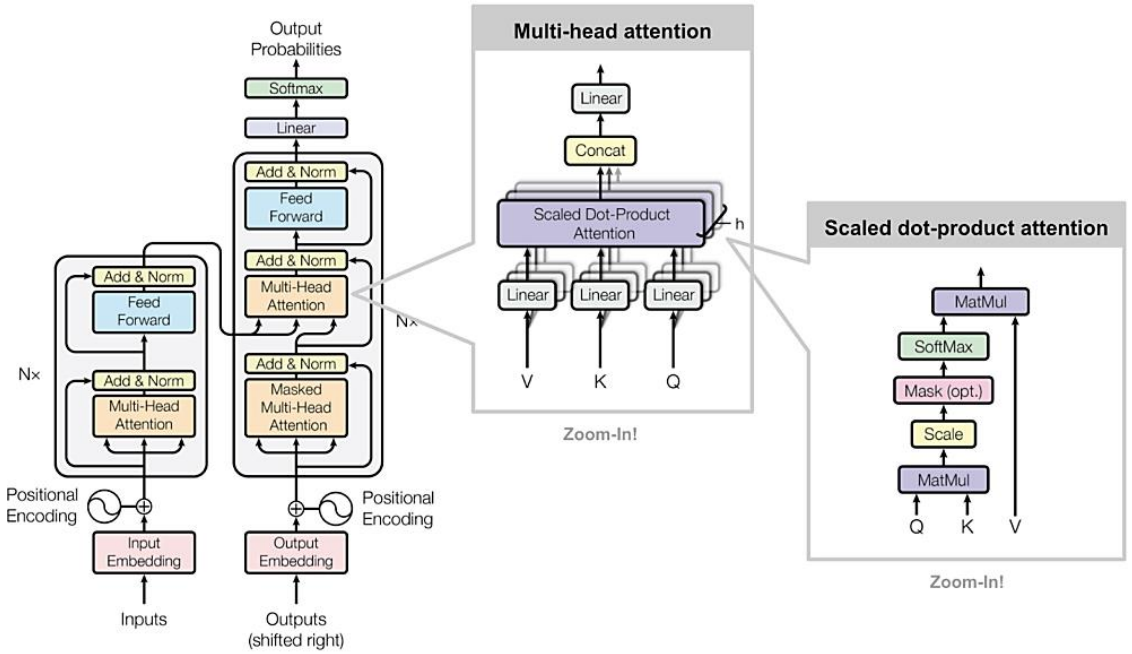
$$\mathbf{q}^{(k)} = \sigma\left(\mathbf{W}_{oq}\mathbf{o}_m^{(k)} + \mathbf{W}_{hq}\mathbf{h}^{(k-1)} + \mathbf{b}_q\right)$$

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

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

Transformer Networks

- ***Transformer networks*** have been initially designed for processing test data in Large Language Models, such as GPT-3, ChatGPT, etc.
  - 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



Generative adversarial network (GAN)

- Generative models are capable of making new plausible data.
- The term "adversarial" in this case pertains to a unique architecture for training an effective generator network.
- Random input vectors are given to the generator to make plausible samples that are ideally discriminated 50:50 in a fully trained model. However, some issue in training GANs include:
  - Vanishing gradient
    - Best if discriminator starts in a less robust state so it is improving along with the generator rather than being too sophisticated from the start
    - Modified minimax loss can help with this, proposed in original paper, maximizing log(D(G(z))) instead of minimizing it.
  - Mode Collapse
    - Local minimum in discriminator training, Wasserstein loss can help with this, uses critic model, D(x) – D(G(z)), rather than a threshold valued discriminator
  - Failure to Converge
    - Regularization of discriminator can help with this
    - Over training generator past random feedback from discriminator must be monitored

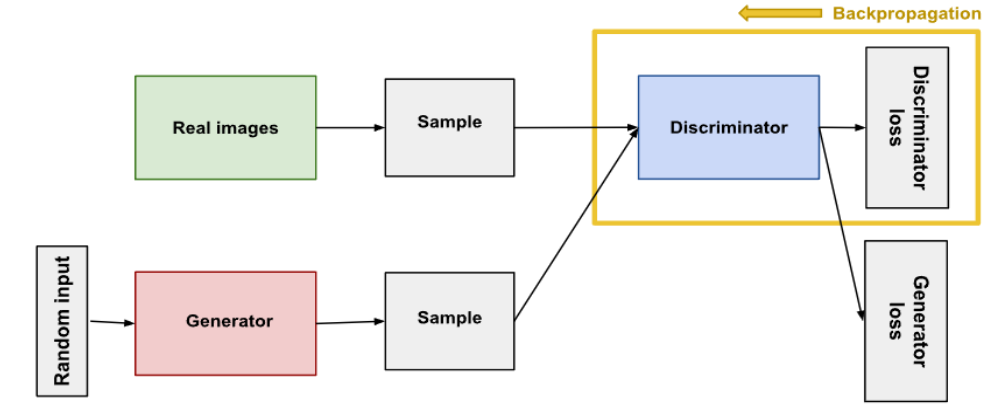


Figure 1: Backpropagation in discriminator training.

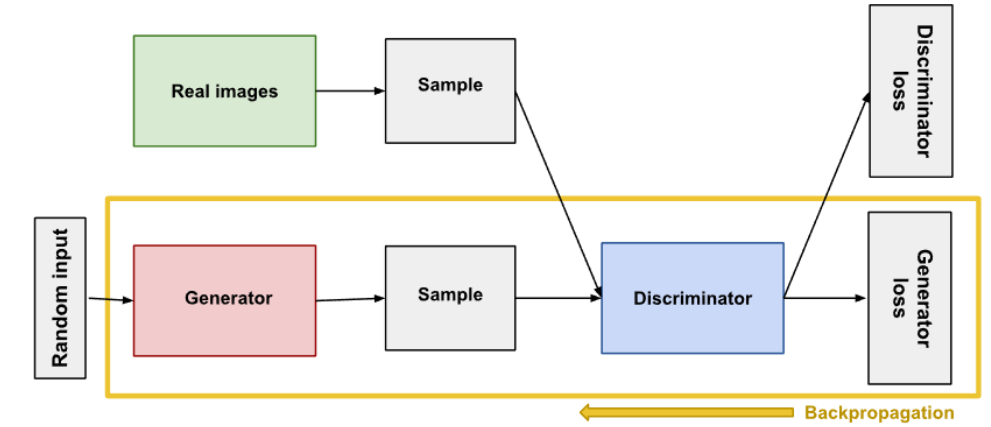


Figure 1: Backpropagation in generator training.

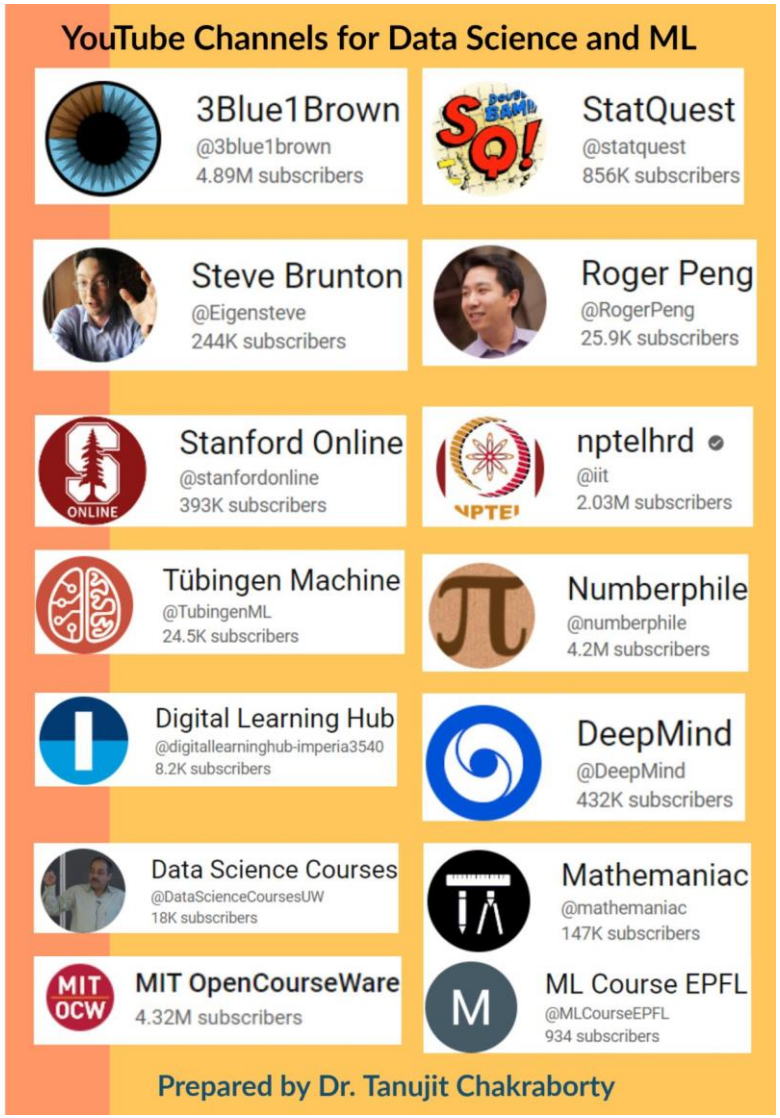
https://developers.google.com/machine-learning/gan/discriminator

Generative Adversarial Networks. Goodfellow 2014

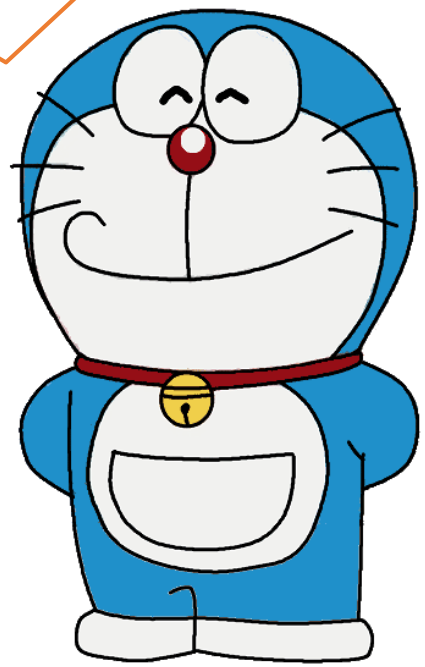
Slide Acknowledgements

1. Hung-yi Lee – Deep Learning Tutorial
2. Ismini Lourentzou – Introduction to Deep Learning
3. James Hays, Brown – Machine Learning Overview
4. Aleksandar (Alex) Vakanski— Deep Learning Presentation
5. Param Vir Singh, Shunyuan Zhang, Nikhil Malik – Deep Learning
6. Sebastian Ruder – An Overview of Gradient Descent Optimization Algorithms (link)

Other Useful Resources



Happy Learning



https://www.deeplearningbook.org/
https://d2l.ai/