Neural Network Basics

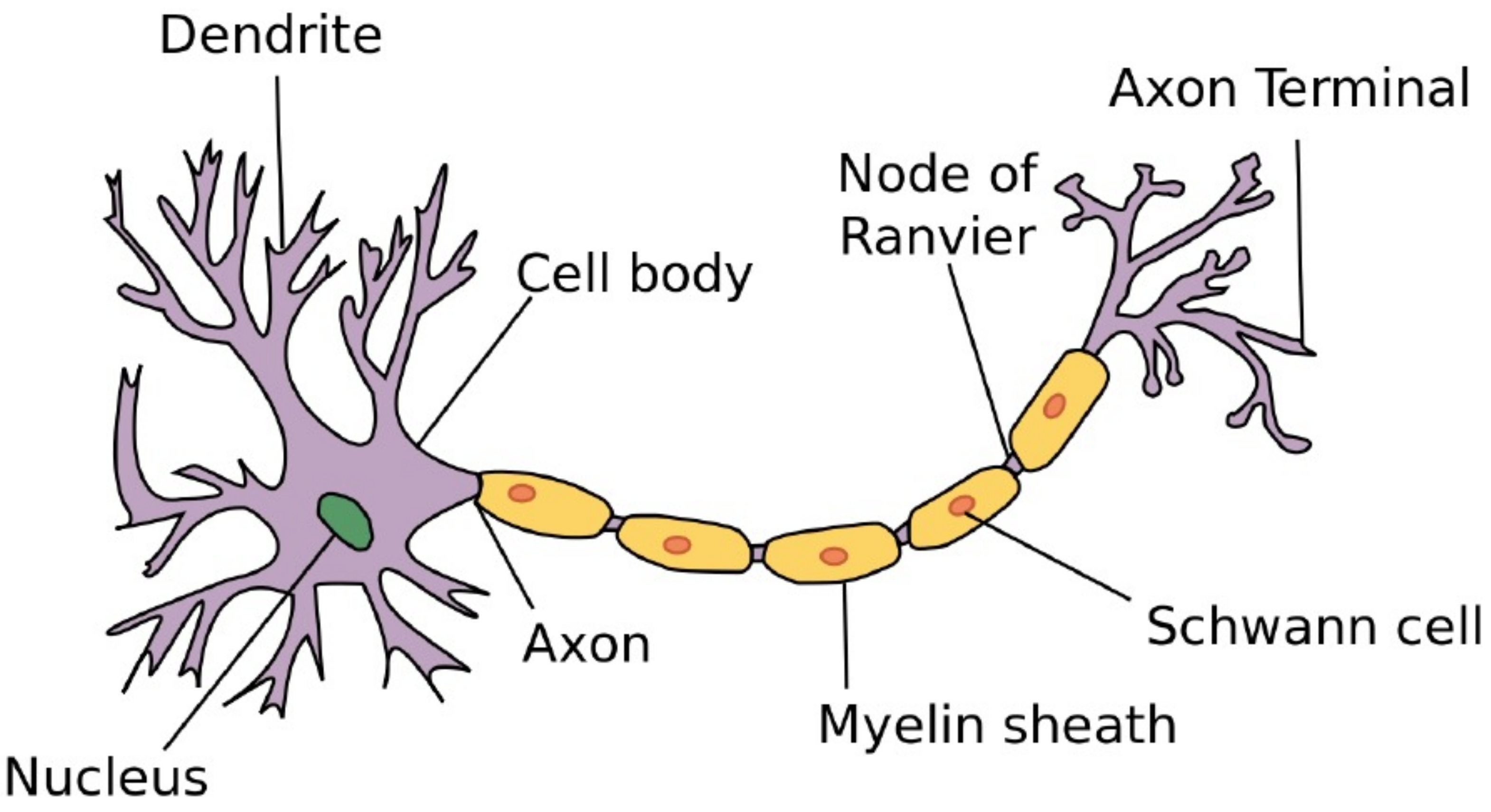
Dr. Unchalisa Taetragool

Department of Computer Engineering, Faculty of Engineering
King Mongkut's University of Technology Thonburi

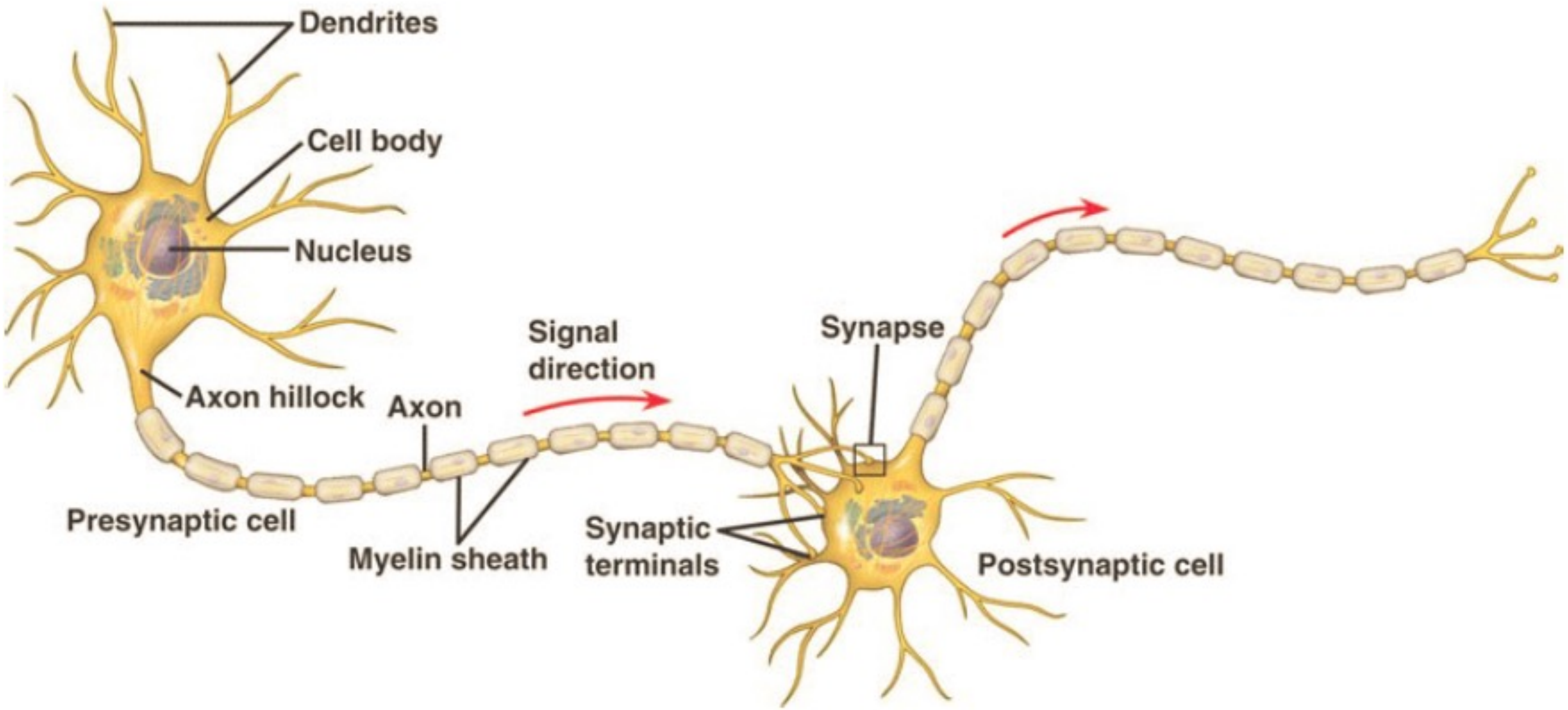
Outline

- Introduction to neural network
- Single neuron and simple network
- Training neural network
- NN hyperparameters
- Avoiding overfitting through regularization

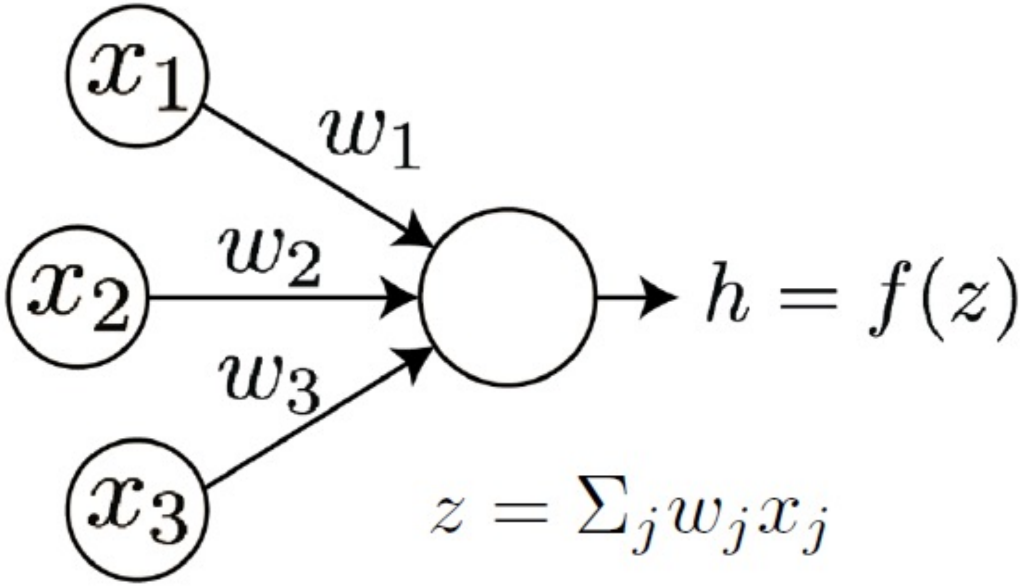
Human Neuron Cell



Human Neuron Interaction



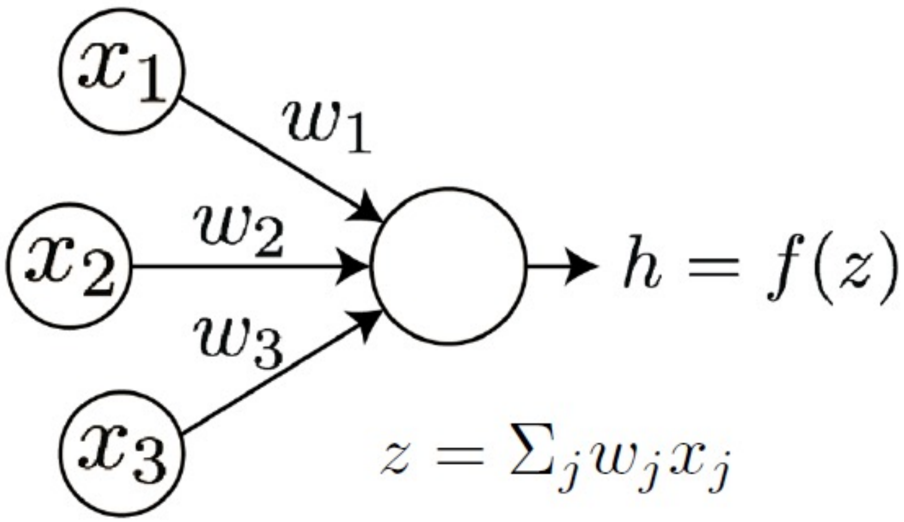
An Artificial Neuron

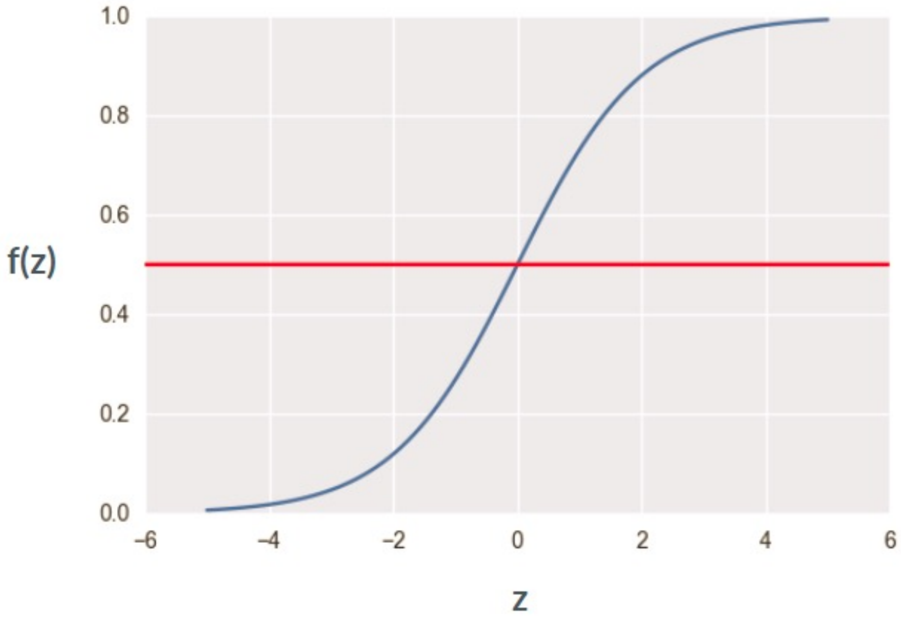


- **x**: features
- **w**: weights (parameters)
- **z**: sum of weights * features (neuron's current)
- **f**: activation function
- **h**: the model

The Activation Function

Signal received by a neuron (z) is passed to an "**Activation Function**" to get output (neural activity).

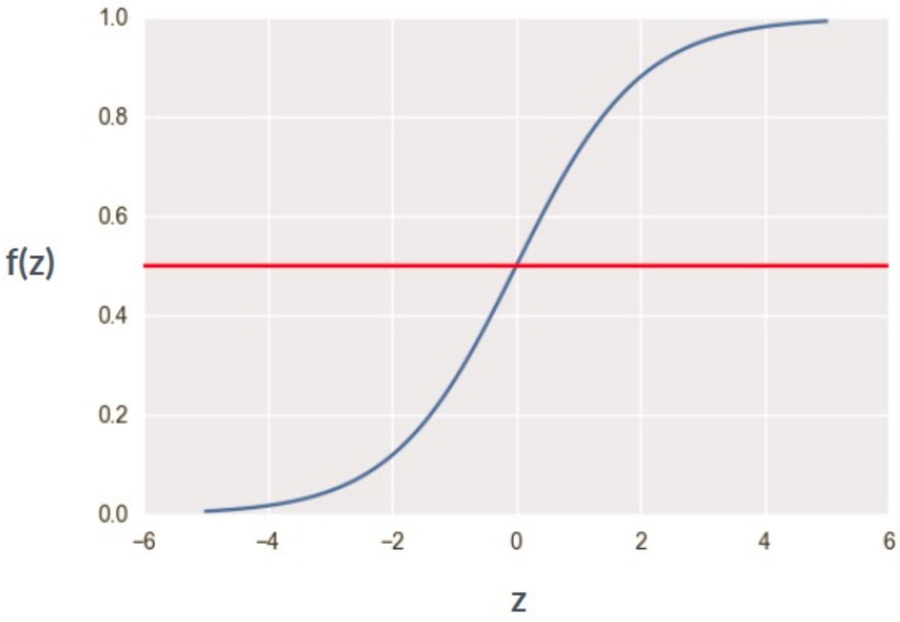




A common activation is **sigmoid** function.

The Activation Function

A common activation function is **sigmoid** or **logistic** function.



A **sigmoid** or **logistic** functional form:

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

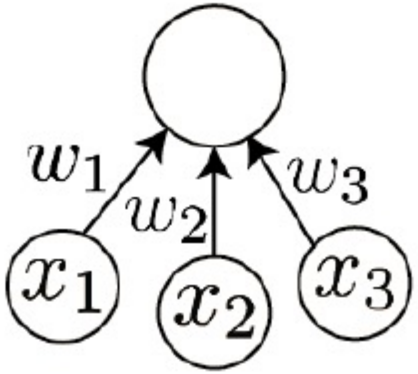
Single Neuron Quiz

Find the current input (z) into the top neuron. What class (y’) would the neuron predicts for each sample?

$$h = f(\Sigma_j w_j x_j)$$

x1	x2	x3	z	y′
1	0	0		
1	1	0		
1	0	1		
1	1	1		

w1	-1.5
w2	1
w3	1



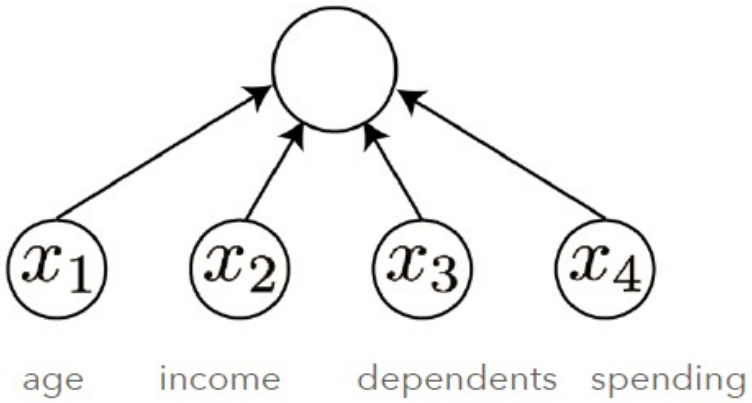
y’ = 1, if f(z) > 0.5 or z > 0
y’ = 0, if f(z) < 0.5 or z < 0

A Simple Example

Let's look at a simple 2-layer neural network for example:

Suppose we want to predict if a given bank customer will be good or bad loan taker.

$$h = f(\Sigma_j w_j x_j)$$



x1	x2	x3	x4	history
40	50	0	30	1
25	40	2	35	1
18	10	0	12	0
34	22	1	10	1

A Simple Example

We first need to do preprocessing, such as normalization and standardization.

x1	x2	x3	x4	history
40	50	0	30	1
25	40	2	35	1
18	10	0	12	0
34	22	1	10	1

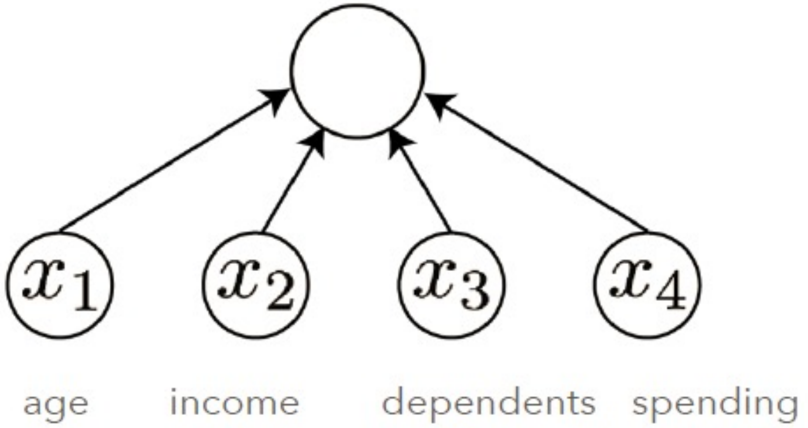
x1	x2	x3	x4	history
0.44	0.63	0	0.6	1
0.28	0.50	0.5	0.7	1
0.20	0.13	0	0.24	0
0.38	0.28	0.25	0.2	1

A Simple Example

Then fit the neural network to the data.

Suppose after fitting, here are the weight numbers.

$$h = f(\Sigma_j w_j x_j)$$

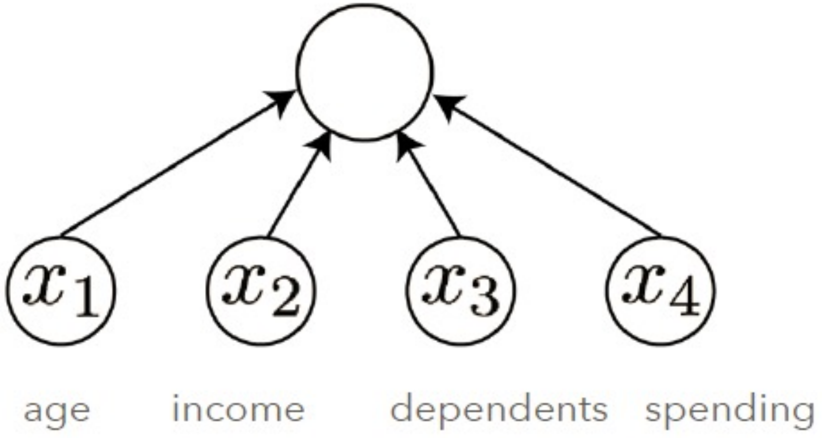


w1	0.7
w2	0.6
w3	-0.1
w4	-0.2

A Simple Example

Let us make prediction for a single customer…

$$h = f(\Sigma_j w_j x_j)$$



	X	W	X*W
age	0.44	0.7	0.31
income	0.63	0.6	0.38
dependent	0.00	-0.1	0.00
spending	0.60	-0.2	-0.12
		sum:	0.57
		h:	0.64

What the output means

Neural network classification

	X	W	X*W
age	0.44	0.7	0.31
income	0.63	0.6	0.38
dependent	0.00	-0.1	0.00
spending	0.60	-0.2	-0.12
		sum:	0.57
		h:	0.64

h indicates the probability of customer being good

h = 0.64

64% chance that he will be good

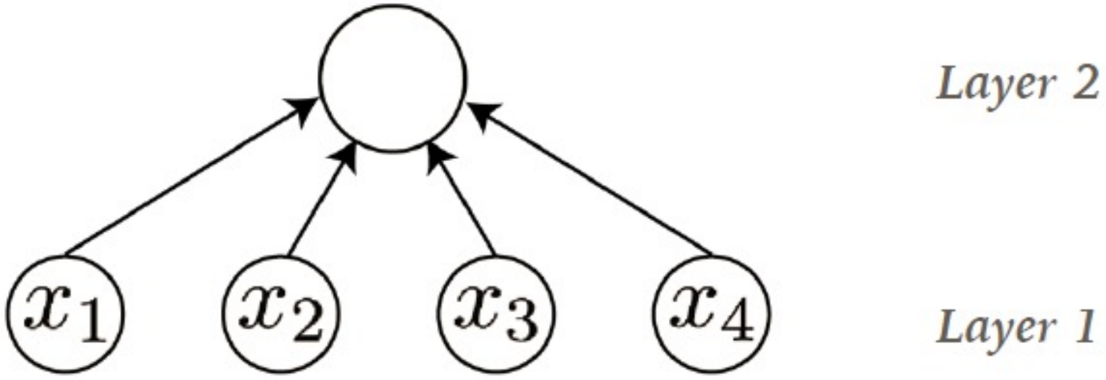
36% chance that he will be bad

Two layer neural network is almost equivalent to logistic regression.

2-Layer Perceptron

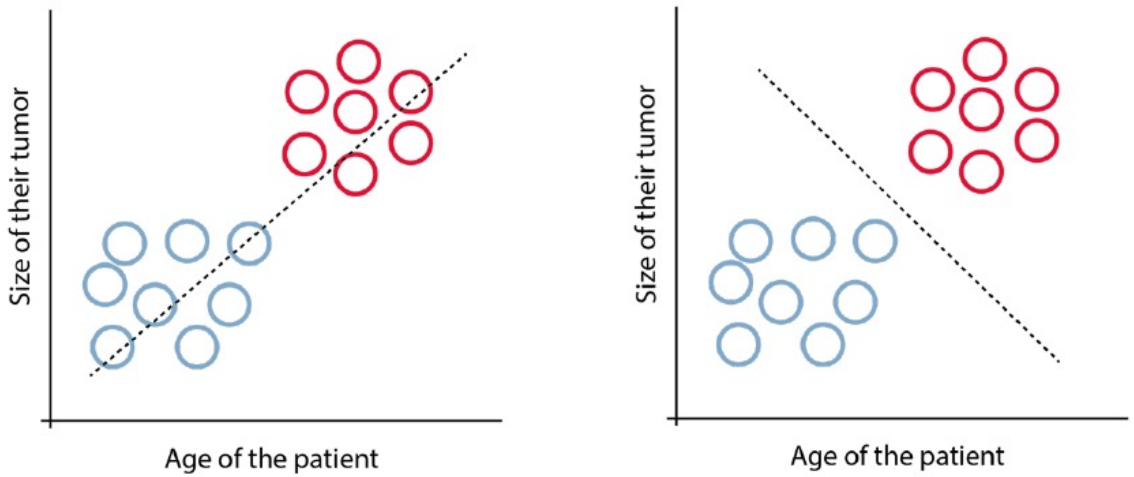
So far, we have seen only neural network with two layers, so called multilayer perceptron

$$h = f(\Sigma_j w_j x_j)$$



Linear Decision Boundary

2-layer neural network fits a linear decision boundary. It is the so-called "Linear Classifier." For high-dimension, you might think of it as a decision hyperplane.



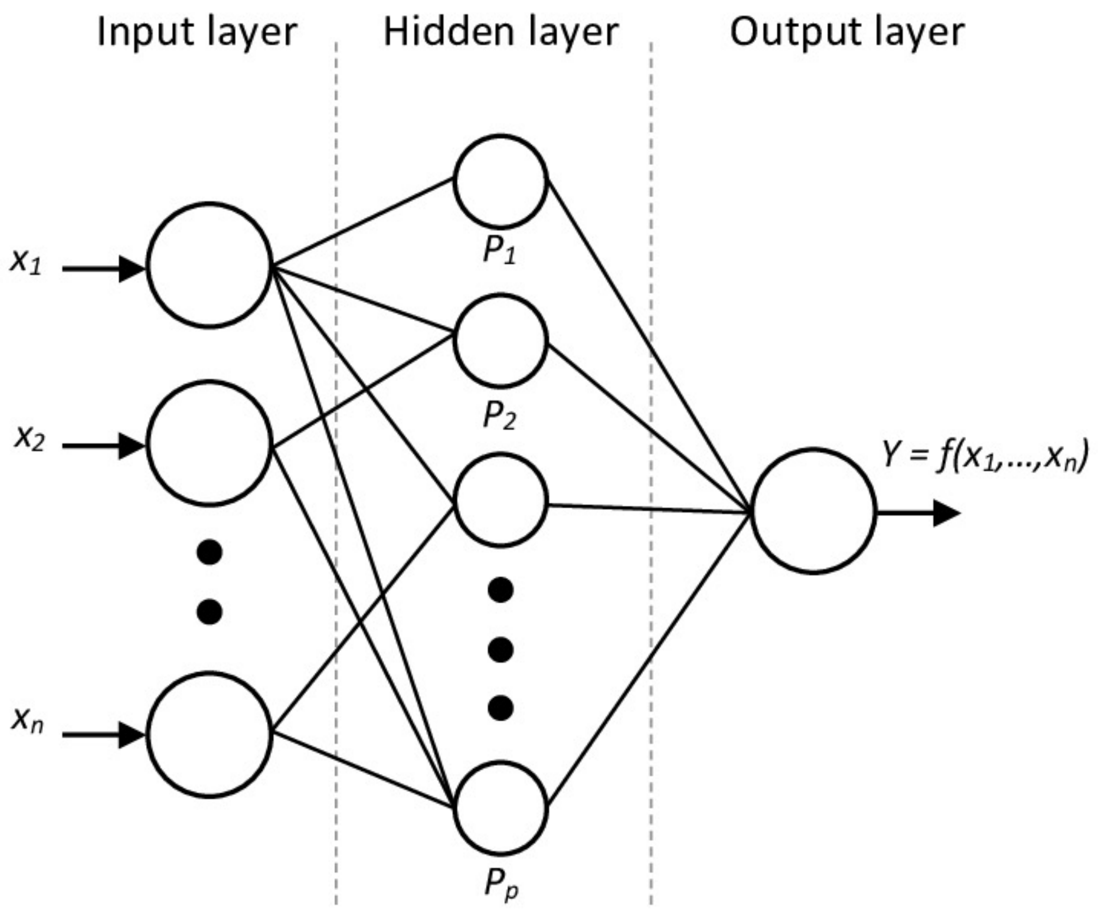
$w_1 = 0, w_2 = 1$

cost function high

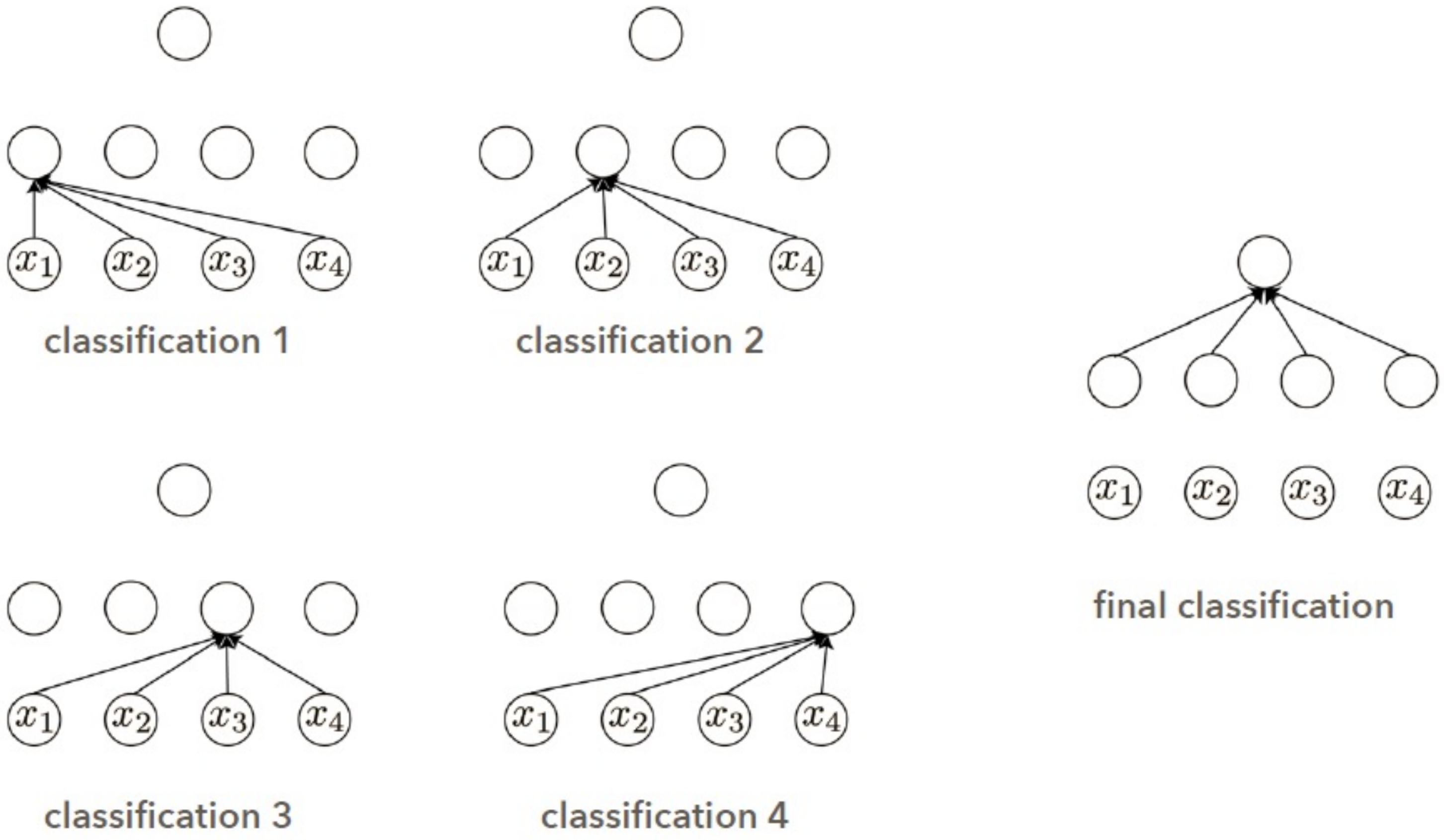
$w_1 = 1, w_2 = -1$

cost function low

Multilayer Perceptron (MLP)

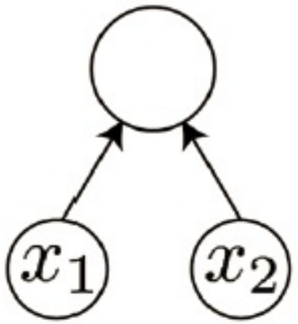


Multilayer Perceptron (MLP)

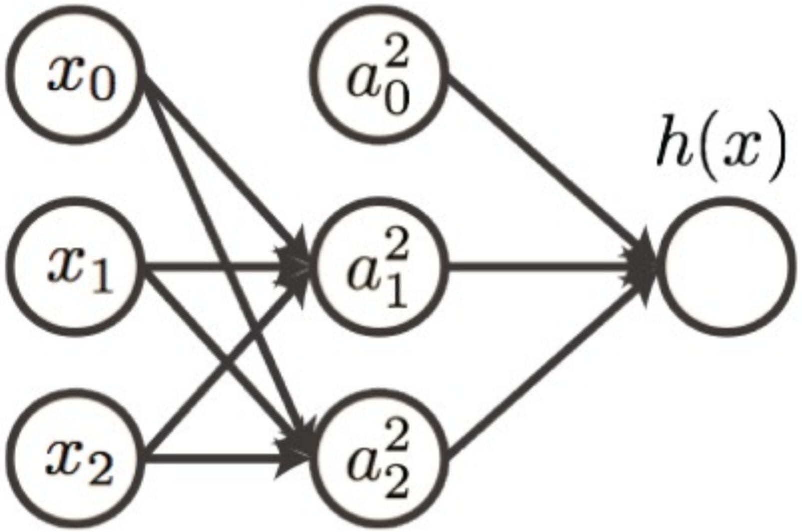


MLP Quiz

How many layers, units, and weights we have in these two neural networks?



Network 1



Network 2

Training Neural Network

- Each neuron receives input from their friends and pass those inputs through a logistic function. Each neuron performs classification
- It sends the vote (answer) to friends in the next layer
- Before training, each neuron is not accurate about its classification. The purpose of training is to adjust the weight to make these classifications more accurate.

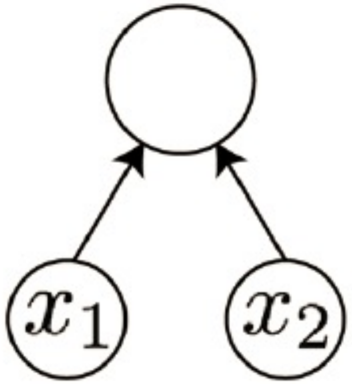
Training Neural Network

- Initialize neural network with number of layers and units we desire
- Initialize the network weights to be random numbers
- Measure the goodness of our neural network with a cost function (at first cost function should be high)
- Adjust the weights with a learning algorithm to minimize cost function

Training Neural Network Example

We will start simple by training our neural network to perform regression task with 2D input

$$h = f(\Sigma_j w_j x_j)$$



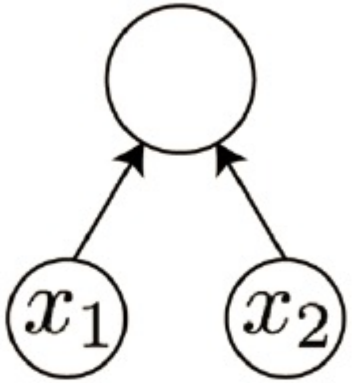
Linear Activation Function

$$f(z) = z$$

Training Neural Network Example

The purpose of regression is to adjust W to minimize regression cost function (sum of square error)

$$h = f(\Sigma_j w_j x_j)$$



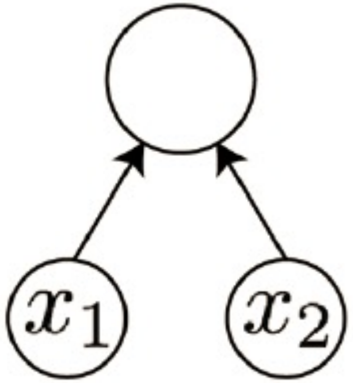
Cost Function

$$\begin{aligned} E(W) &= \Sigma_i E^i \\ &= \Sigma_i (h^i - y^i)^2 \\ &= \Sigma_i ((w_1 x_1^i - w_2 x_2^i) - y^i)^2 \end{aligned}$$

Weight Initialization

$$\vec{w} = \begin{bmatrix} w_1 & w_2 \end{bmatrix}$$

$$h = f(\Sigma_j w_j x_j)$$



- We will pick random numbers for the weights
- Note that the weights cannot start at zeros
- Often times, these random initial conditions are constrained to be small.

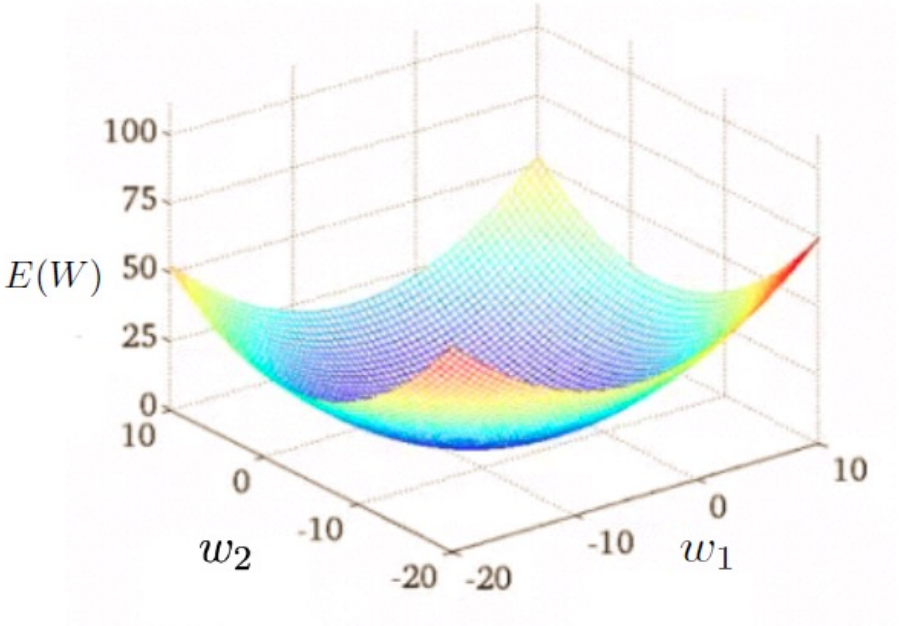
Weight Adjustment Algorithm

- In this example, we will use **gradient descent algorithm** to adjust weights.

$$\underbrace{w_{ij}}_{\text{new weight}} := \underbrace{w_{ij}}_{\text{old weight}} - \underbrace{\alpha}_{\text{learning rate}} * \underbrace{\frac{\partial\ Cost}{\partial\ w_{ij}}}_{\text{gradient of cost function}}$$

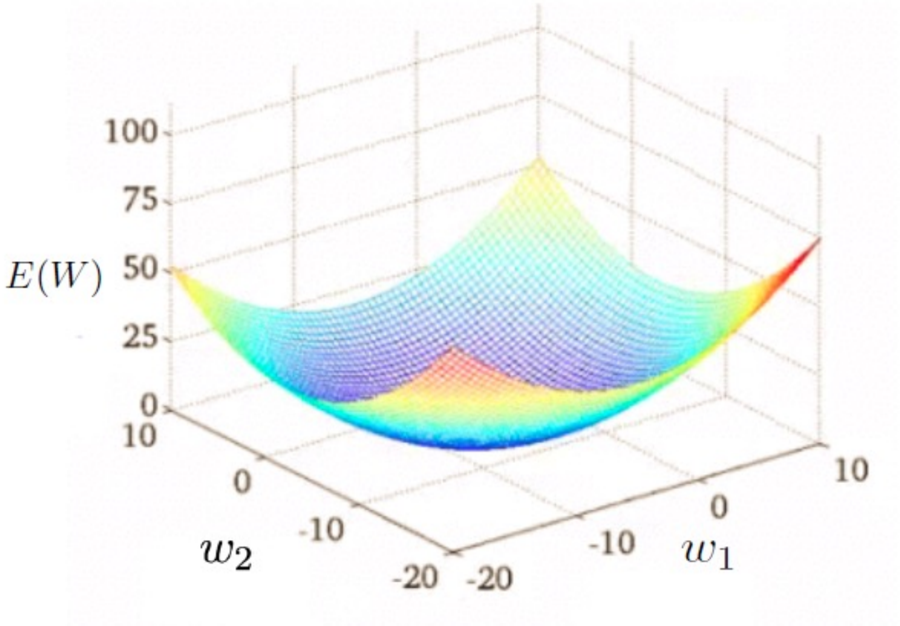
Understanding Gradient Descent

- In this example, we use gradient descent algorithm to adjust neural network weights.

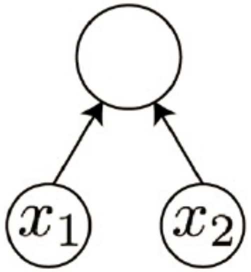


- To understand gradient descent, let’s start with understanding the idea of cost function landscape.
- Given a set of x and y, we can plot cost function as a function of w1 and w2.

Cost Function Landscape



$$h = f(\Sigma_j w_j x_j)$$

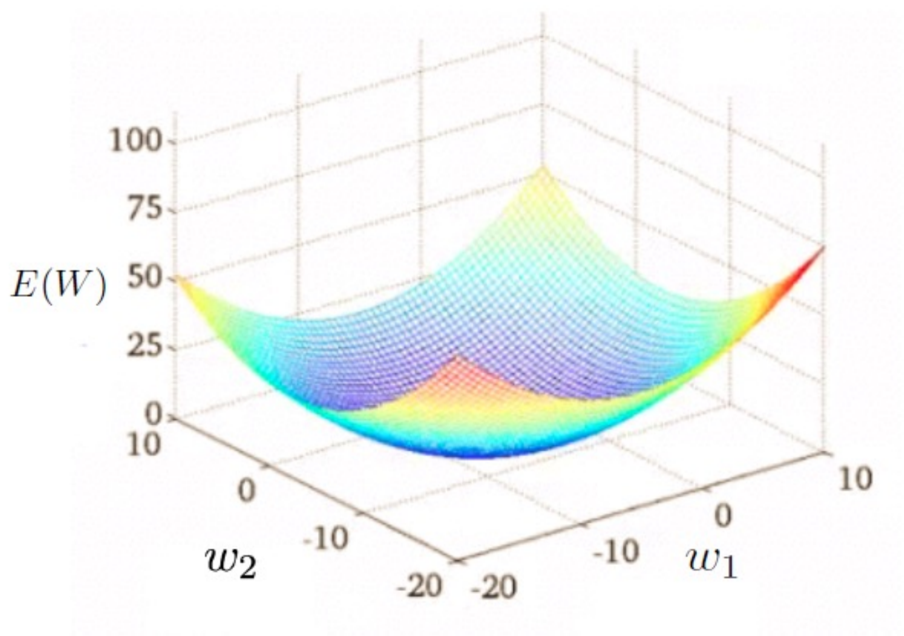


Cost Function

$$\begin{aligned} J(W) &= \Sigma_i E^i \\ &= \Sigma_i (h^i - y^i)^2 \\ &= \Sigma_i ((w_1 x_1^i - w_2 x_2^i) - y_i)^2 \end{aligned}$$

Gradient Descent Algorithm

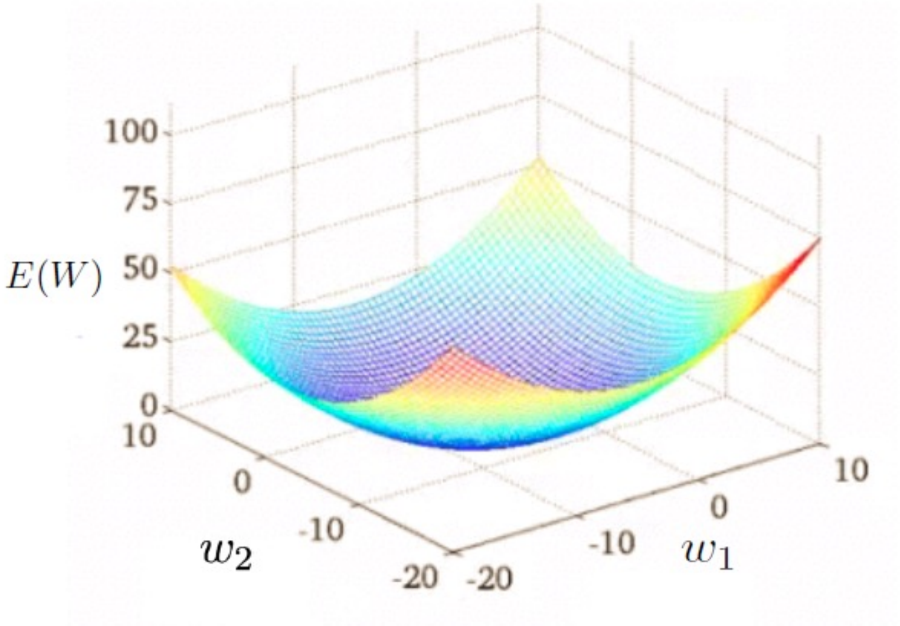
Getting to the bottom of the bowl with gradient descent



- Pick any pair of w1 and w2, getting dropped at any point in the landscape.
- Find a gradient at that point.
- Gradient $(-\nabla J(\theta_0, \theta_1))$ will always point to the steepest direction down the bowl.

Gradient Descent Algorithm

Getting to the bottom of the bowl with gradient descent

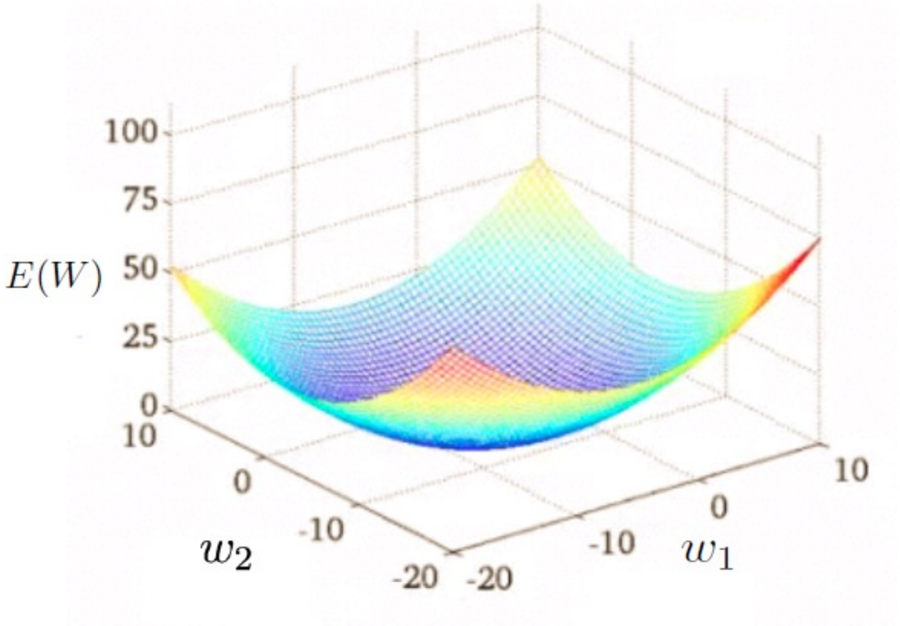


- Take a step down the bowl with the length of the footstep = α
- Each step, you will move from one point to another:

$$\begin{bmatrix}\theta_0\\ \theta_1\end{bmatrix} \rightarrow \begin{bmatrix}\theta_0\\ \theta_1\end{bmatrix} - \alpha \nabla J(\theta_0, \theta_1)$$

Gradient Descent Algorithm

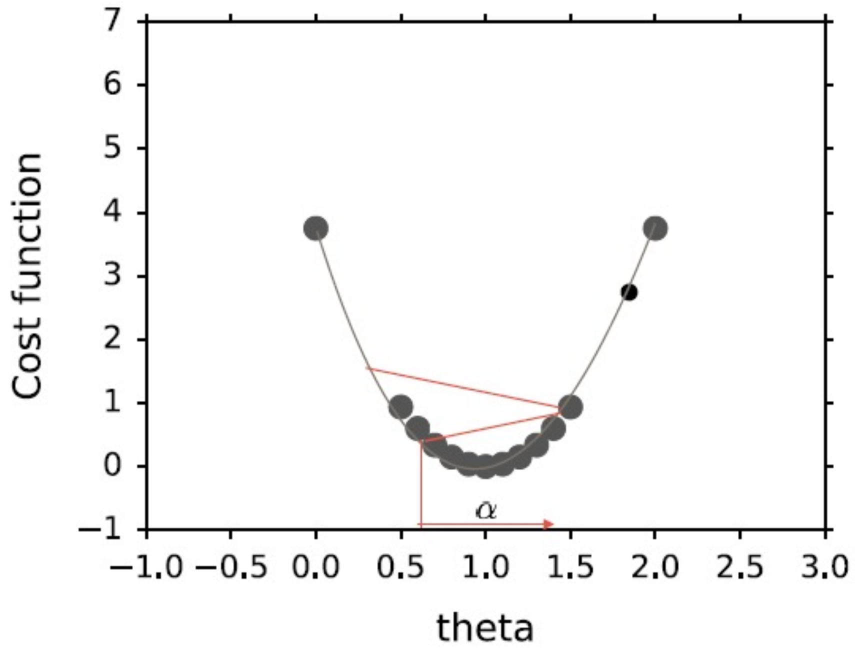
Summary of GD algorithm



- Continue walking with the same rule, over and over.
- Eventually, gradient will be around zero, and your step is tiny
- No matter where you step at the bottom, no lower points can be found
- At that point, you have reached the solution!

Picking the Right Alpha

$$w_{ij} := w_{ij} - \alpha * \frac{\partial\ Cost}{\partial\ w_{ij}}$$



- Alpha is usually between 0 and 1
- Large alpha: big step downhill
- Small alpha: small step
- You don't want too big or too small alpha

Picking the Right Alpha

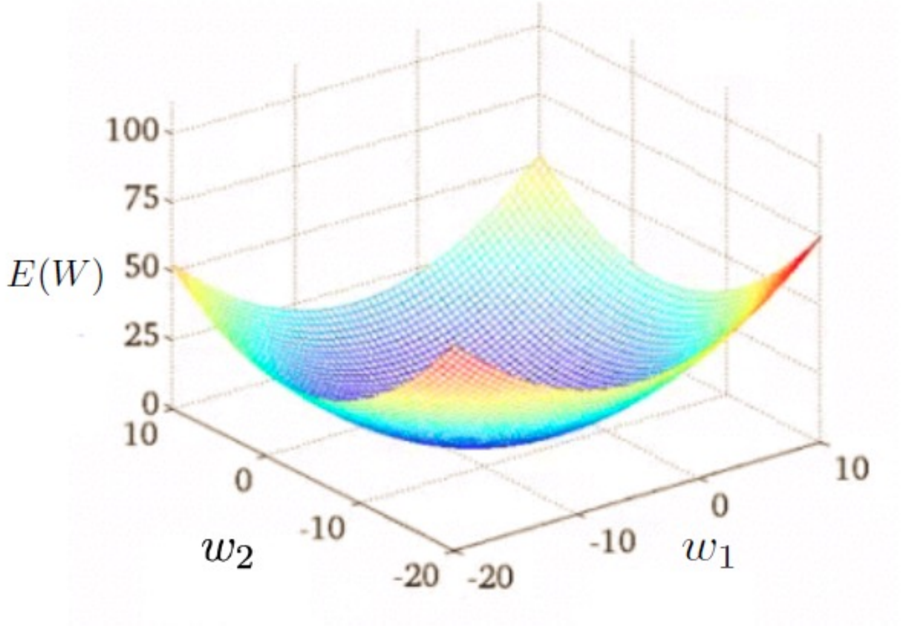
- The gradient is big at the top of the bowl and scale smaller at the bottom of the bowl
- $\nabla J(\theta_0, \theta_1)$ controls both direction and step size.
- So at the bottom of the bowl, you don't need to scale down the learning rate, because the decreasing gradient will take care of it.
- If you overshoot the minimum though you will not see it again.

Gradient Descent Summary

- Randomly initialize w1 and w2
- Calculate the gradient
- Update the algorithm with the following formula:

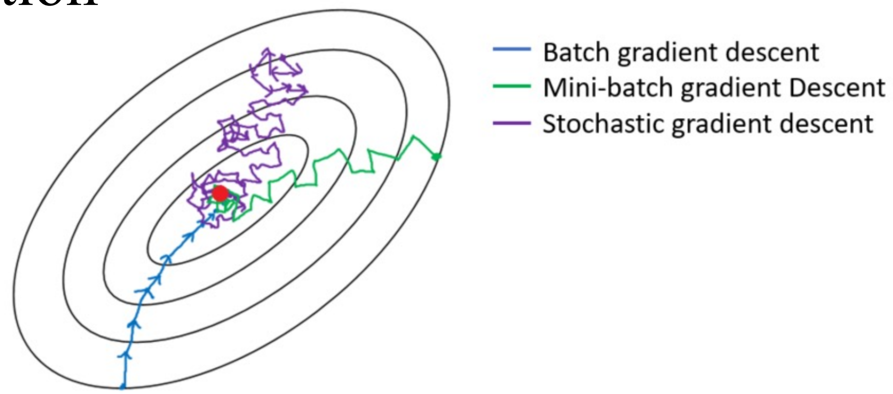
$$\begin{bmatrix}\theta_0 \\ \theta_1\end{bmatrix} \to \begin{bmatrix}\theta_0 \\ \theta_1\end{bmatrix} - \alpha \nabla J(\theta_0, \theta_1)$$

- Monitor the cost function. Cost should be lower any time you update alpha.
- When cost function is low enough, stop updating.



Batch/Mini-Batch/Stochastic Gradient Descent

- Batch gradient descent
  - Use all n training instances in each iteration
- Mini-batch gradient descent
  - Use b training instances in each iteration
- Stochastic gradient descent (SGD)
  - Use 1 training instance in each iteration

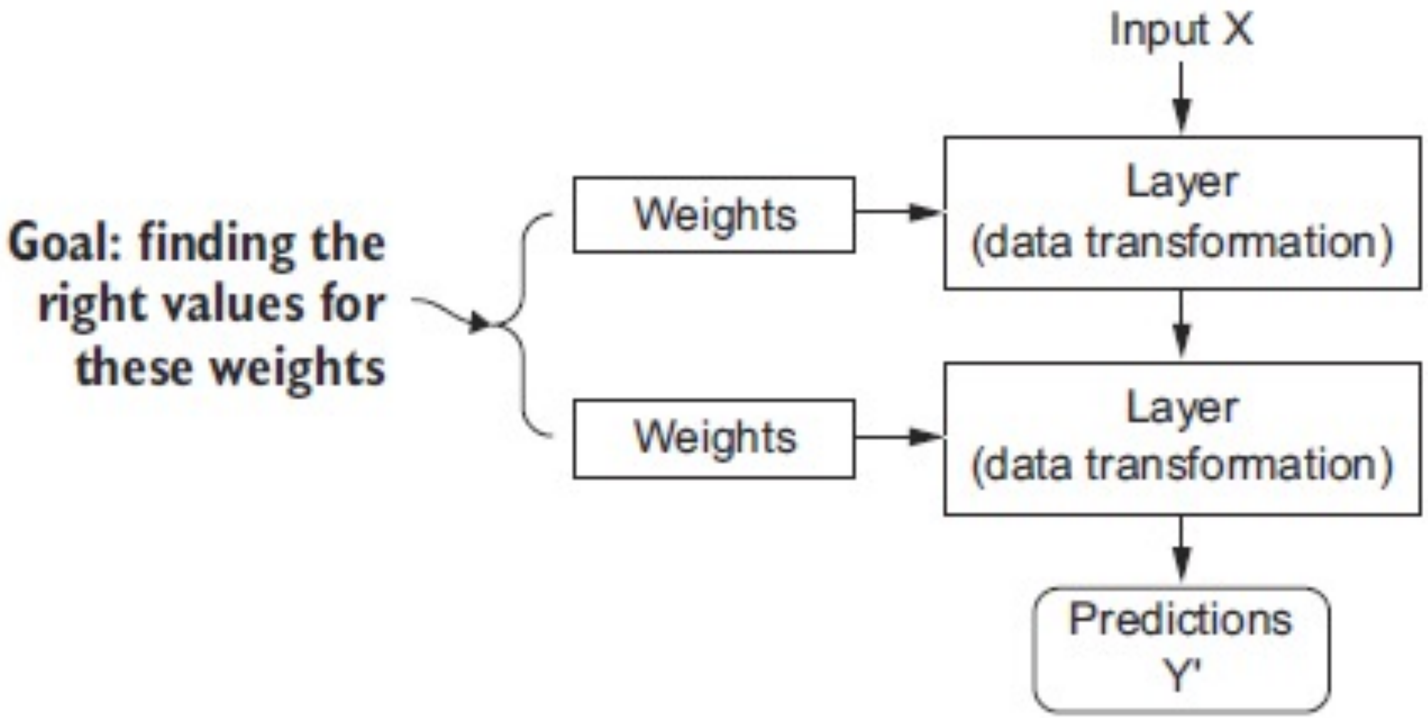


Epoch/Batch/Mini-Batch

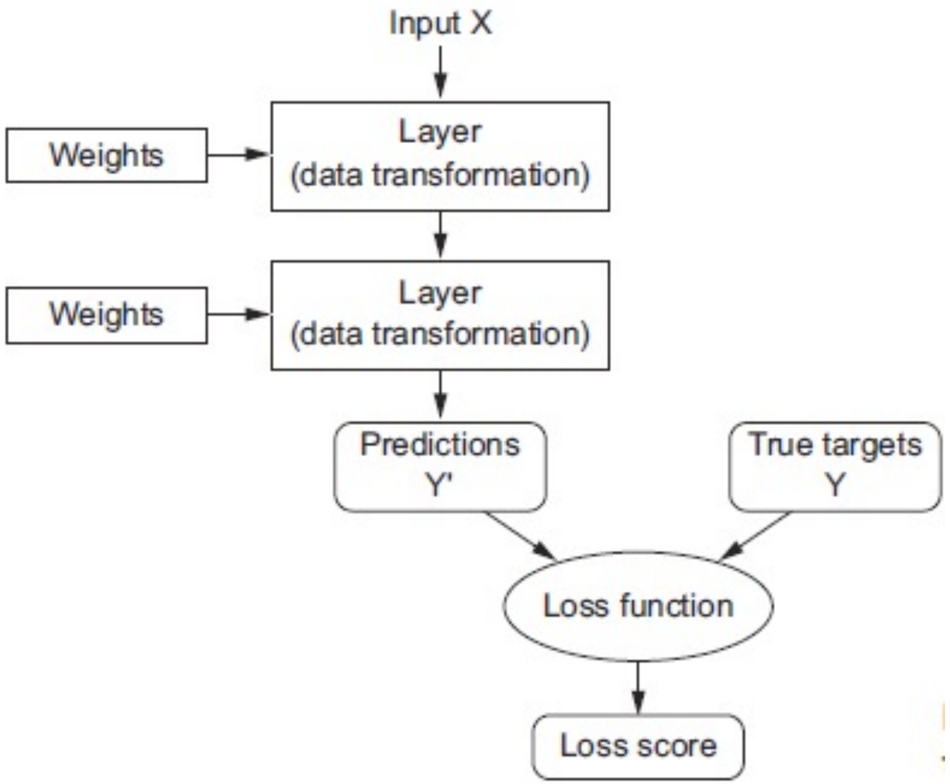
- Assuming you have n training instances
- 1 Epoch means your algorithm sees **EVERY** training instance once
- Batch update:
  - Every parameter update requires your algorithm see each of the n instances exactly once
  - Every epoch, your parameters are updated once
- Mini-batch update with batch size = b:
  - Every parameter update requires your algorithm see b of n instances
  - Every epoch, your parameters are updated about n/b times
- SGD update:
  - Every parameter update requires your algorithm see 1 of n instances
  - Every epoch, your parameters are updated about n times

Stochastic Gradient Descent

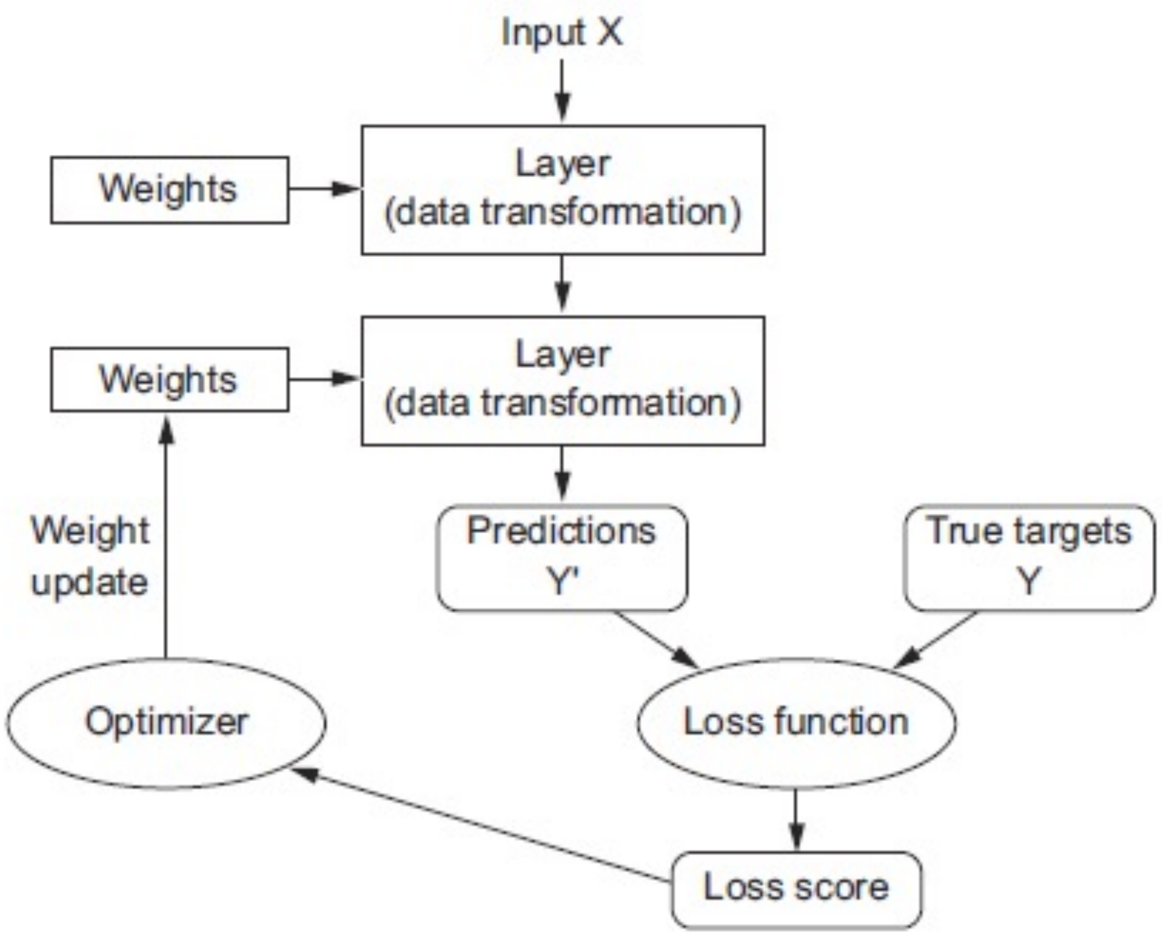
How Neuron Networks Work



How Neuron Networks Work



How Neuron Networks Work



Data representations for neural networks

- Data stored in multidimensional Numpy “arrays”, also called “tensors”
- A tensor is a container for data—almost always numerical data!

Scalars (0D tensors)

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

In the context of tensors,
a *dimension* is often called an *axis*

Vectors (1D tensors)

```
>>> x = np.array([12, 3, 6, 14])
>>> x
array([12, 3, 6, 14])
>>> x.ndim
1
```

Matrices (2D tensors)

```
>>> x = np.array([[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]])
>>> x.ndim
2
```

Real-World Examples of Data Tensors

- ***Vector data***—2D tensors of shape (samples, features)
- ***Timeseries data** or sequence data*—3D tensors of shape (samples, timesteps, features)
- ***Images***—4D tensors of shape (samples, height, width, channels) or (samples, channels, height, width)
- ***Video***—5D tensors of shape (samples, frames, height, width, channels) or (samples, frames, channels, height, width)

Neural Network Cost Function

Cost function: sum of errors from classifying $\sum_i (E_i)$

$$E_i = -(y^i log(h(x^i)) + (1 - y^i) log(1 - h(x^i)))$$

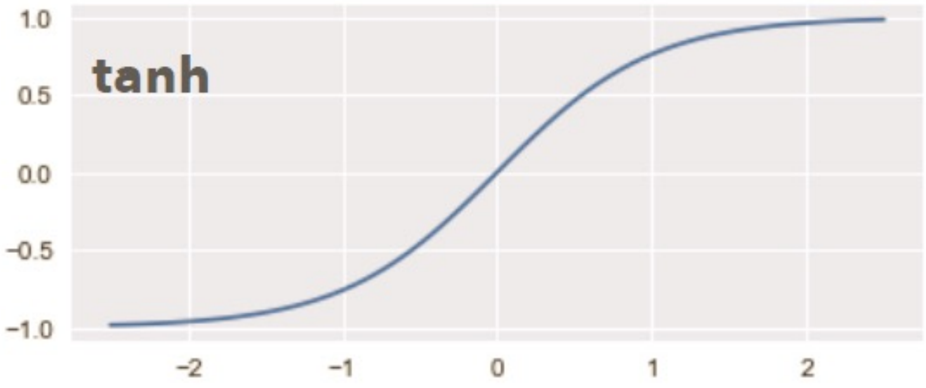
This cost function is 'cross entropy' error, defining how actual classes match your class predictions.

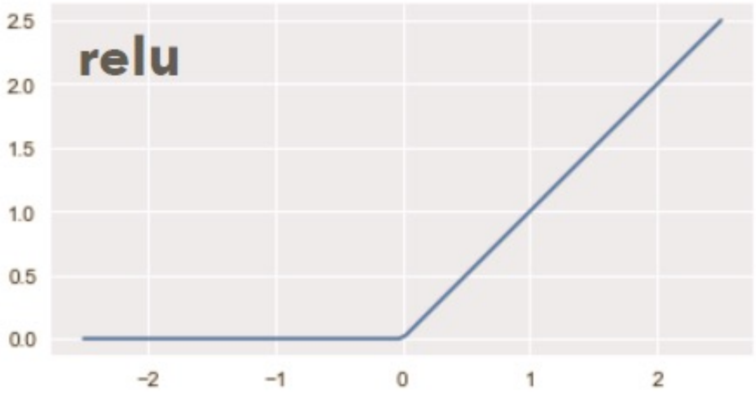
h	y	cost
0.8	1	0.10
0.1	1	1.00
0.1	0	0.05

Neural Network Hyperparameters

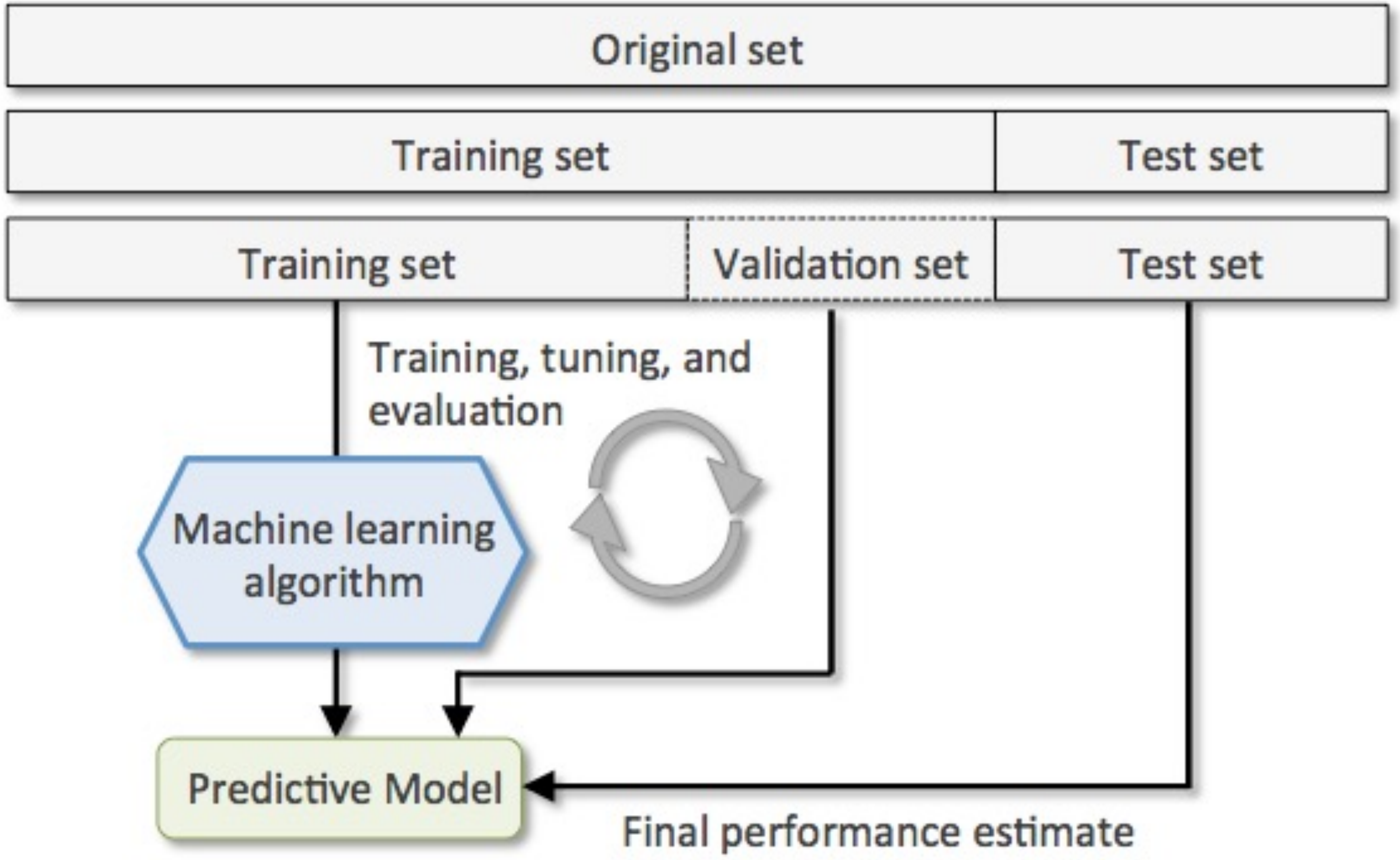
- Not only can you use any imaginable how neurons are interconnected, but even in a simple MLP you can change
  - the number of layers
  - the number of neurons per layer
  - the type of activation function to use in each layer
  - the weight initialization logic
  - and much more.

The Activation Functions



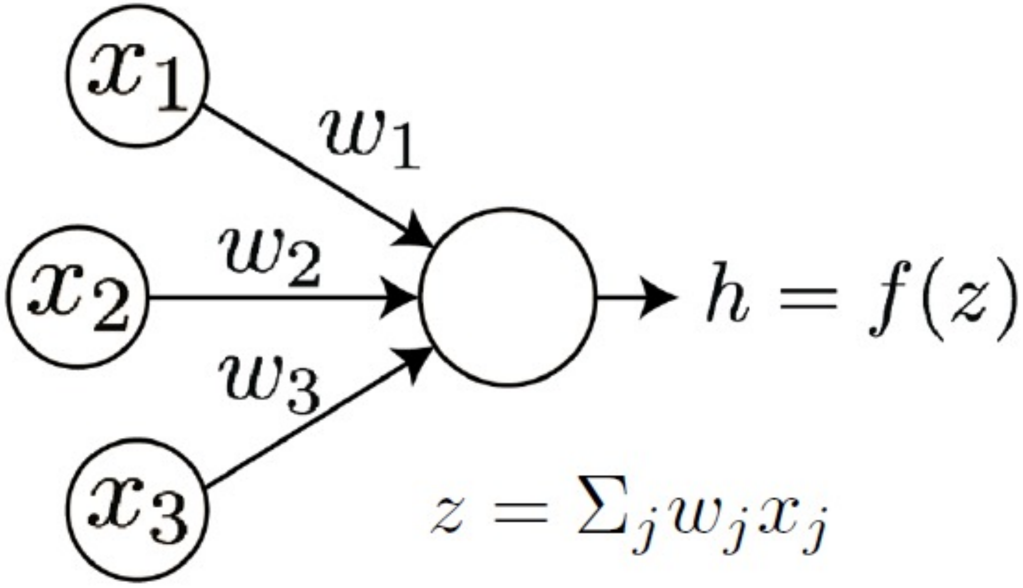


Training-Validating-Testing Data



Lab: Introduction to Keras

An Artificial Neuron



- **x**: features
- **w**: weights (parameters)
- **z**: sum of weights * features (neuron's current)
- **f**: activation function
- **h**: the model

An Artificial Neuron

```
import keras
from keras import models
from keras import layers
```

```
neuron = models.Sequential()
```

```
neuron.add(layers.Dense(1,
                        activation='relu',
                        input_shape=(3*1,),
                        kernel_initializer=initializers.RandomNormal(stddev=0.001)
                        ))
```

h: the model

- z: sum of weights * features
- f:activation function
- x: features

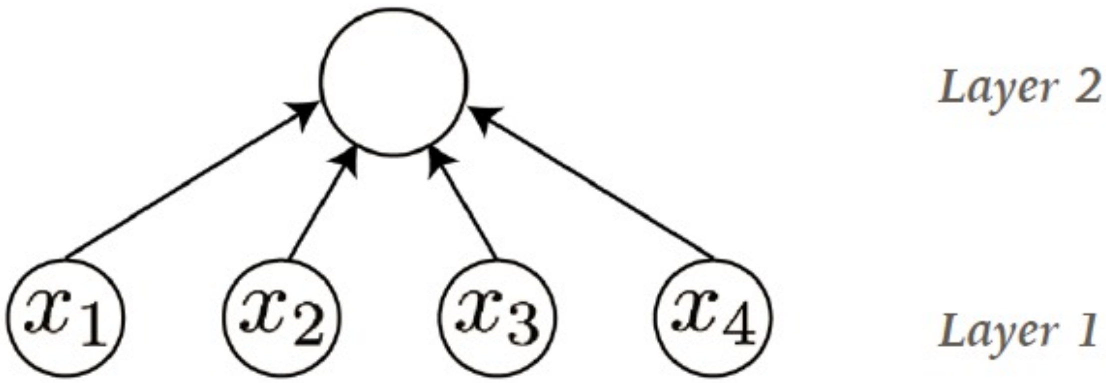
Input size is the shape
Of train data.

- w: weights

2-Layer Perceptron

So far, we have seen only neural network with two layers, so called multilayer perceptron

```
neuron.add(layers.Dense(4, activation='relu', input_shape=(3*1,),))
neuron.add(layers.Dense(1,activation='sigmoid'  ))
```



Training Neural Network in Keras

```
neuron.compile(optimizer='sgd',
               loss='categorical_crossentropy',
               metrics=['accuracy'])
```

```
neuron.summary()
```

```
history = neuron.fit(train_feature,
                     train_labels,
                     epochs=30,
                     batch_size=128,
                     validation_split=0.1)
```

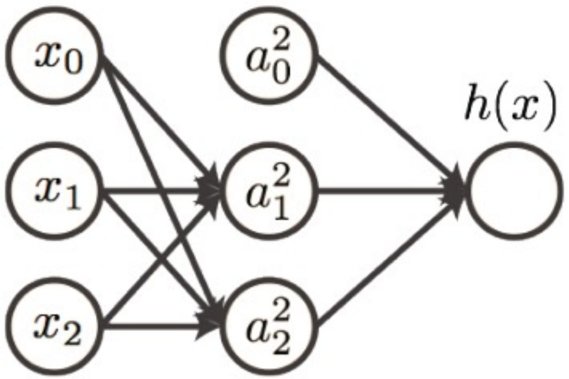
Hyperparameters in Neural Network

Hyperparameters in Neural Network

- NN Model
  - Number of layers
  - Number of neurons per layer
  - Type of activation function to use in each layer
  - Weight initialization logic
  - Optimizer
- Optimizer
  - Learning rate
  - Mini-batch size
  - Number of epochs

Bias Units

- Bias units allows us to add any constant to the computation of each layer
- In regression, this is similar to the term θ_0
- This provides a baseline for activity of neurons in each layer.



$$a_1^2 = g^2(W_{10}^1 x_0 + W_{11}^1 x_1 + W_{12}^1 x_2)$$

$$a_2^2 = g^2(W_{20}^1 x_0 + W_{21}^1 x_1 + W_{22}^1 x_2)$$

$$h(x) = g^3(W_{10}^2 a_0^2 + W_{11}^2 a_1^2 + W_{12}^2 a_2^2)$$

$$g^L(z) = z$$

BIG DATA EXPERIENCE

Number of Hidden Layers

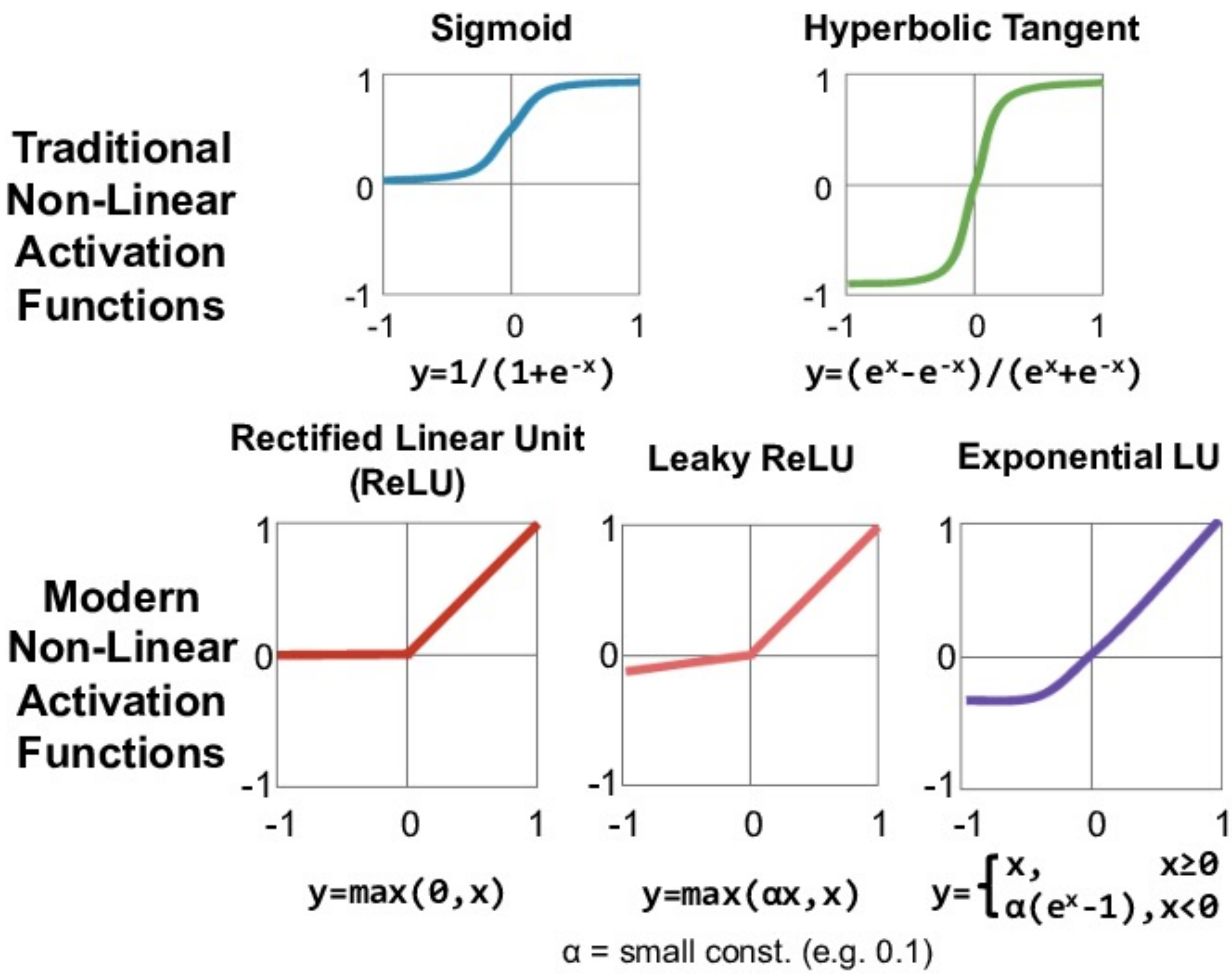
- For many problems, you can start with just one or two hidden layers and it will work just fine
- For more complex problems, you can gradually ramp up the number of hidden layers, until you start overfitting the training set
- Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers (or even hundreds, but not fully connected ones)

Number of Neurons per Hidden Layer

- Obviously, the number of neurons in the input and output layers is determined by the type of input and output your task requires
- As for the hidden layers, a common practice is to size them to form a funnel, with fewer and fewer neurons at each layer—the rationale being that many low-level features can merge into far fewer high-level features
  - You may simply use the same size for all hidden layers
- A simpler approach is to pick a model with more layers and neurons than you actually need, then use early stopping to prevent it from overfitting (and other regularization techniques, especially *dropout)*

Activation Functions

- Activation functions are used to **introduce nonlinearity** to models
- In most cases you can use the ReLU activation function in the hidden layers
  - It is a bit faster to compute than other activation functions
- For classification tasks, the softmax activation function is generally a good choice for the output layer
- For regression tasks, you can simply use no activation function at all



Sigmoid
Hyperbolic Tangent
Traditional Non-Linear Activation Functions
y=1/(1+e^-x)
y=(e^x-e^-x)/(e^x+e^-x)
Rectified Linear Unit (ReLU)
Leaky ReLU
Exponential LU
Modern Non-Linear Activation Functions
y=max(0,x)
y=max(αx,x)
y= x, x≥0 ; α(e^x-1), x<0
α = small const. (e.g. 0.1)

Activation Functions

	Pro	Con
Linear	• It gives a range of activations, so it is not binary activation	• Derivative is a constant. That means, the gradient has no relationship with X
ReLU	• It avoids and rectifies vanishing gradient problem	• It should only be used within hidden layers of a Neural Network Model.
Sigmoid	• It is nonlinear in nature • It's good for a classifier	• It gives rise to a problem of "vanishing gradients"
Tanh	• The gradient is stronger for tanh than sigmoid - derivatives are steeper	• Tanh also has the vanishing gradient problem

Weight Initialization

- Gradient descent uses randomness during initialization
  - in order to find a good enough set of weights for the specific mapping function from inputs to outputs that is being learned
- Why Not Set Weights to Zero?
  - Symmetric weights lead to symmetric gradient updates and the network can't force these 'neurons' to learn different things
  - Also lead to the **dying *ReLU*** problem $\boldsymbol{ReLU(z) = \max(0, z)}$
    - When the input to the ReLU is a negative number, the gradient is 0
    - If the gradients remain 0, and the network can no longer learn using this neuron

Weight Initialization

- Traditionally, the weights of a neural network were set to small random numbers.
- Keras offers a host of NN initialization methods
  - **Zeros**: Initializer that generates tensors initialized to 0.
  - **Ones**: Initializer that generates tensors initialized to 1.
  - **Constant**: Initializer that generates tensors initialized to a constant value.
  - **RandomNormal**: Initializer that generates tensors with a normal distribution.
  - **RandomUniform**: Initializer that generates tensors with a uniform distribution.
  - **TruncatedNormal**: Initializer that generates a truncated normal distribution.
  - **VarianceScaling**: Initializer capable of adapting its scale to the shape of weights.
  - **Orthogonal**: Initializer that generates a random orthogonal matrix.
  - **Identity**: Initializer that generates the identity matrix.
  - **lecun_uniform**: LeCun uniform initializer.
  - **glorot_normal**: Glorot normal initializer, also called Xavier normal initializer.
  - **he_uniform**: He uniform variance scaling initializer.

Faster Optimizers

- Training a very large deep neural network can be painfully slow
- Ways to speed up training (and reach a better solution)
  - applying a good initialization strategy for the connection weights
  - using a good activation function
  - using Batch Normalization
  - reusing parts of a pretrained network
  - using a faster optimizer than the regular Gradient Descent

Faster Optimizers

- Momentum optimization
  - A method that helps accelerate SGD in the relevant direction and dampens oscillations
  - When using momentum, we push a ball down a hill
    - The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way
  - The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions
- Nesterov Accelerated Gradient (NAG)
  - A smarter ball: it has a notion of where it is going so that it knows to slow down before the hill slopes up again

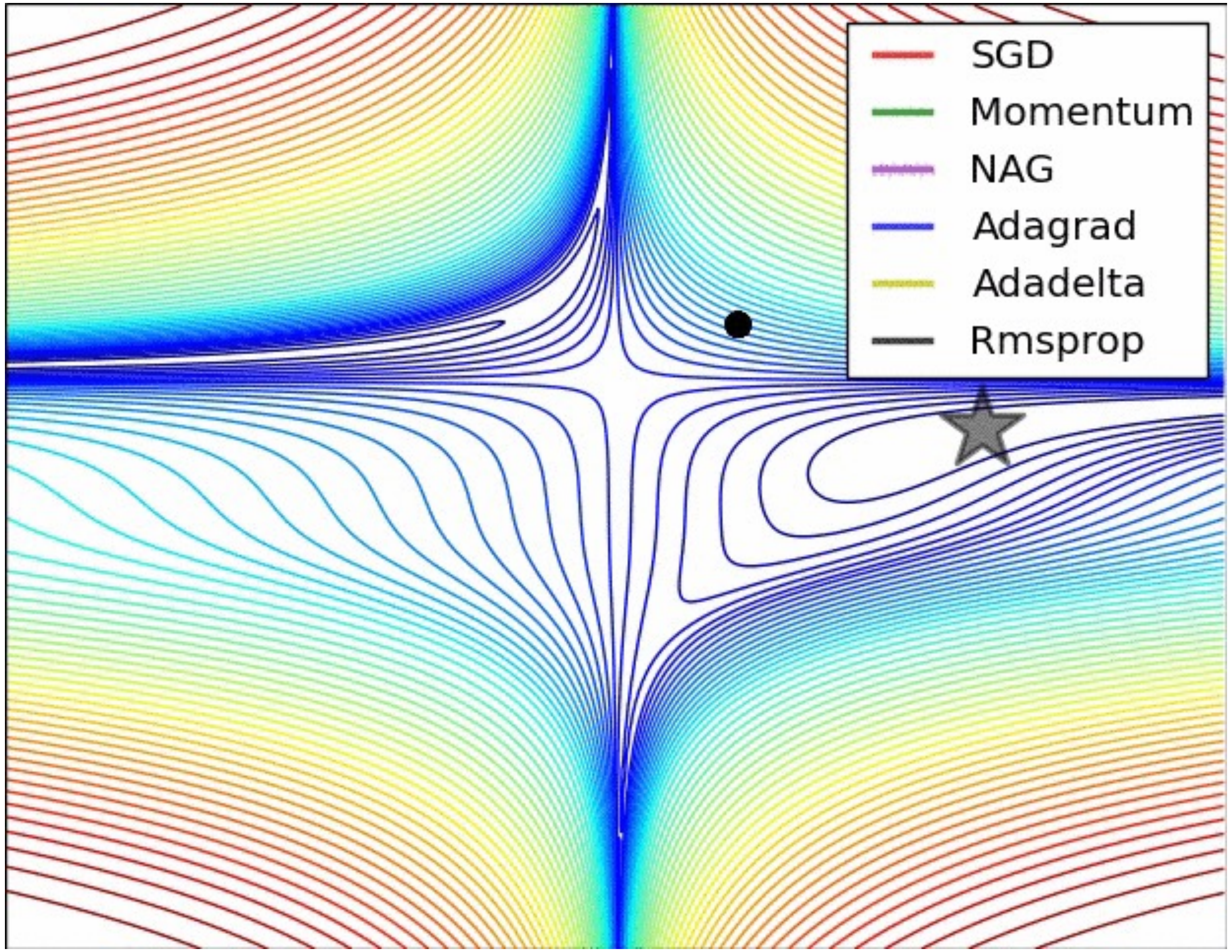
Faster Optimizers

- AdaGrad
  - An algorithm for gradient-based optimization
  - It adapts the learning rate to the parameters,
    - performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features
    - larger updates (i.e. high learning rates) for parameters associated with infrequent features

Faster Optimizers

- Adaptive Moment Estimation (Adam) Optimization
  - Another method that computes adaptive learning rates for each parameter
  - Adam also keeps an exponentially decaying average of past gradients, similar to momentum
    - like a heavy ball with friction

Faster Optimizers



Credit: ruder.io

Optimizer Hyperparameters: Learning Rate

Credit: towardsdatascience.com

Optimizer Hyperparameters

- Mini-batch size
  - A larger mini-batch size
    - utilizes matrix multiplication in the training calculations
    - but needs more memory for the training process
  - A smaller mini-batch size
    - induces more noise in error calculations
    - more useful in preventing the training process from stopping at local minima
  - Good value for mini-batch size = 32
- Number of epochs
  - Increase the number of epochs until the validation accuracy starts decreasing even when training accuracy is increasing (overfitting)

Overfitting

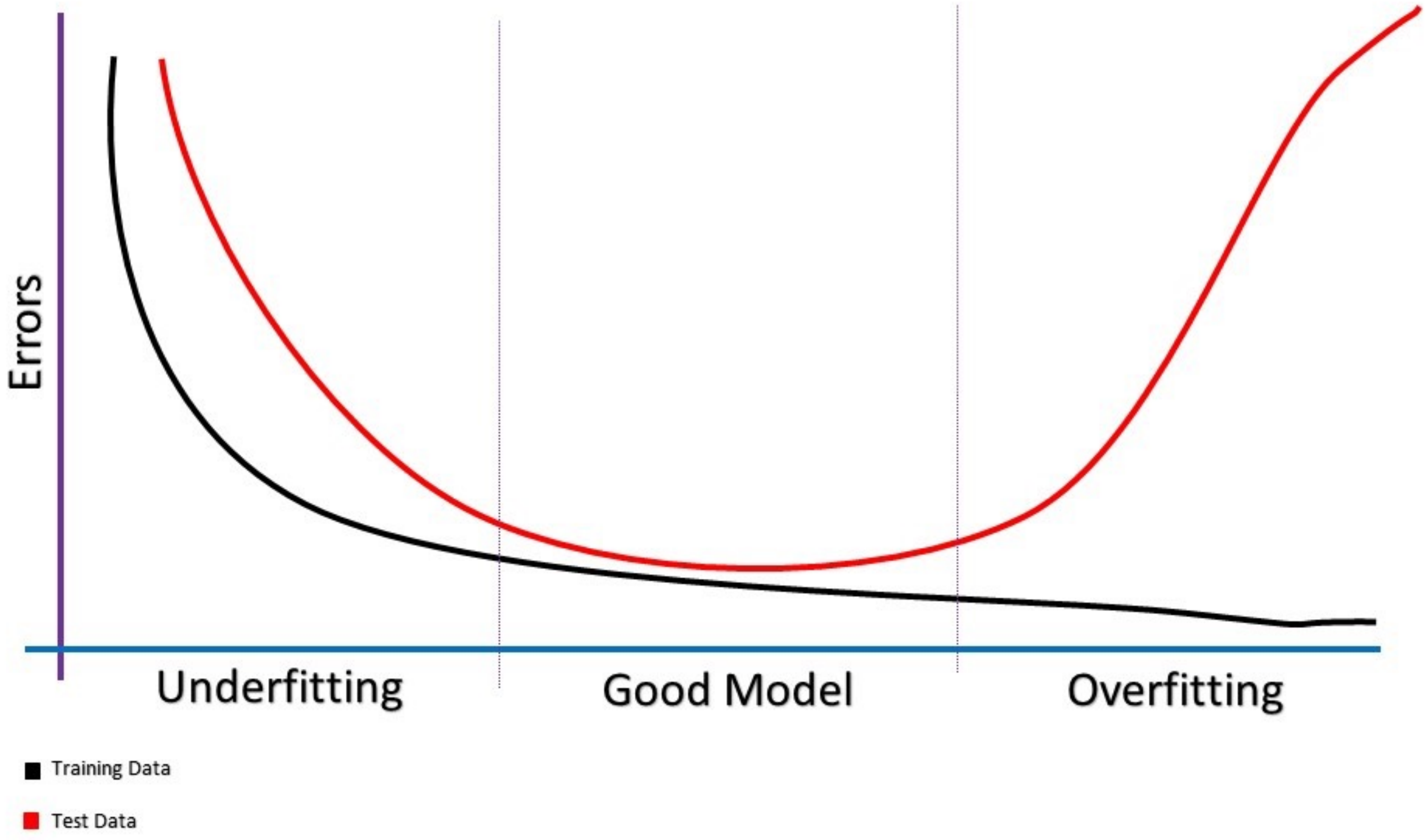
Overfitting

- Deep neural networks typically have tens of thousands of parameters, sometimes even millions
- With so many parameters, the network has an incredible amount of freedom and can fit a huge variety of complex datasets
- BUT this great flexibility also means that it is prone to overfitting the training set

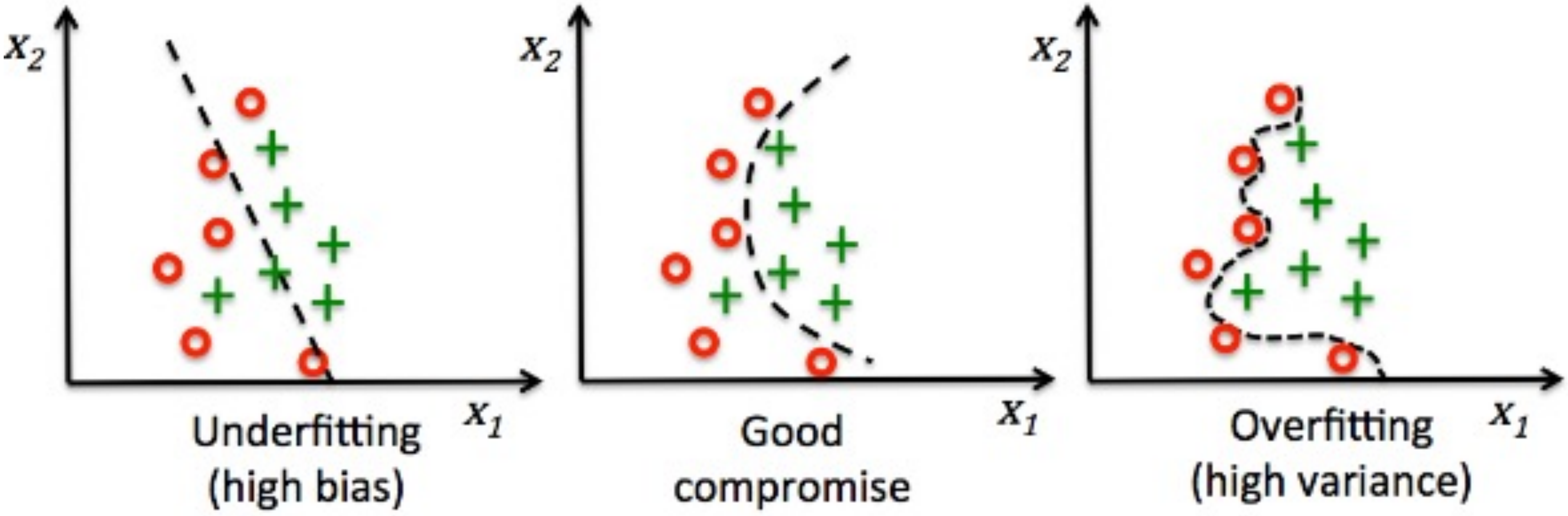
Overfitting
Case 1: test with training data

- Learning the parameters of a prediction function and testing it on the same data is a methodological mistake:
  - a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data.
- This situation is called **overfitting**.

Overfitting
Case 2: too complex model



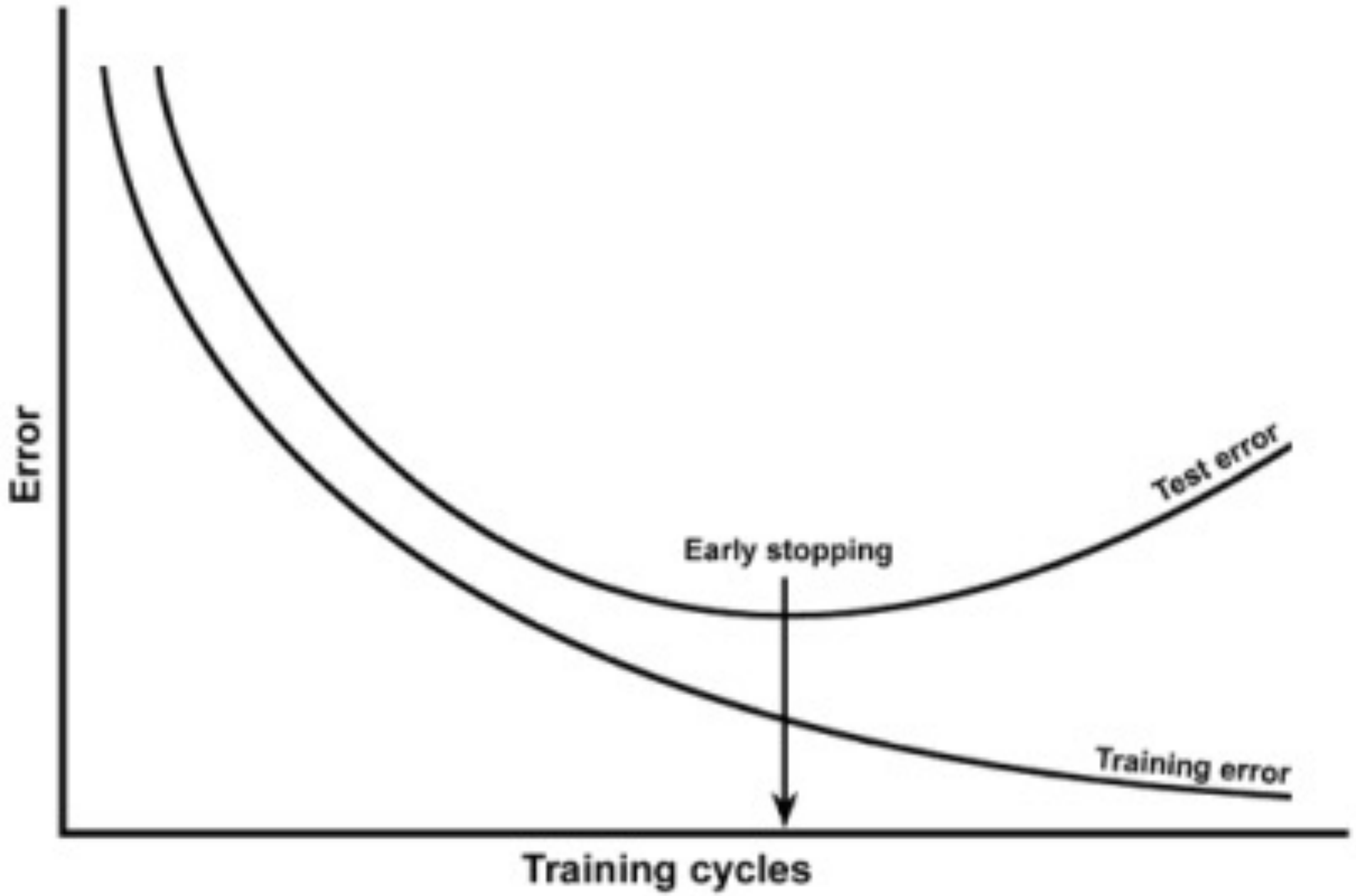
Bias vs Variance



Avoiding Overfitting

- Early stopping
  - just interrupt training when its performance on the validation set starts dropping
- L1 and L2 regularization
- Dropout

Early Stopping



Avoiding Overfitting

- L1 and L2 regularizations
  - Regularization makes slight modifications to the learning algorithm such that the model generalizes better
  - These update the general cost function by adding another term known as the regularization term
  - In L2 (weight decay),

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|^2$$

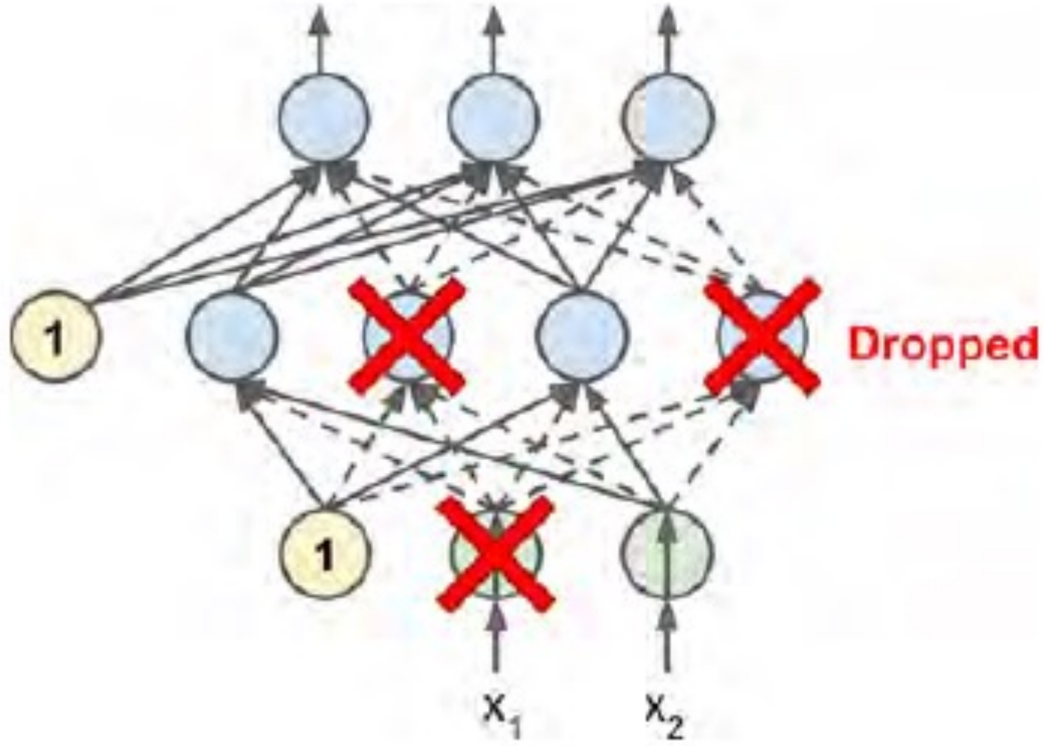
  - In L1 ,

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|$$

Avoiding Overfitting: Dropout

- Dropout
  - The most popular regularization technique for deep neural networks
  - A simple algorithm:
    - At every training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being temporarily "dropped out"
      - meaning it will be entirely ignored during this training step
      - but it may be active during the next step
  - The hyperparameter p is called the dropout rate, and it is typically set to 50%.
  - After training, neurons don't get dropped anymore

Avoiding Overfitting: Dropout



Dropout Regularization

Keras Model training

https://keras.io/api/models/model_training_apis/

compile

In [ ]:

```
Model.compile(
    optimizer="rmsprop",
    loss=None,
    metrics=None,
    loss_weights=None,
    weighted_metrics=None,
    run_eagerly=None,
    steps_per_execution=None,
    **kwargs
)
```

fit

In [ ]:

```
Model.fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose=1,
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
    class_weight=None,
    sample_weight=None,
    initial_epoch=0,
    steps_per_epoch=None,
    validation_steps=None,
    validation_batch_size=None,
    validation_freq=1,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
)
```