Introduction to Neural Networks

Kun Hu

April 13, 2020

Overview

Perceptron (Single Neuron) Flow Chat

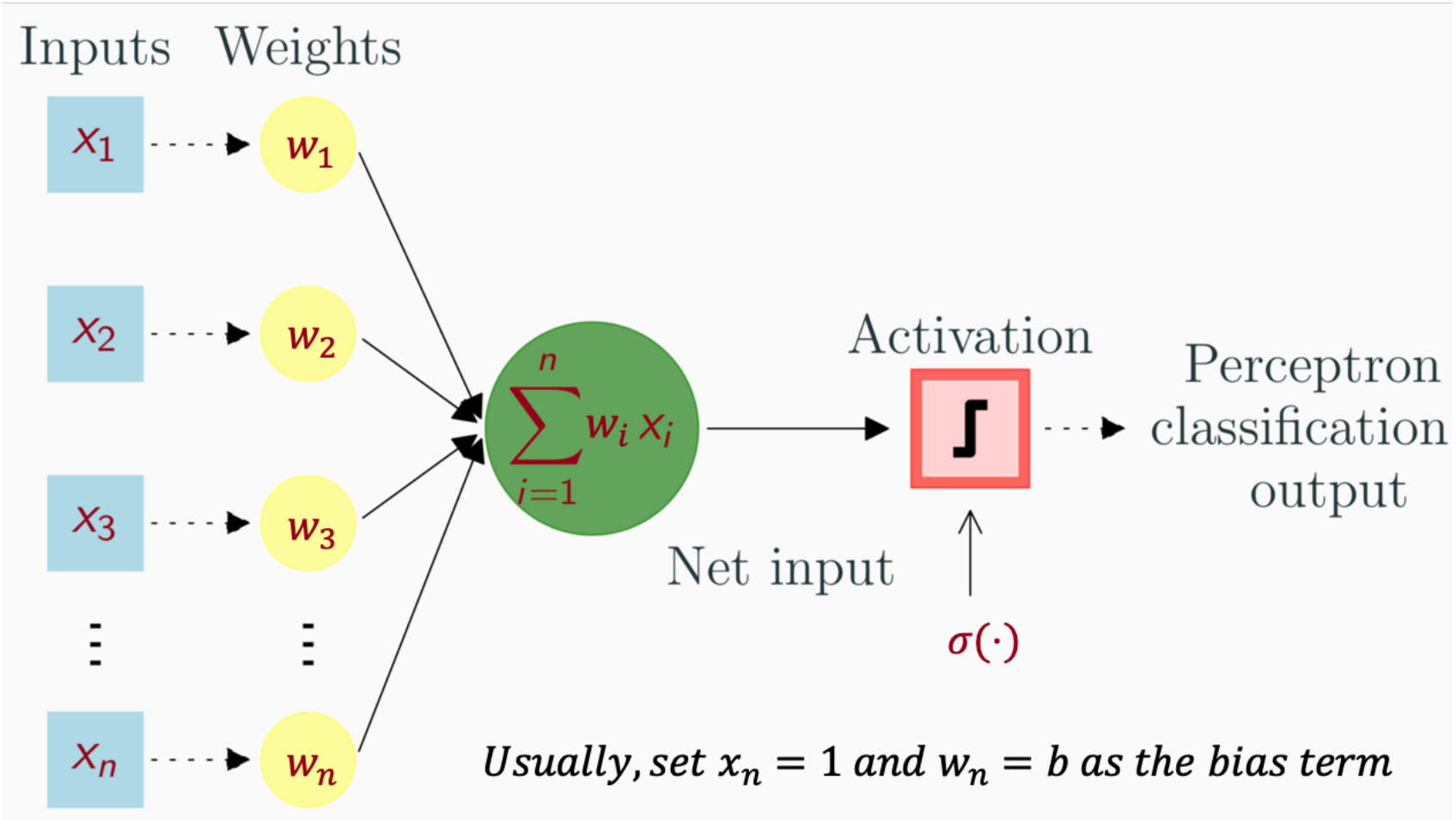
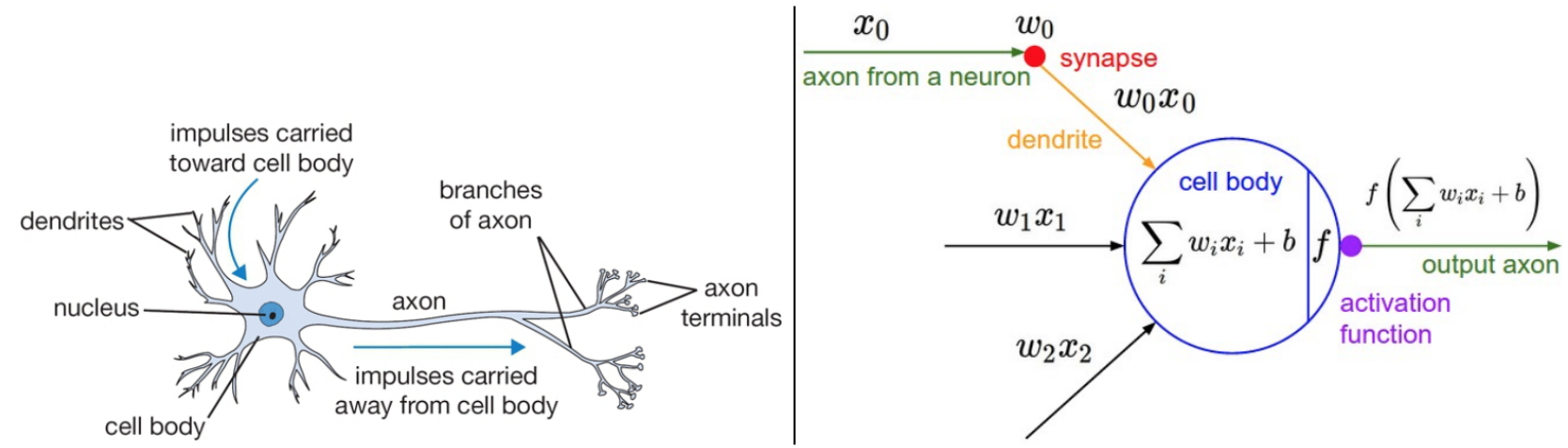


Figure: neuron: a mapping from $\mathbb{R}^N$ to $\mathbb{R}$

Source: Fernandez-Villaverde 2018: Lectures on Machine Learning for Economists I

Biological Analog



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

MultiLayer Perceptrons

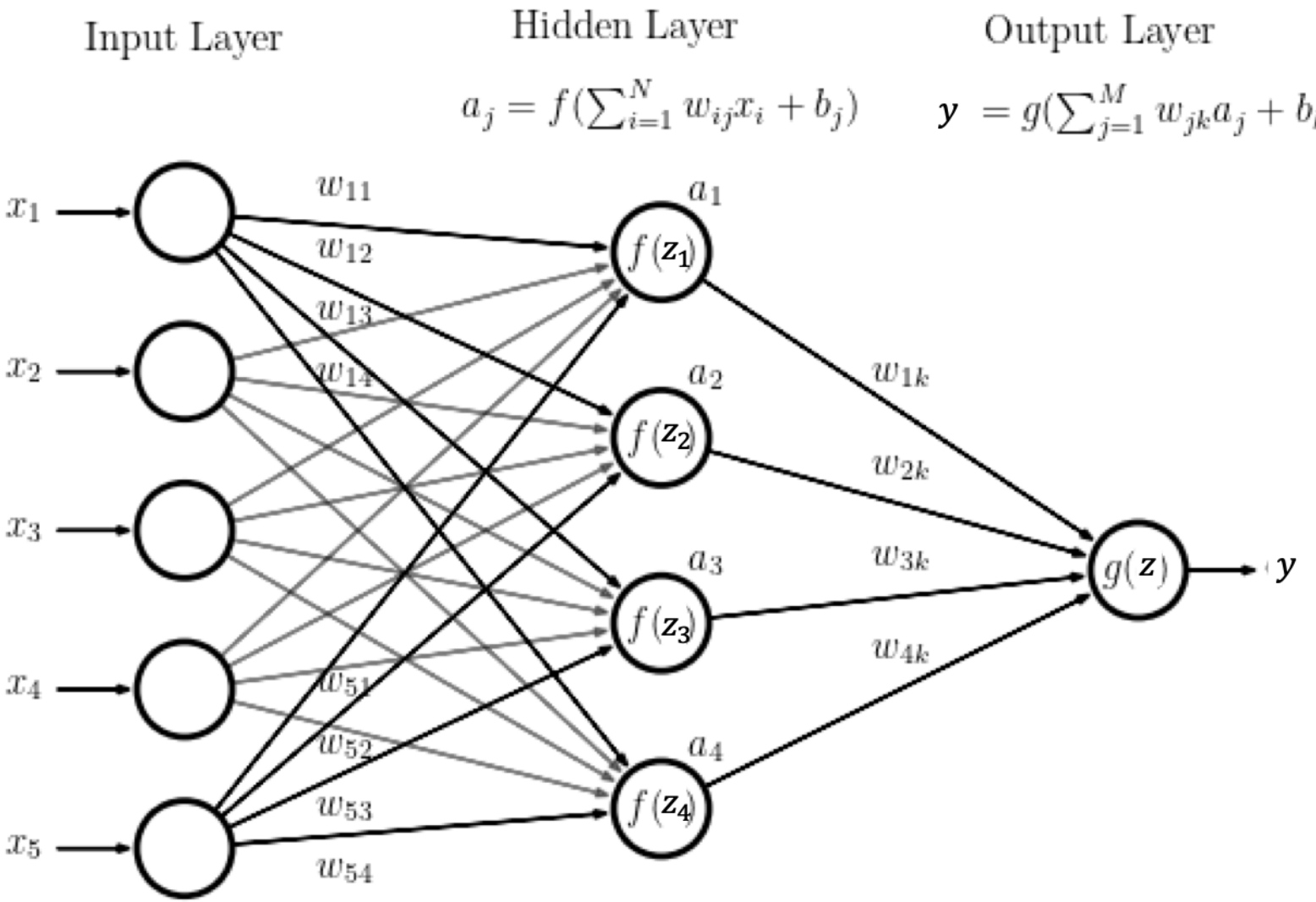
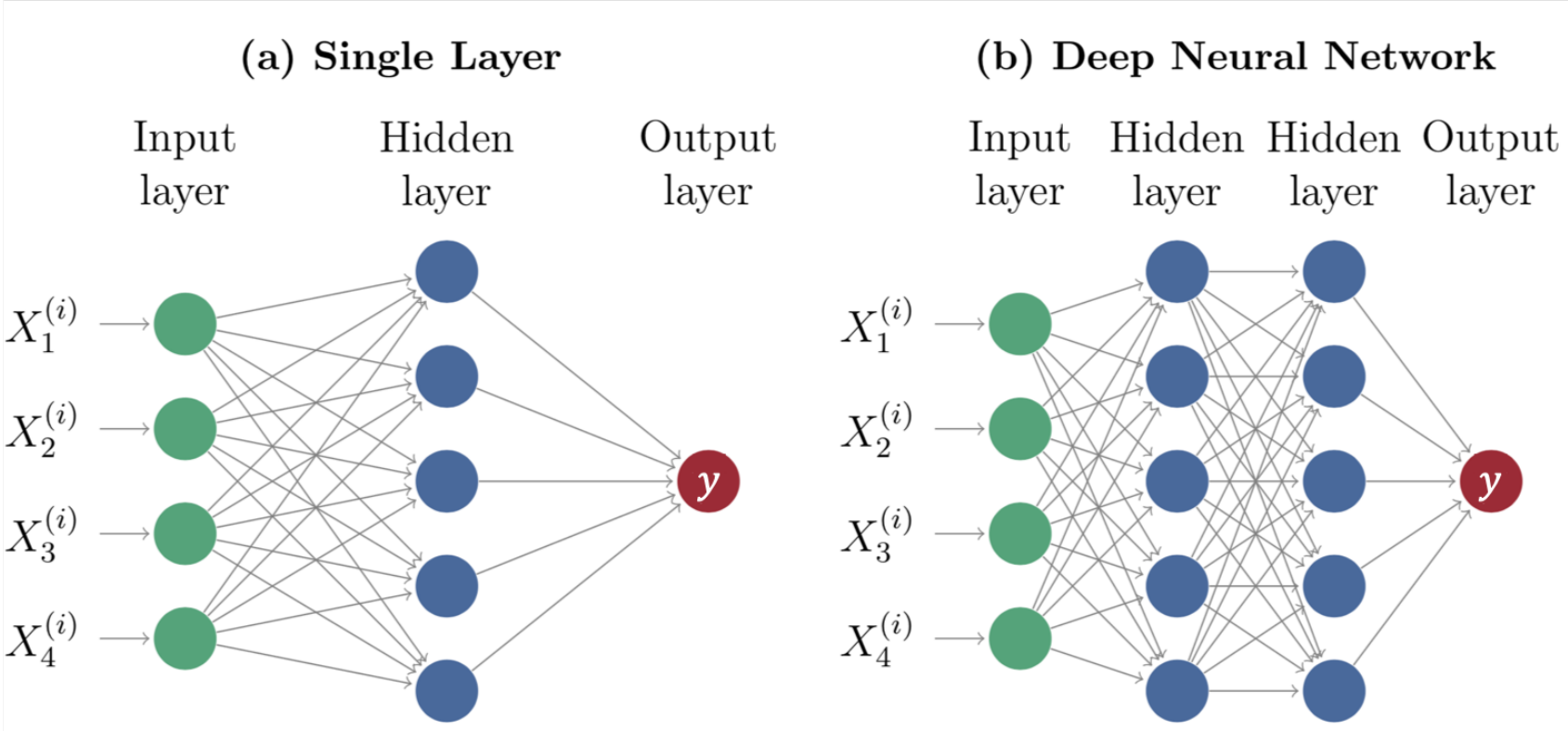


Figure: A MultiLayer Perceptrons with One Hidden Layer

(Deep) Feedforward Neural Network



- i is observation i's index; each one has 4 features (green circle)
- blue circle: linear and non-linear transformation ($\phi(z)$)
- red circle: linear combination to produce the final output

Source: Duarte 2018

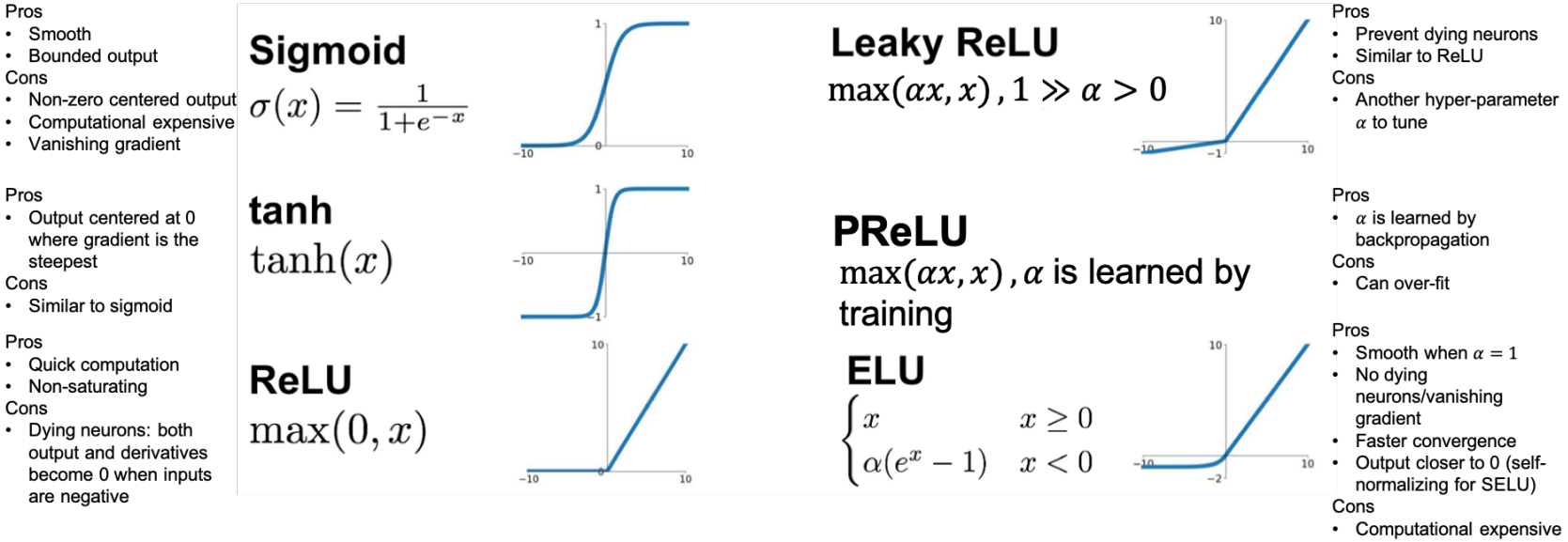
Activation Function

Transformation of the linear combination of inputs

$$\mathbf{a}^l = \phi(\mathbf{z}^l) = \phi(\mathbf{b}^l + \mathbf{w}^l \cdot \mathbf{a}^{l-1})$$

- $l = 1, ..., L$ is the layer index, with $l = L$ as the output layer
- $\mathbf{b}$ is sometimes referred as "bias"
- Without nonlinear $\phi(z)$, y is just a linear combination of $\mathbf{x}$

Activation Functions



- In general, SELU $>$ ELU $>$ leaky ReLU (and its variants) $>$ ReLU $>$ tanh $>$ sigmoid
- For network structure that prevents self-normalizing, ELU can outperform SELU
- Actual performance varies by problem. Better to trail-and-error.

Loss (Cost) Functions

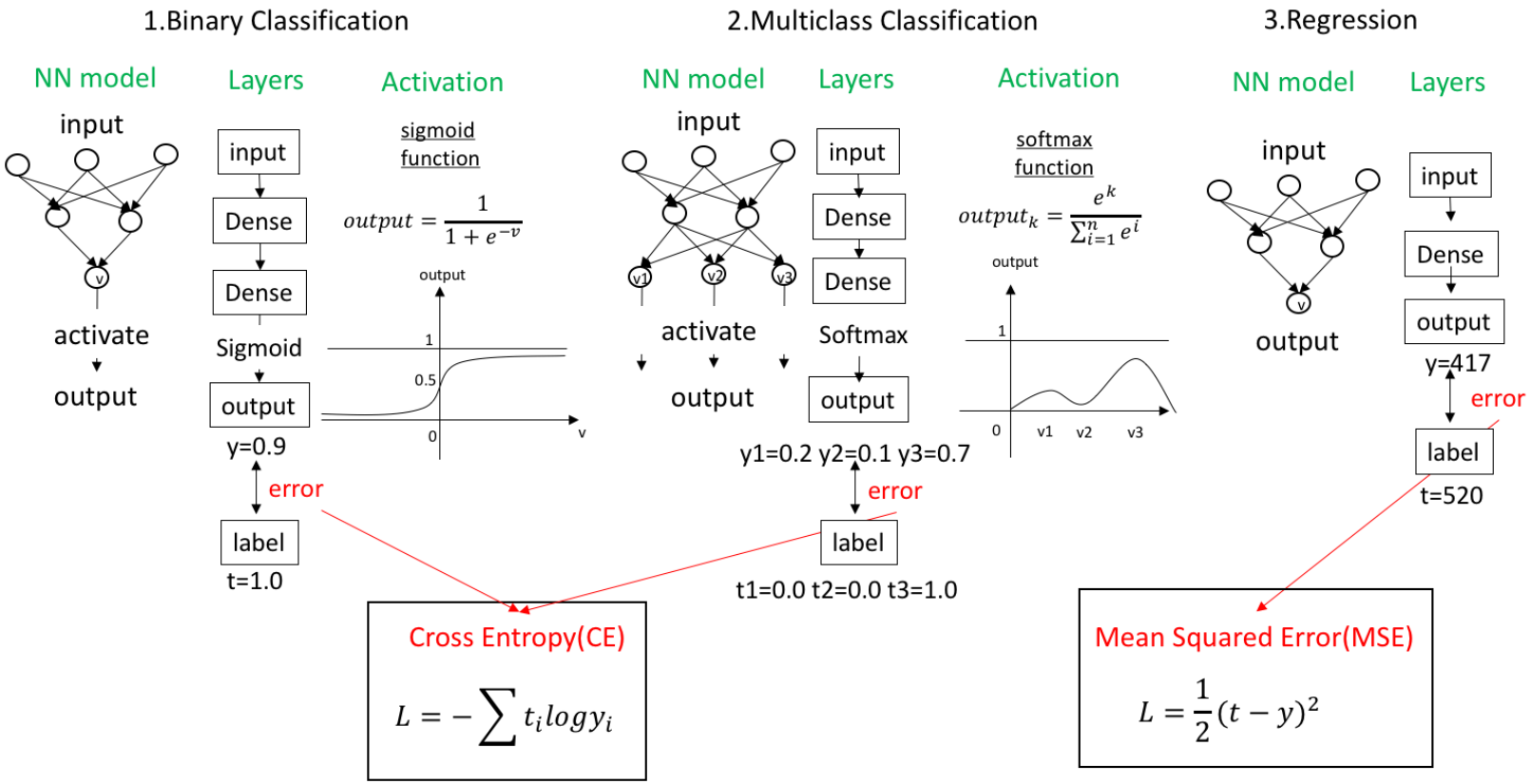


Figure: Common Loss Functions

Source: ReNom

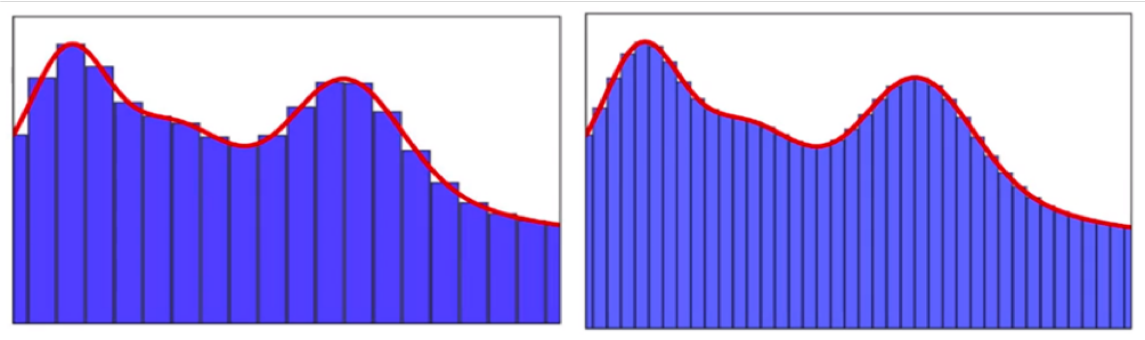
Universal Approximation Theorem

Hornik, K., Stinchcombe, M., and White, H. (1989)

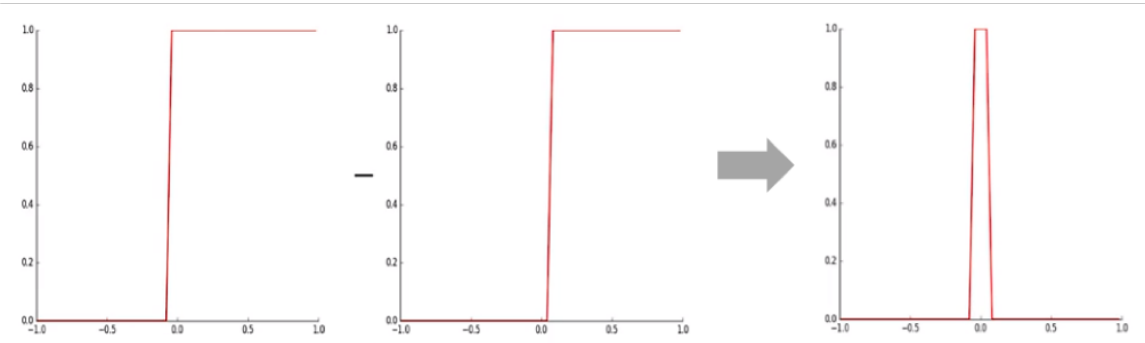
One-hidden layer feed-forward network with arbitrary squashing functions can approximate any Borel measurable function.

- It requires a sufficient amount of neurons (unbounded width)
- Extension to bounded width and Lebesgue-measurable functions: Lu. et al 2017, Hanin and Sellke 2017

Universal Approximation Theorem (Proof by Graphs*)



1. Decompose a function into a series of rectangles (bars)



2. Each rectangle can be approximated by the difference between two sigmoid functions

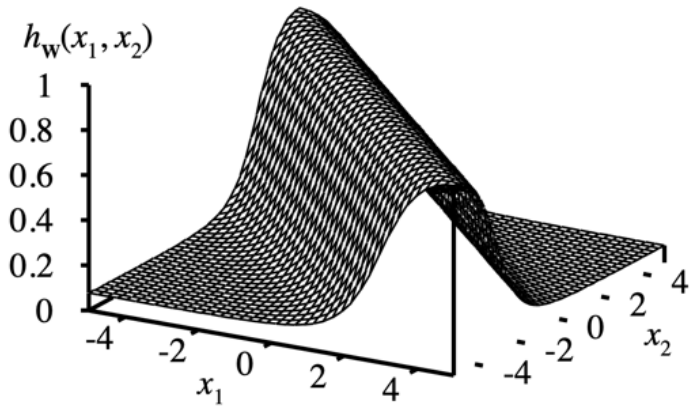
A. Take a very steep sigmoid function

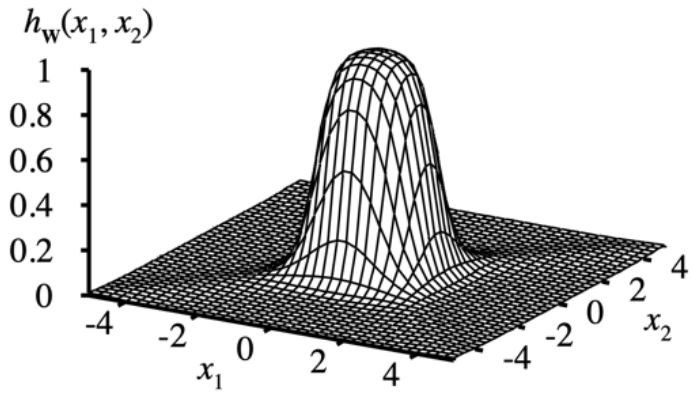
B. Copy the function but shift rightward a bit

C. Subtract B from A to get a rectangle

Source: NKumar

Universal Approximation Theorem (Proof by Graphs*)





Combine two opposite-facing threshold functions to make a ridge

Combine two perpendicular ridges to make a bump

Add bumps of various sizes and locations to fit any surface

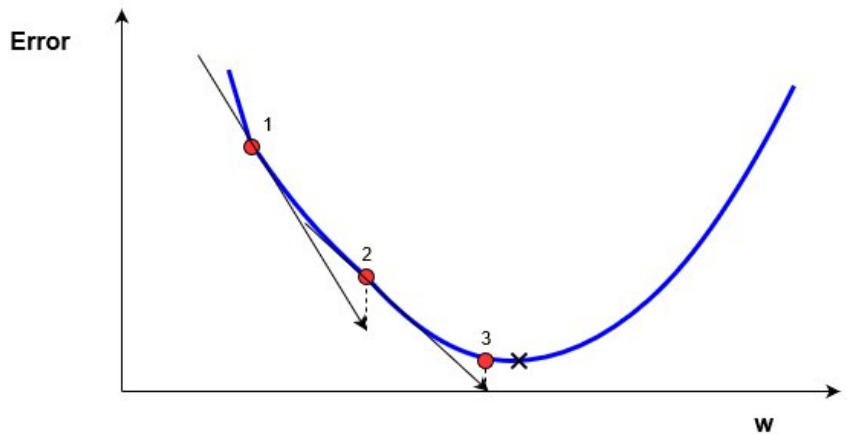
Source: AIMA

Finding the NN Parameters $\mathbf{\Theta} = \{\mathbf{w}^l, \mathbf{b}^l\}_{l=1,\ldots,L}$

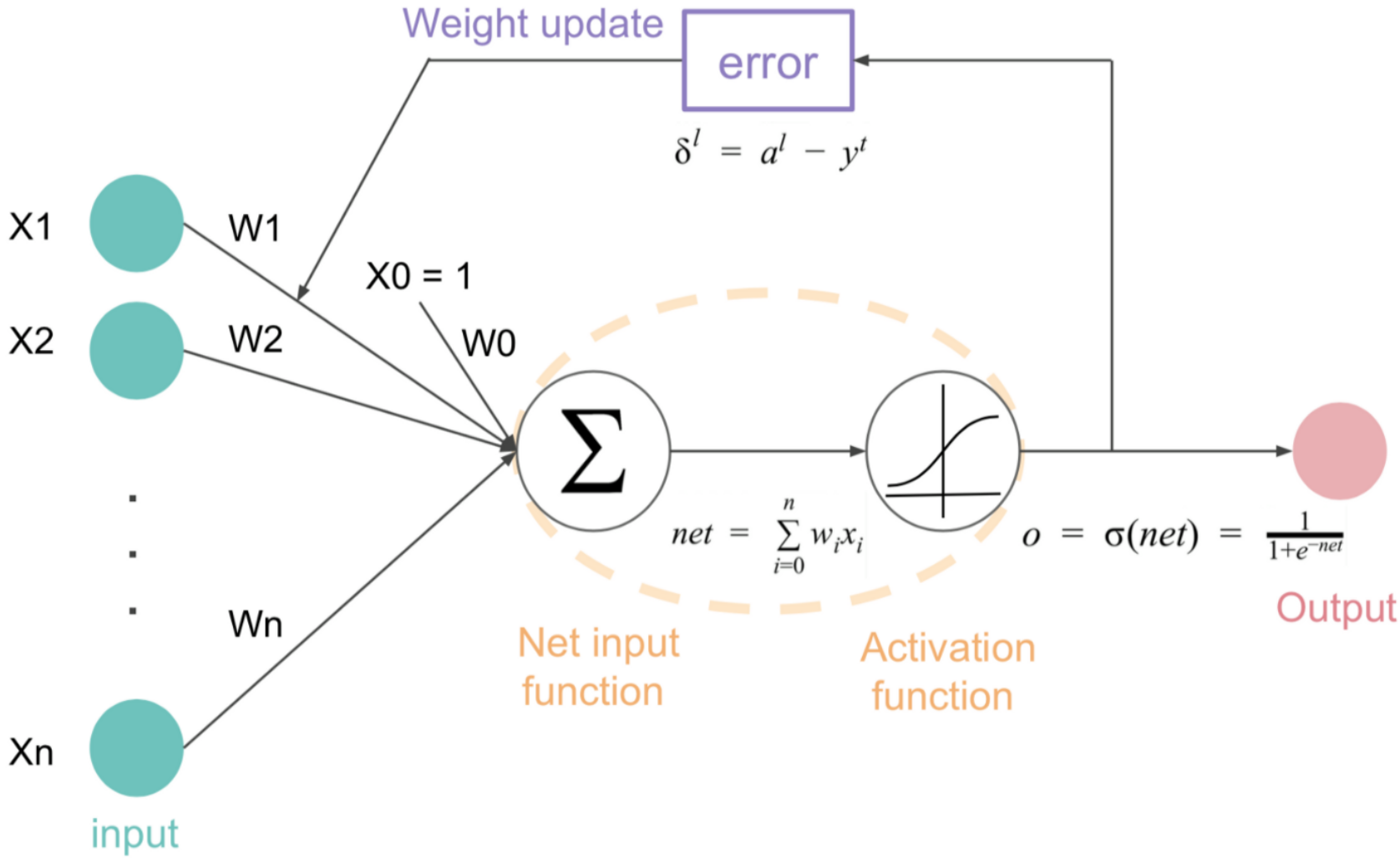
$$\mathbf{\Theta}^* = \arg\min_{\mathbf{\Theta}} \sum_{i=1}^{N} \mathbf{C}(\mathbf{Y}_i, \mathbf{X}_i; \mathbf{\Theta})$$

- $\mathbf{X}_i$ is the input vector for observation i, $\mathbf{Y}_i$ is the target
- $\mathbf{C}(\cdot)$ is the loss function
- $\mathbf{\Theta}$ is updated according to (ϵ is the learning rate)

$$\mathbf{\Theta}' = \mathbf{\Theta} - \epsilon \nabla_{\mathbf{\Theta}} \sum_{i=1}^{1 \leq J \leq N} \mathbf{C}(\mathbf{Y}_i, \mathbf{X}_i; \mathbf{\Theta})$$



How to Calculate Gradient: Backpropagation (Chain Rule)



Source: Purit Punyawiwat

How to Calculate Gradient: Backpropagation (Chain Rule)

A one-unit per-layer network without bias term.

Cost function $C = \frac{1}{2}(a^2 - y)^2$

Forward propagation to calculate output

$$x \to \sigma(w^1 x = z^1) \to a^1 \to \sigma(w^2 a^1 = z^2) \to a^2 \to \frac{1}{2}(a^2 - y)^2 = C$$

Backward propagation to calculate derivative

$$\delta^2 \equiv \frac{\partial C}{\partial z^2} = \nabla C \cdot \sigma'(z^2) = \frac{\partial C}{\partial a^2}\frac{\partial a^2}{\partial z^2} = (a^2 - y)\sigma'(z^2)$$

$$\delta^1 \equiv \frac{\partial C}{\partial z^1} = \frac{\partial C}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1} = \delta^2 w^2 \sigma'(z^1)$$

$$\frac{\partial C}{\partial w^2} = \frac{\partial C}{\partial z^2}\frac{\partial z^2}{\partial w^2} = \delta^2 a^1$$

$$\frac{\partial C}{\partial w^1} = \frac{\partial C}{\partial z^1}\frac{\partial z^1}{\partial w^1} = \delta^1 x$$

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad \text{(BP2)}$$

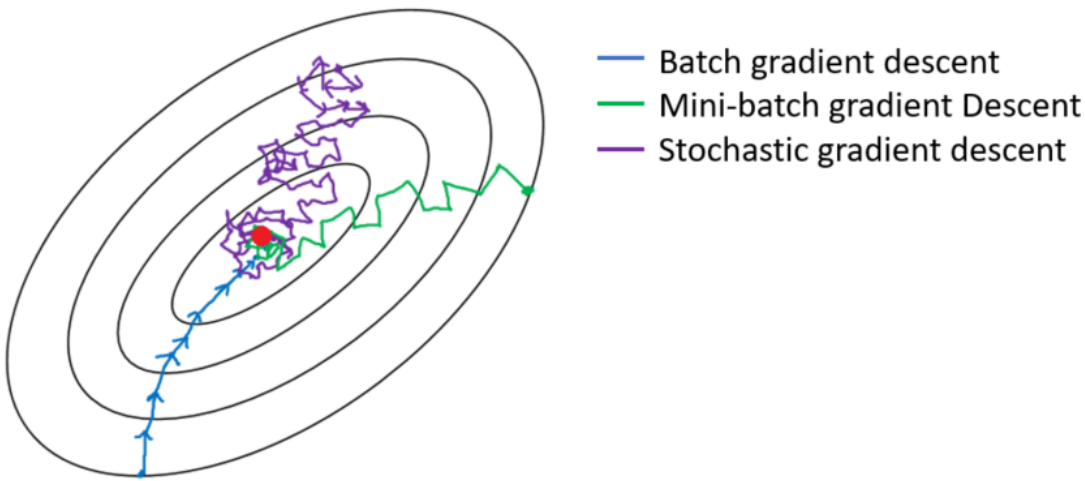
$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \quad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \quad \text{(BP4)}$$

Source: Neural Networks and Deep Learning

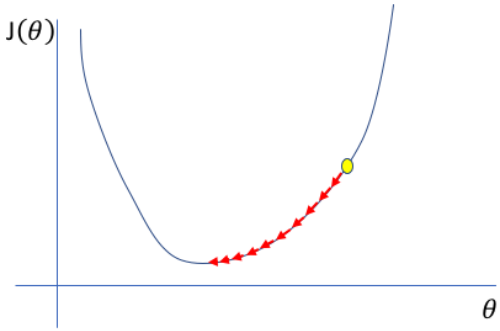
Gradient Descent (GD)

- Batch GD $J = N$ (all observations): high computation cost when N is large; can be trapped in local minimum
- Stochastic GD $J = 1$ (randomly choose one obs): noisy and continue to wander around minimum (slow convergence)
- Mini-batch GD $J \approx [2^5, 2^9]$ (batch sampling): most popular



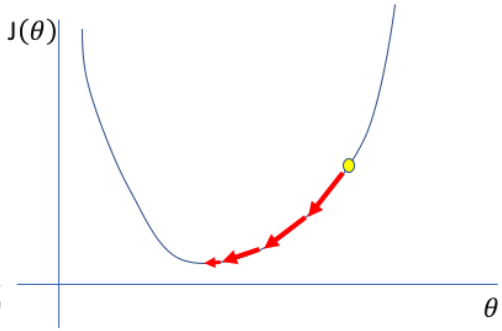
Source and reference: Adventures in Machine Learning; Andrew Ng; Imad Dabbura

Too low



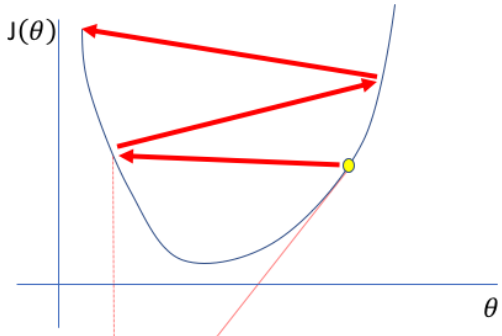
A small learning rate requires many updates before reaching the minimum point

Just right

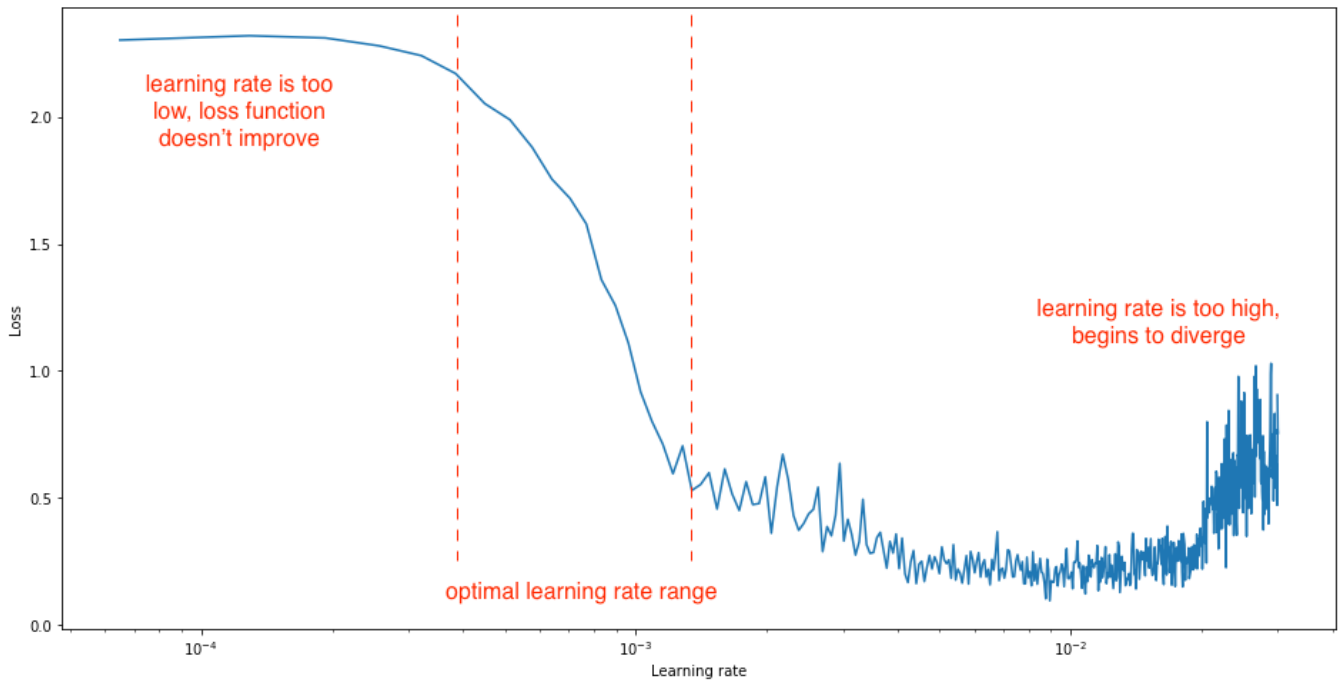


The optimal learning rate swiftly reaches the minimum point

Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors



Gradient Descent with Momentum

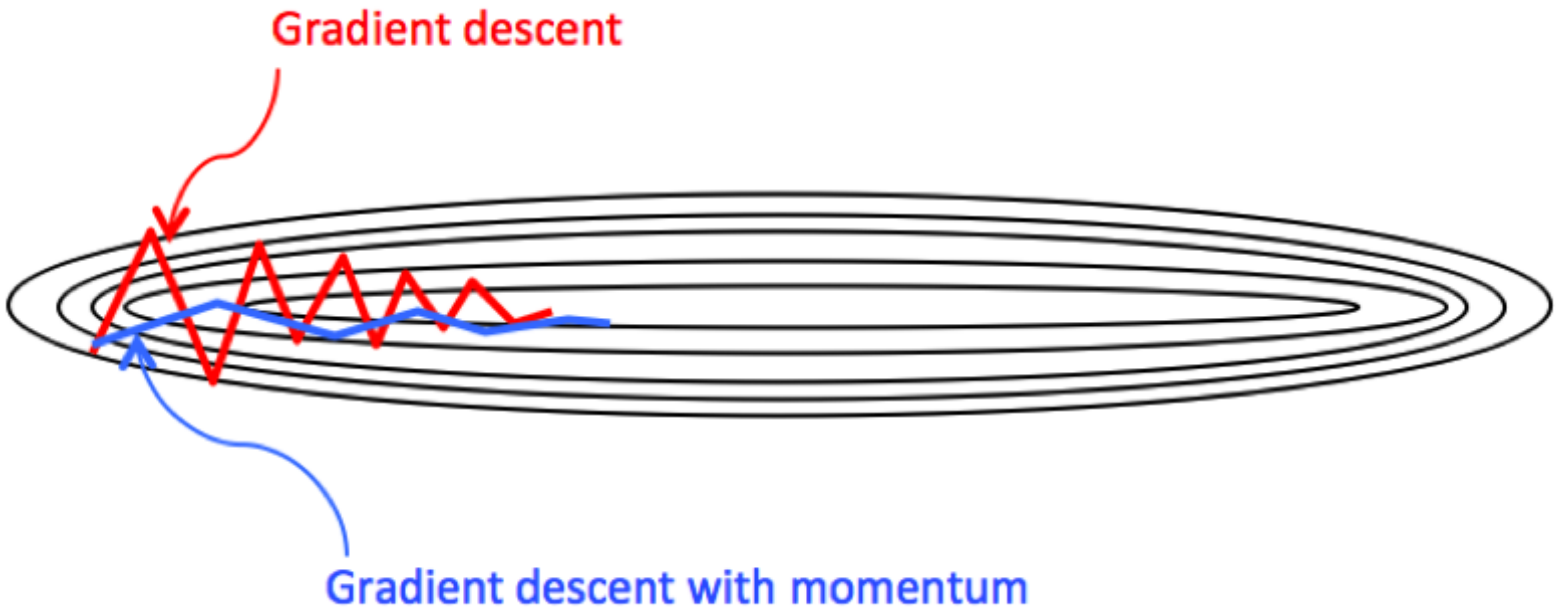


Figure: Update by the exponential moving average of gradients

- $\mathbf{m} \leftarrow \beta \mathbf{m} + (1 - \beta) \nabla_{\mathbf{\Theta}} C$
- $\mathbf{\Theta} \leftarrow \mathbf{\Theta} - \epsilon \mathbf{m}$
- Cancel out the gradient in directions where it oscillates; speed up in directions where it accumulates

Gradient Descent with RMSProp

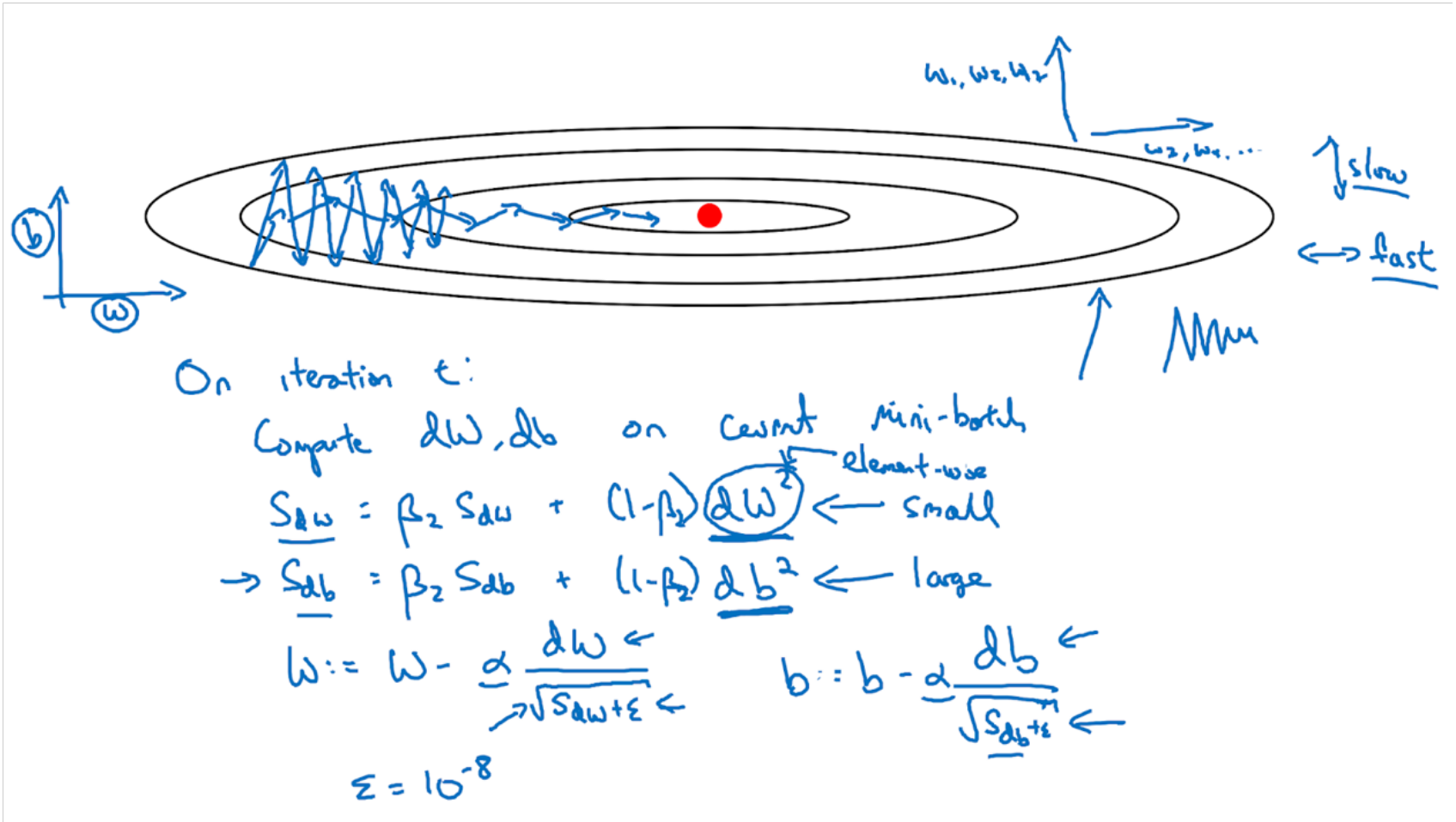


Figure: Adaptive Learning Rate

Other Considerations

- Weight Initialization (LeCun, Glorot/Xavier, He Initialization)
- Optimization: Adagrad, RMSprop, Adam, Nesterov Accelerated Gradient, Nadam ...
- Batch normalization (initializing inputs with zero mean and unit variance)
- Gradient Clipping
- Learning rate scheduling
- Regularization: l_1 and l_2 Norm, Dropout/Monte Carlo Dropout, Early Stopping
- Bagged and Boosted NN, Bayesian NN

See HOML Chapter 11 for reference.

Outline

Build and Train A Neural Network From Scratch (link)

```python
# Transfer neuron activation
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
    return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network)-1:
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += (neuron['weights'][j] * neuron['delta'])
                errors.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(expected[j] - neuron['output'])
        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])

# Update network weights with error
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] += l_rate * neuron['delta']

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
    for epoch in range(n_epoch):
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)
```

Machine Learning Packages

Keras

- High-level API Standard
- Very easy to learn

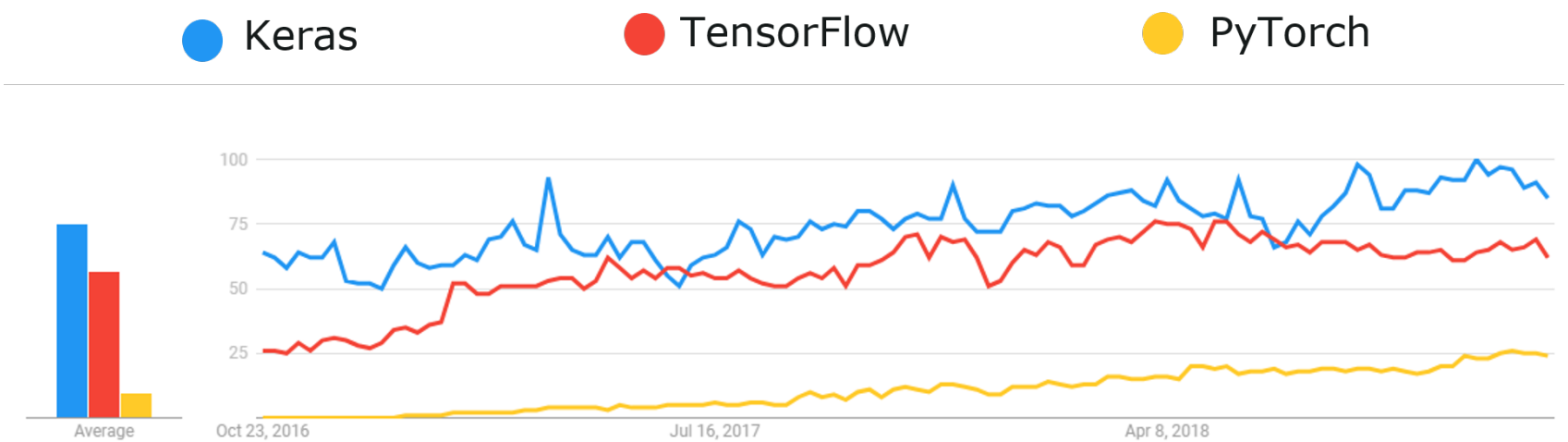
TensorFlow

- Use Keras as its core API
- Easy to deploy into production

PyTorch

- More Pythonic
- Popular in research community

Popularity of Keras, TensorFlow and PyTorch



Source: edureka!

Keras is very easy to use!

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout

# Generate dummy data
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
y_test = np.random.randint(2, size=(100, 1))

model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

Figure: MLP for binary classification

If NN is so easy and so powerful, then ...

- Why don't we see it used everywhere?
- Why is gradient boosting more often the winner of Kaggle contests?

Neural Network Is Not An "Off-the-Shelf" Procedure

- Need to encode (or embed) categorical variables
- Not scale invariant and sensitive to outliers
- Too many hyper-parameters to tune (architecture, optimizer, initialization, regularization ...).
- Need domain specific priors (and many years of research) to design a good network

Therefore, for tasks with short timelines and no prior filed knowledge (like Kaggle), gradient boosting (and other tree-based methods) usually wins.

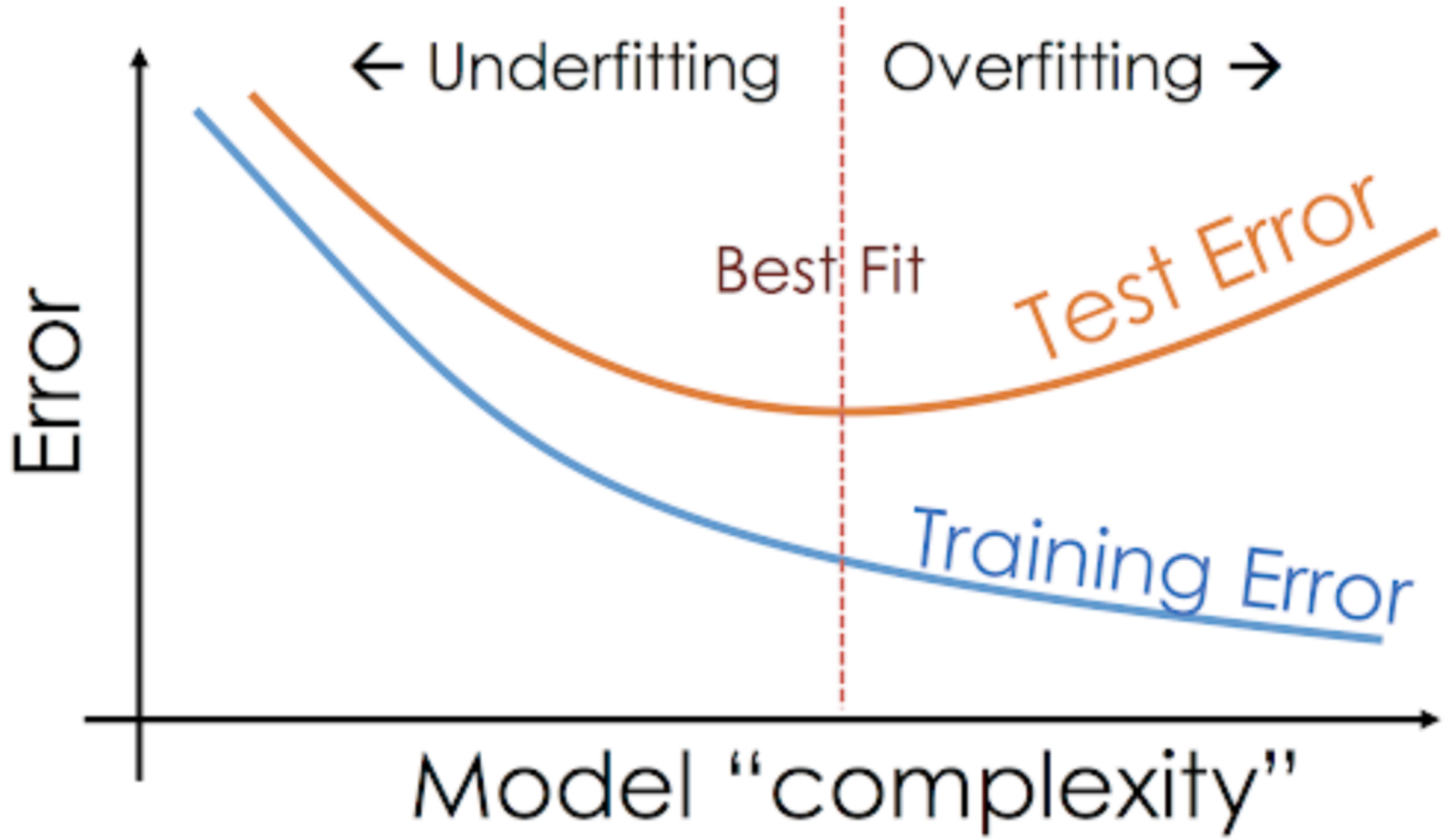
However, for a tasks with

- very high complexity and sufficient data (computer vision, NLP, reinforcement learning ...)
- the right architecture (convolution NN, recurrent NN ...)

NN (deep learning) blows any other method out of the water


CRUISE
LDG GEAR
SPEED BRAKE
FLAPS
END MASTER
END START

How to Tune Hyper-parameters



Minimize Test Error Given Data and Resources

$$\text{Test Error} = \text{Training Error} + \text{Generalization Gap}$$

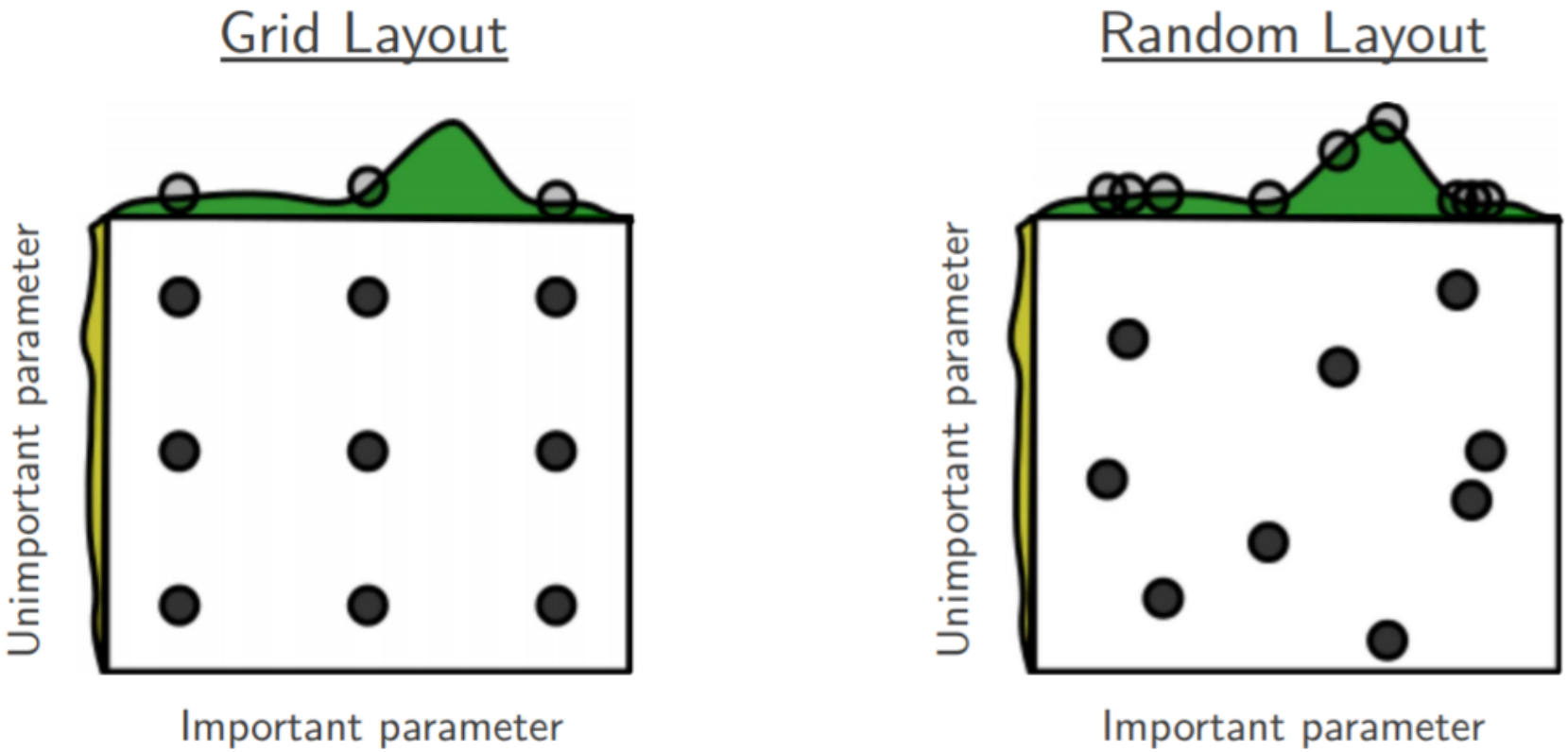
Stretch-pants and Orthogonalization

- "Stretch-pants": Increase model capacity to match the complexity of the task. First try to decrease training error and then use regularization to decrease generalization gap
- Orthogonal: focus on one thing at a time – training error or generalization

Some Best Practices in Hyper-parameter Tuning

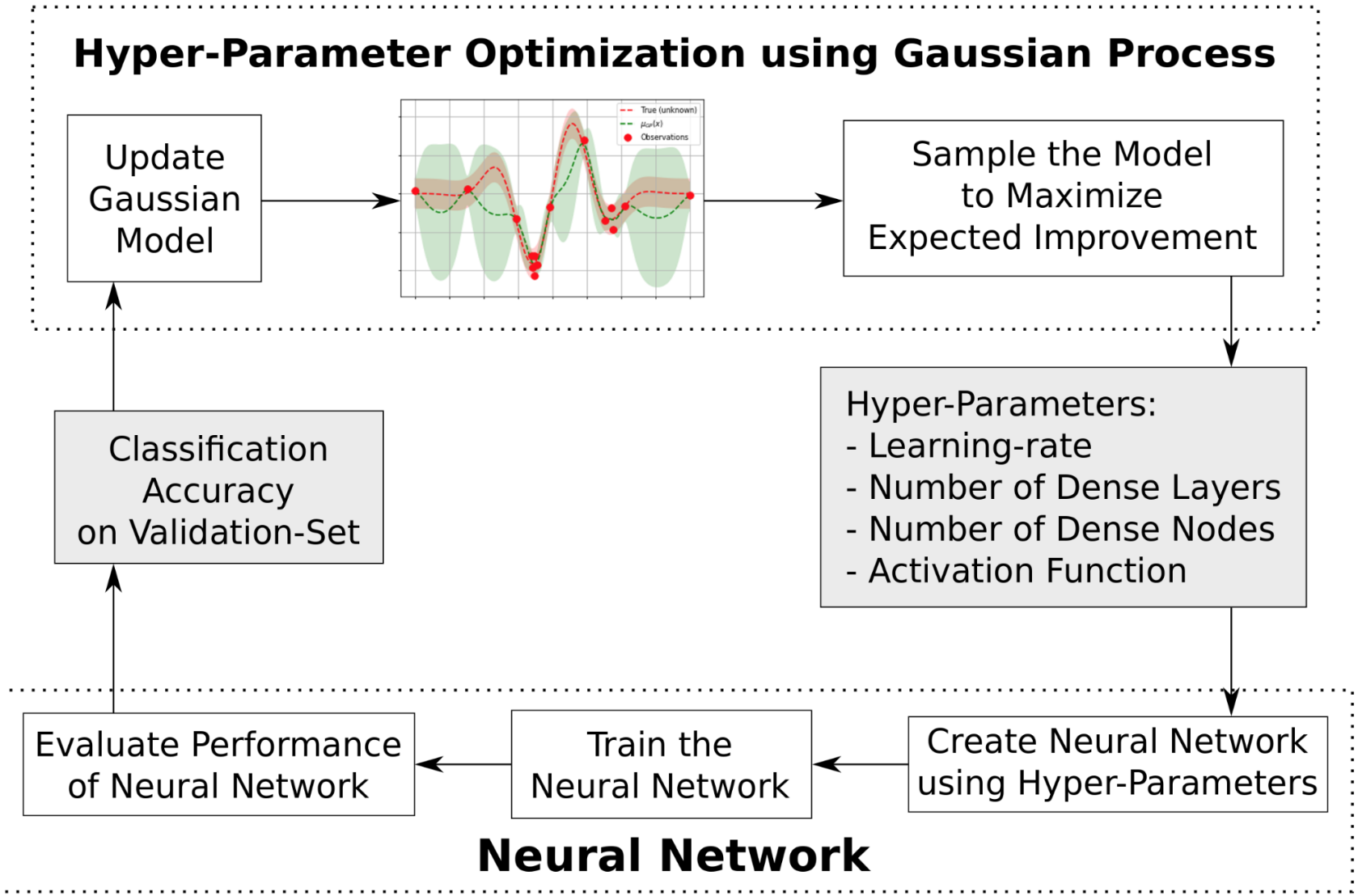
- Always starts with EDA (how many types of data, missing data, irrelevant data ...)
- Debug using a bare-bone model (Is the cost function defined correctly? Does the input tensor have the right dimension? Does the prediction make sense?)
- Trial-and-error and get your hands dirty. (Don't tune it yet; try some configuration and see if you have any insights)
- Define a default baseline model (It doesn't have to be neural network! Try boosting!)
- Can you get more data? (Plot the error against training size on a log scale)

Hands-off Automatic Tuning: Grid and Random Search



- Most of the time, random search is better if you have more than one hyper-parameter to tune.

Hands-off Automatic Tuning: Bayesian Optimization



- This can be very time consuming, and not necessarily better.