Using Evolutionary Algorithms for Machine Learning Calibration

Nuno Antunes Ribeiro

Assistant Professor

Engineering Systems and Design

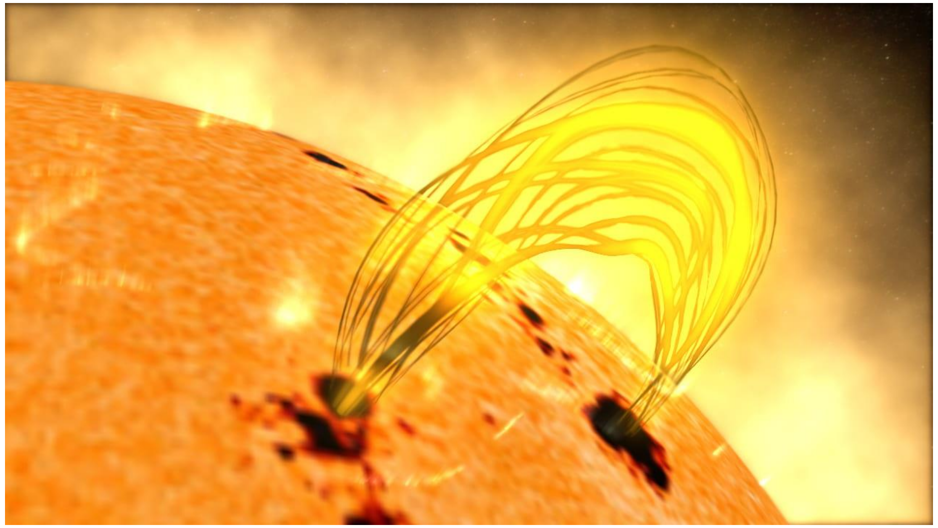
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

EA for Machine Learning Calibration

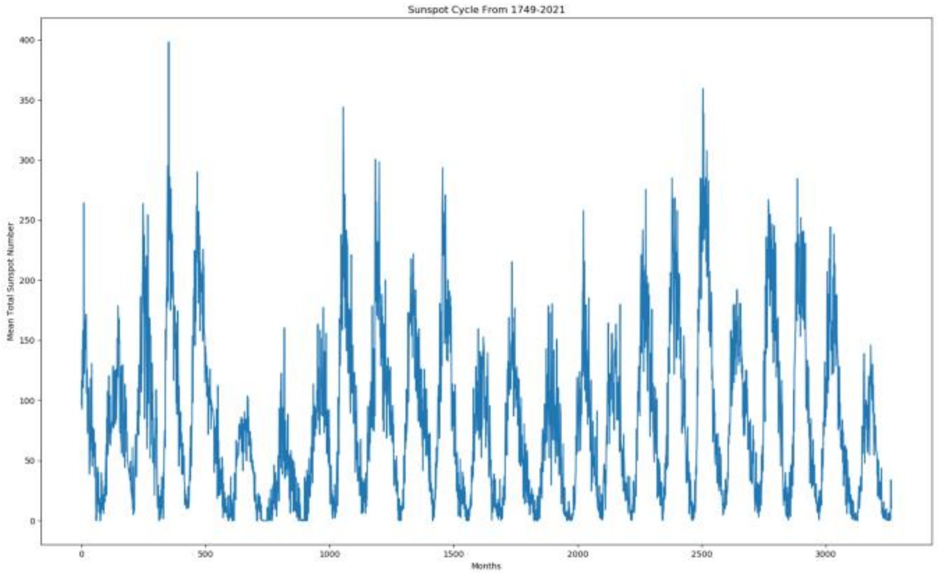
- What needs to be calibrated in a Machine Learning Model?
  - Feature selection
  - Hyperparameter tuning
  - Type of model to apply (e.g. Linear Regression, or Random Forest, or Neural Networks, etc.)
  - ML architecture (e.g. Neural Networks)

- Machine Learning Calibration can be seen as an optimization problem whose goal is to **maximize predictive accuracy**

Today's Class: Predicting Sunspot Cycle



- **Sunspots** have been observed for over four centuries, constituting the longest running, continuous time series of any natural phenomena in the Universe.
- Sunspots spawn severe space weather characterized by **solar flares**, coronal mass ejections, geomagnetic storms, enhanced radiative, and energetic particle flux
- Endangering **satellites**, global **communication systems**, **air-traffic** overpolar routes, and **electric power grids**
- Protection of planetary technologies and space situational awareness is therefore enabled by solar activity predictions.



Sunspots.csv

Example from: https://towardsdatascience.com/unit-3-application-evolving-neural-network-for-time-series-analysis-63c057cb1595

Neural Networks

Nuno Antunes Ribeiro

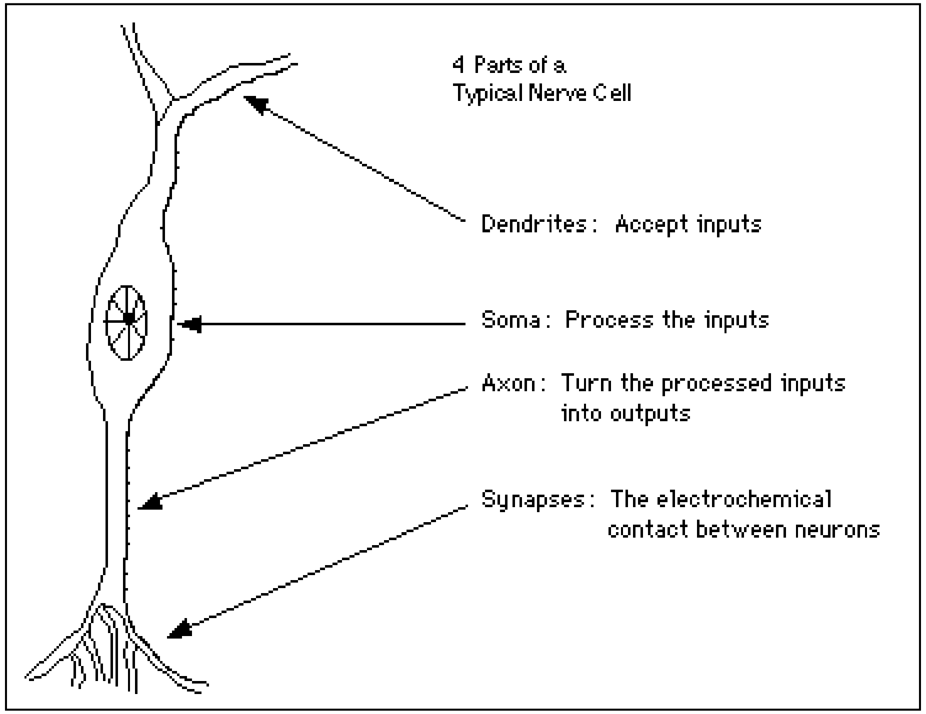
Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

Neural Networks

- We will be attempting to solve this problem through **Neural Networks**.
- Artificial neural networks were derived with biological inspiration from biological neural networks, the basic unit of the brain.
- **Neural Network learns by adjusting the weights** so as to be able to correctly classify the training data
- We will use **Genetic algorithms to calibrate the weights** of the Neural Network



- Fundamental processing element of a neural network is a neuron
- A human brain has 100 billion neurons
- An ant brain has 250,000 neurons

One Neuron as a Network (Perceptron)

- **Classification Problem**:
  - **x1** and **x2** are inputs of the data.
  - **y** is the output of the neuron , i.e the class label.
  - **x1** and **x2** values multiplied by weight values **w1** and **w2** are input to the **neuron x**.
  - Value of **x1** is multiplied by a weight **w1** and values of **x2** is multiplied by a weight **w2**.

Given that

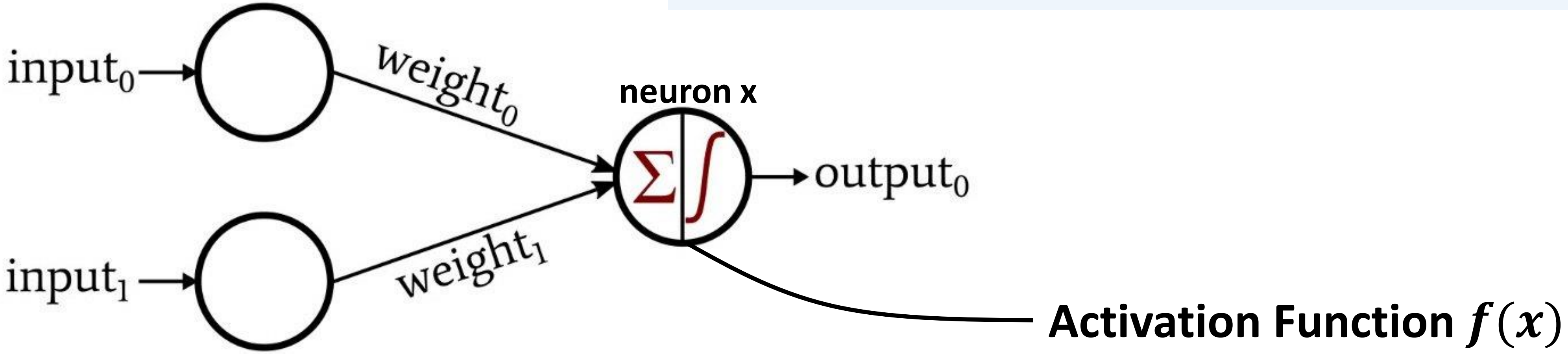
w1 = 0.5 and **w2** = 0.5

Say value of **x1** is 0.3 and value of **x2** is 0.8,

So, weighted sum is :

$$z = \boldsymbol{w1} \times \boldsymbol{x1} + \boldsymbol{w2} \times \boldsymbol{x2} = 0.5 \times 0.3 + 0.5 \times 0.8 = \mathbf{0.55}$$

Perceptron



One Neuron as a Network (Perceptron)

- The neuron receives the weighted sum as input and calculates the output as a function of input as follows :
- $y = f(x)$, where $f(x)$ is defined as
  - $f(x)$ = 0 { when *z* < 0.5 }
  - $f(x)$ = 1 { when *z* >= 0.5 }

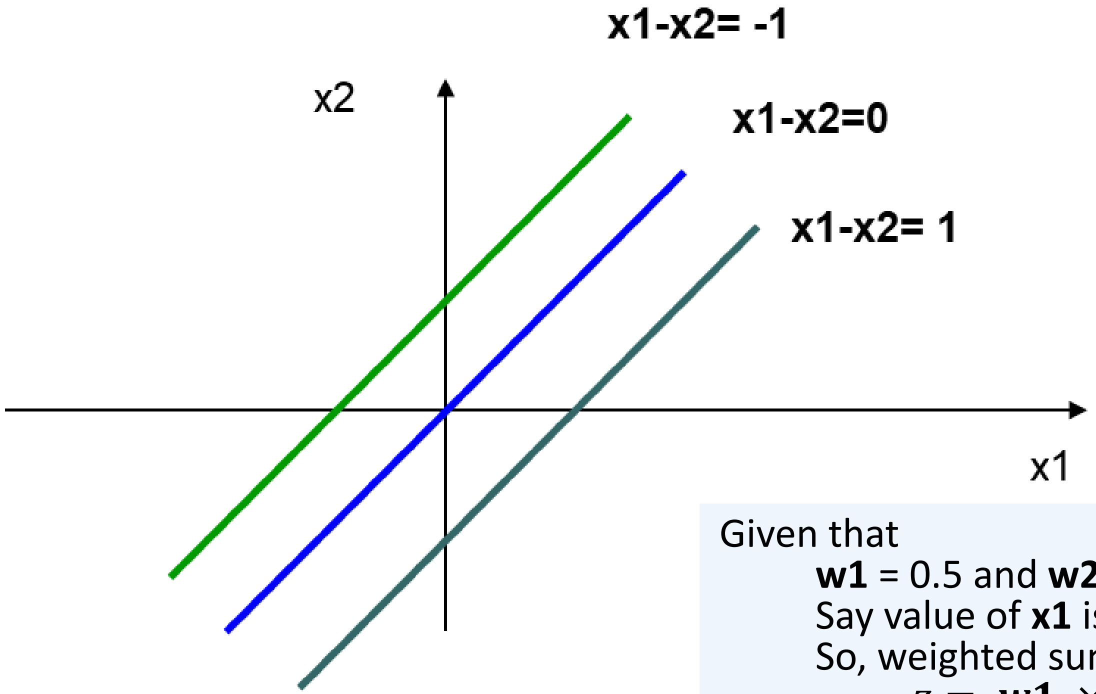
Activation Function

- For our example, *v* (weighted sum) is 0.55, so y = 1 ,
- That means corresponding input attribute values are classified in class 1.

- If for another input values , x = 0.45 , then y = 0,
- We could conclude that input values are classified to class 0.

Bias of a Neuron

- We need the **bias value** to be added to the weighted sum $\sum w_i x_i$ so that we can transform it from the origin.
  - $z = \sum w_i x_i + \boldsymbol{b}$, here b is the bias

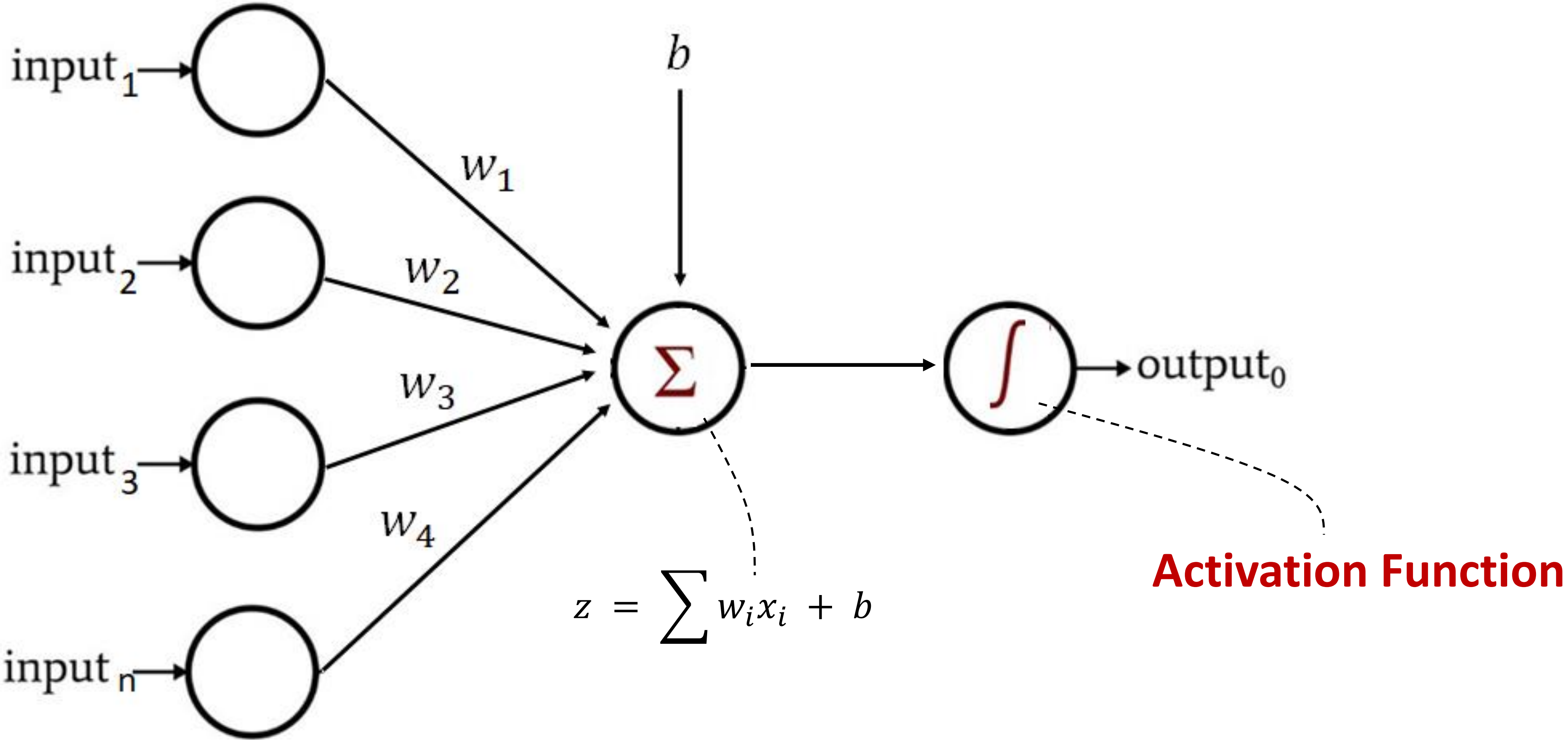


Bias is like the intercept added in a linear equation. It is an additional parameter in the Neural Network which is used to adjust the output along with the weighted sum of the inputs to the neuron.

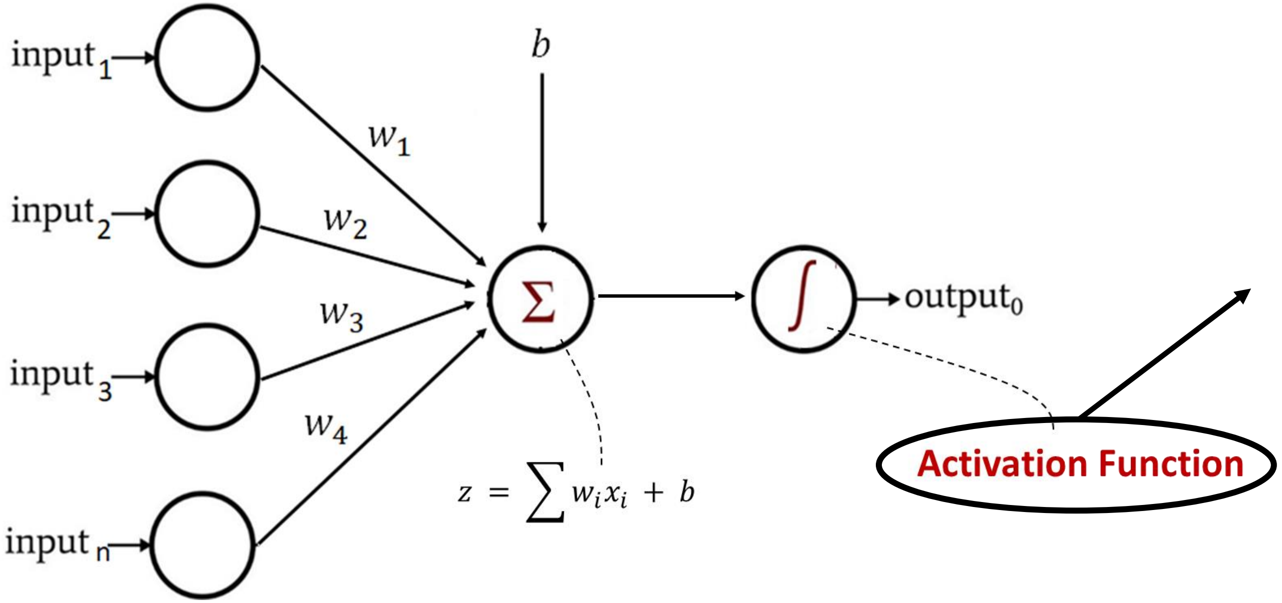
Given that
w1 = 0.5 and **w2** = 0.5 and **b**=-0.1
Say value of **x1** is 0.3 and value of **x2** is 0.8,
So, weighted sum is :

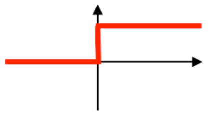
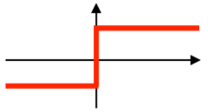
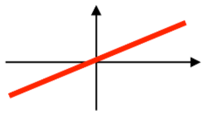
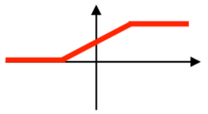
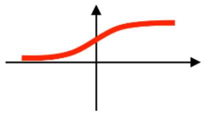
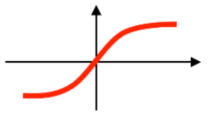
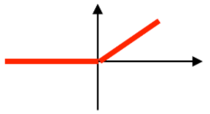
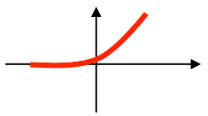
$$z = \boldsymbol{w1} \times \boldsymbol{x1} + \boldsymbol{w2} \times \boldsymbol{x2} + \boldsymbol{b} = 0.5 \times 0.3 + 0.5 \times 0.8 - 0.1 = \mathbf{0.4}$$

One Neuron as a Network (Perceptron)



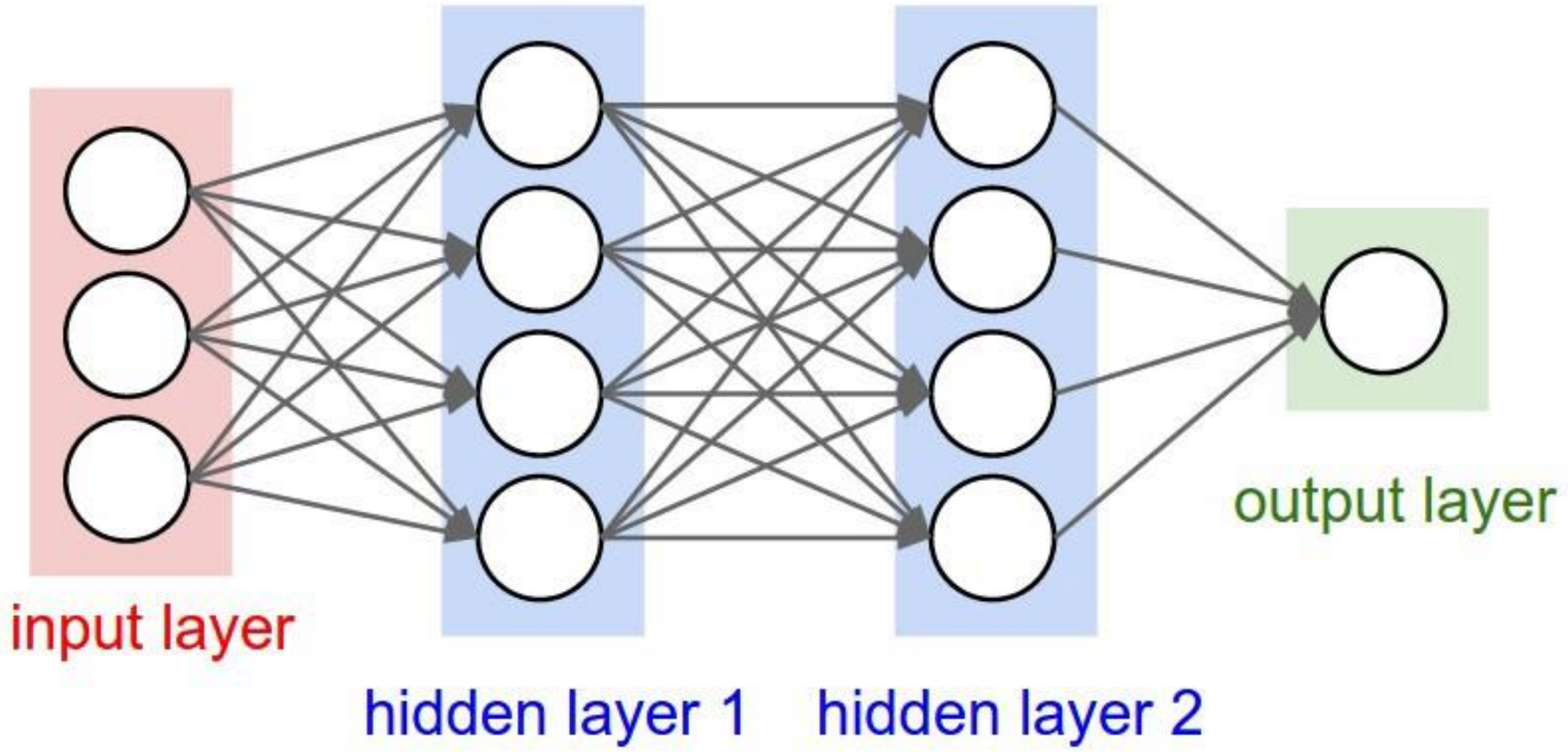
Activation Function



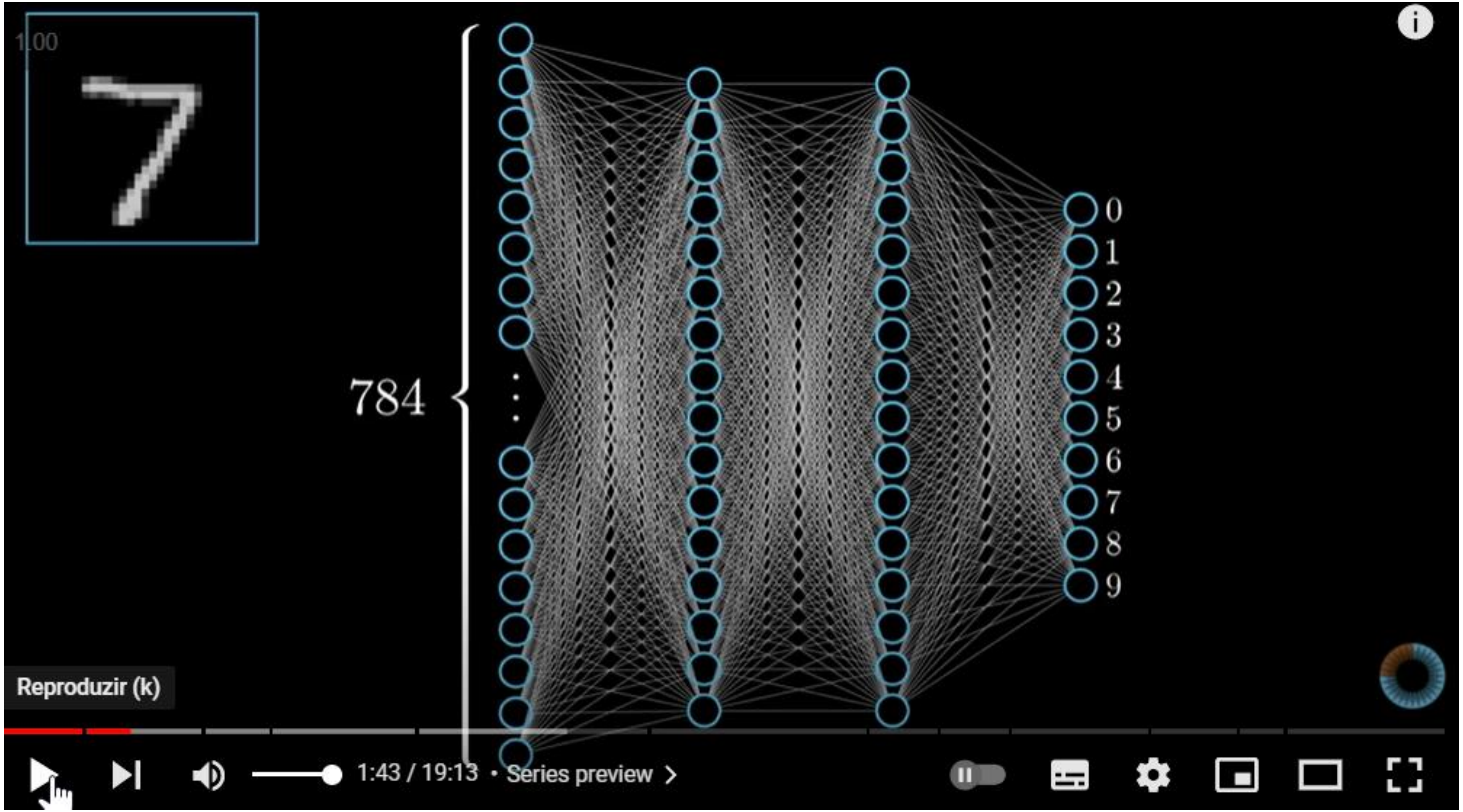
Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Hidden Layers

Adding more layers, allows for more easy representation of the interactions within the input data, as well as allows for more abstract features to be learned and used as input into the next hidden layer.



Neural Networks



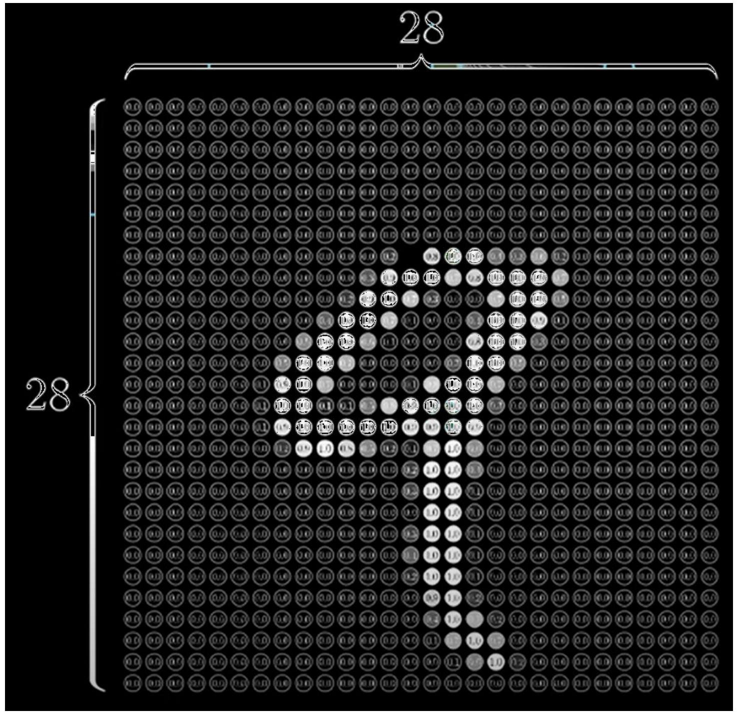
Source: https://www.youtube.com/watch?v=aircAruvnKk

The inputs are fed simultaneously into the **input layer**.

The weighted outputs of these units are fed into **hidden layer**.

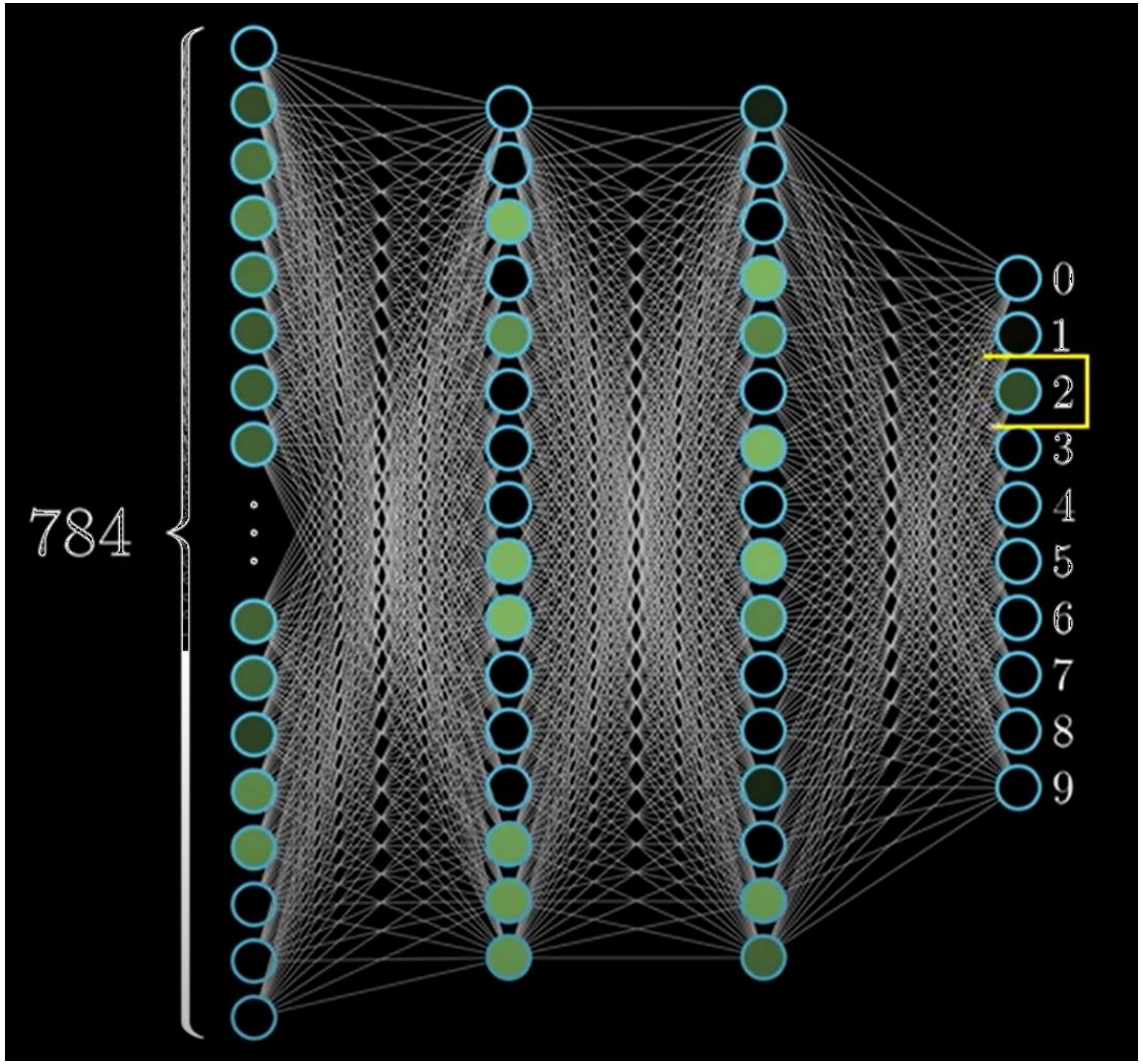
The weighted outputs of the last hidden layer are inputs to units making up the **output layer**.

Inputs and Outputs

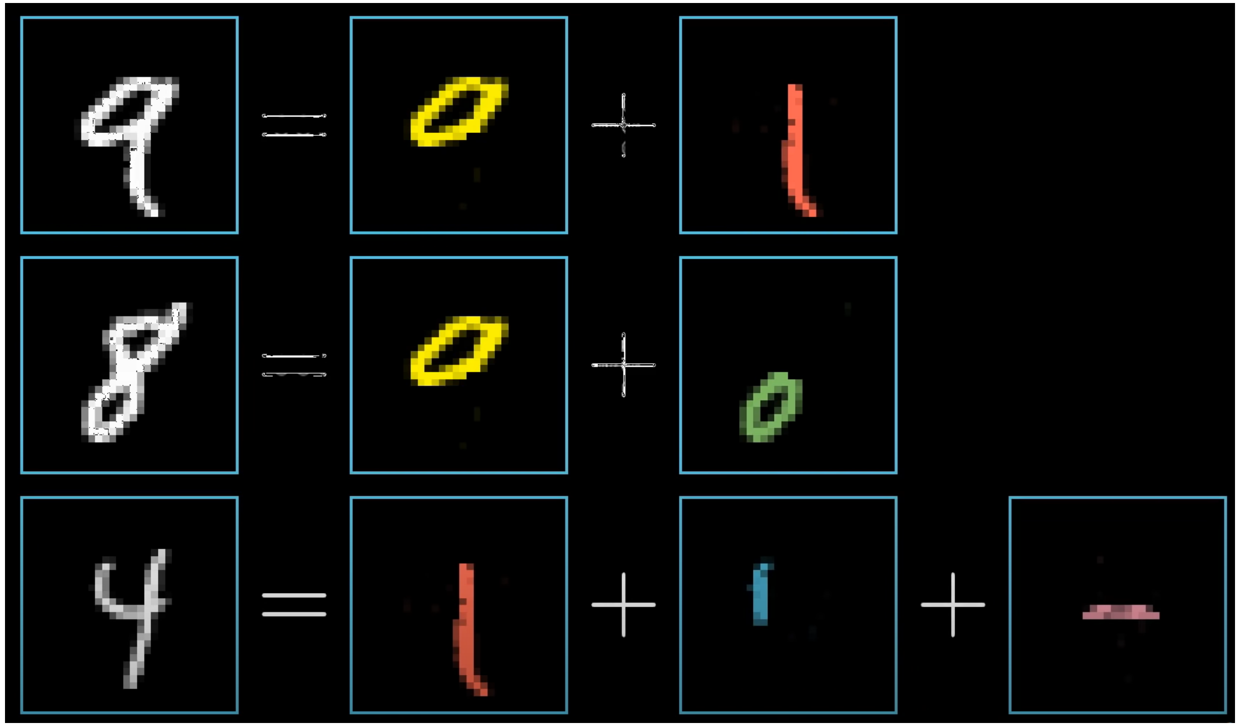


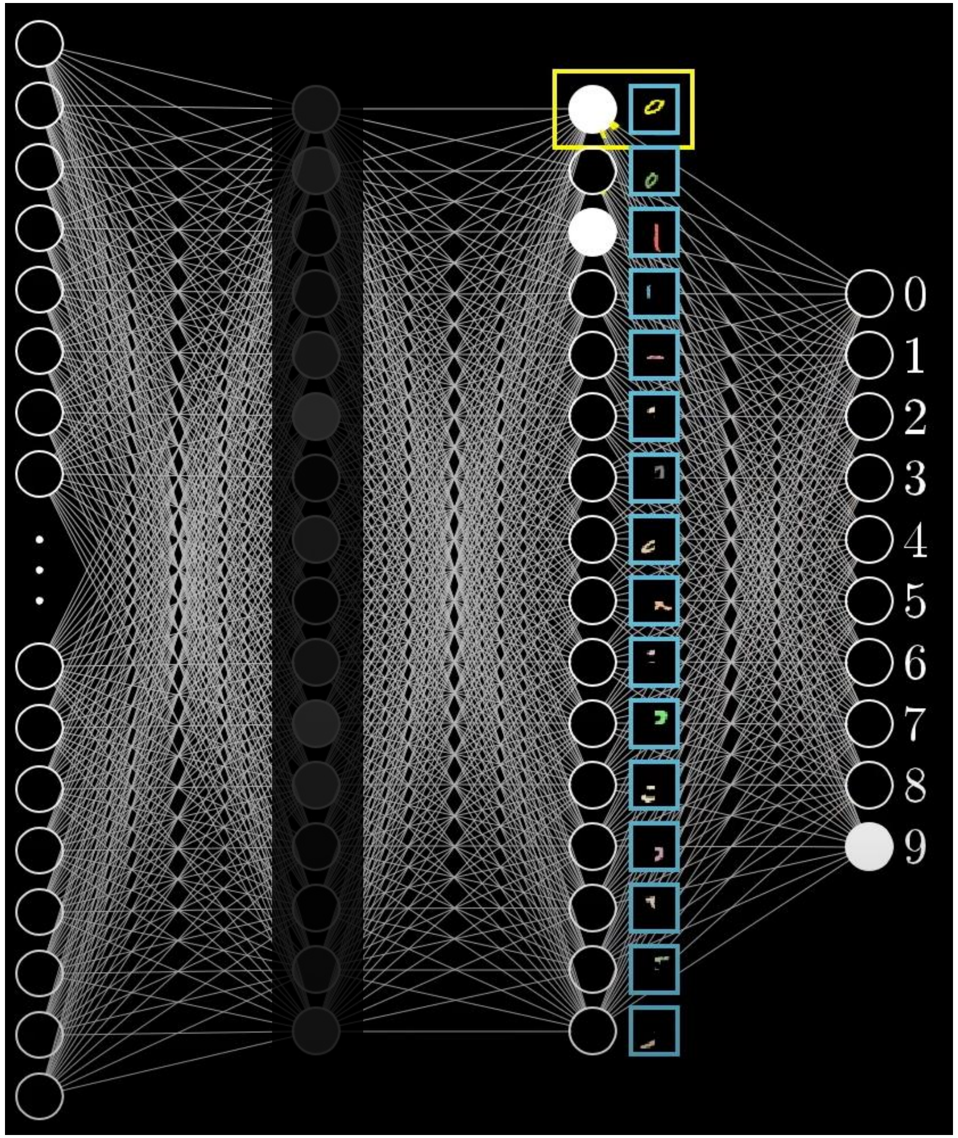
$$\mathbf{28 \times 28 = 784}$$

Number of Features of the Data



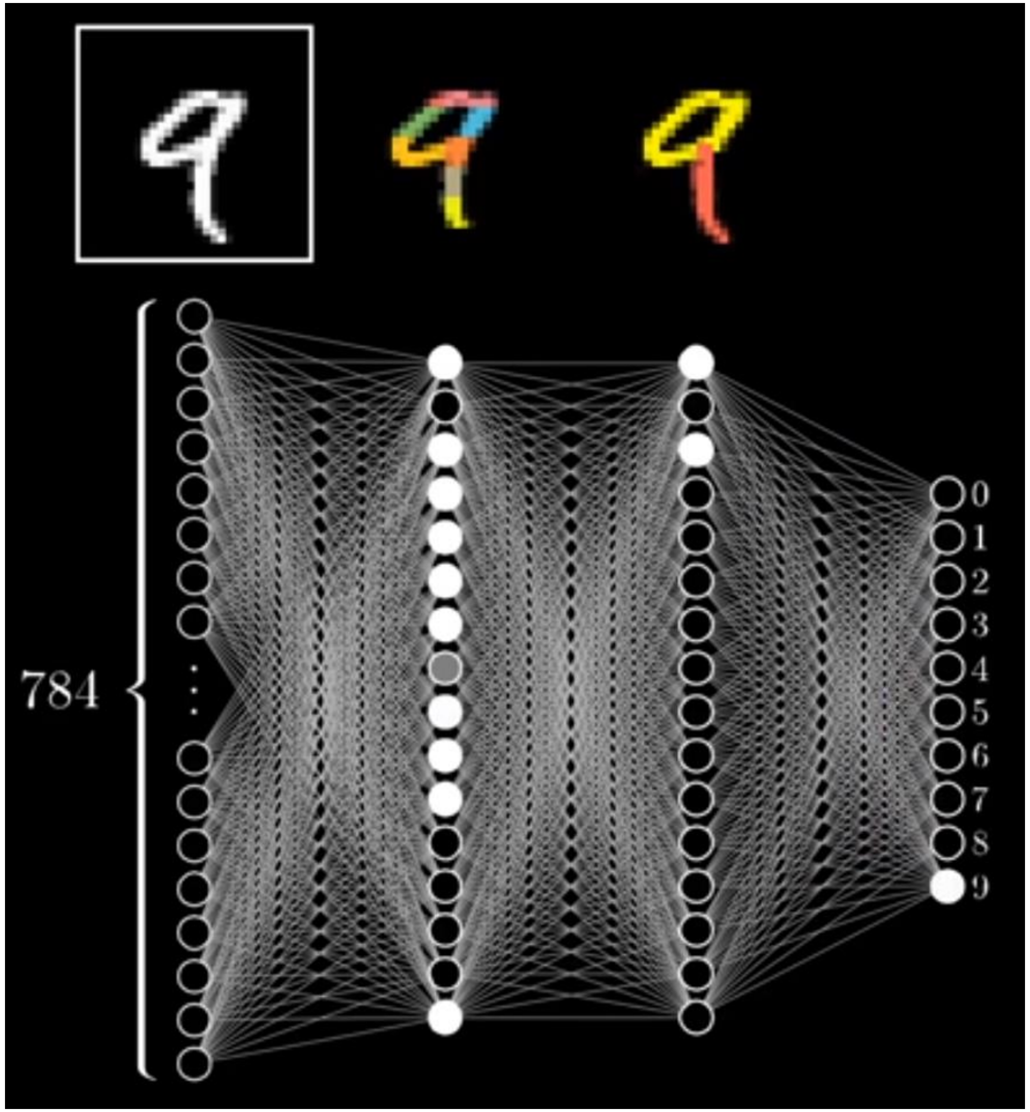
Hidden Layers





Hidden Layers

Adding more layers, allows for more easy representation of the interactions within the input data, as well as allows for more abstract features to be learned and used as input into the next hidden layer.



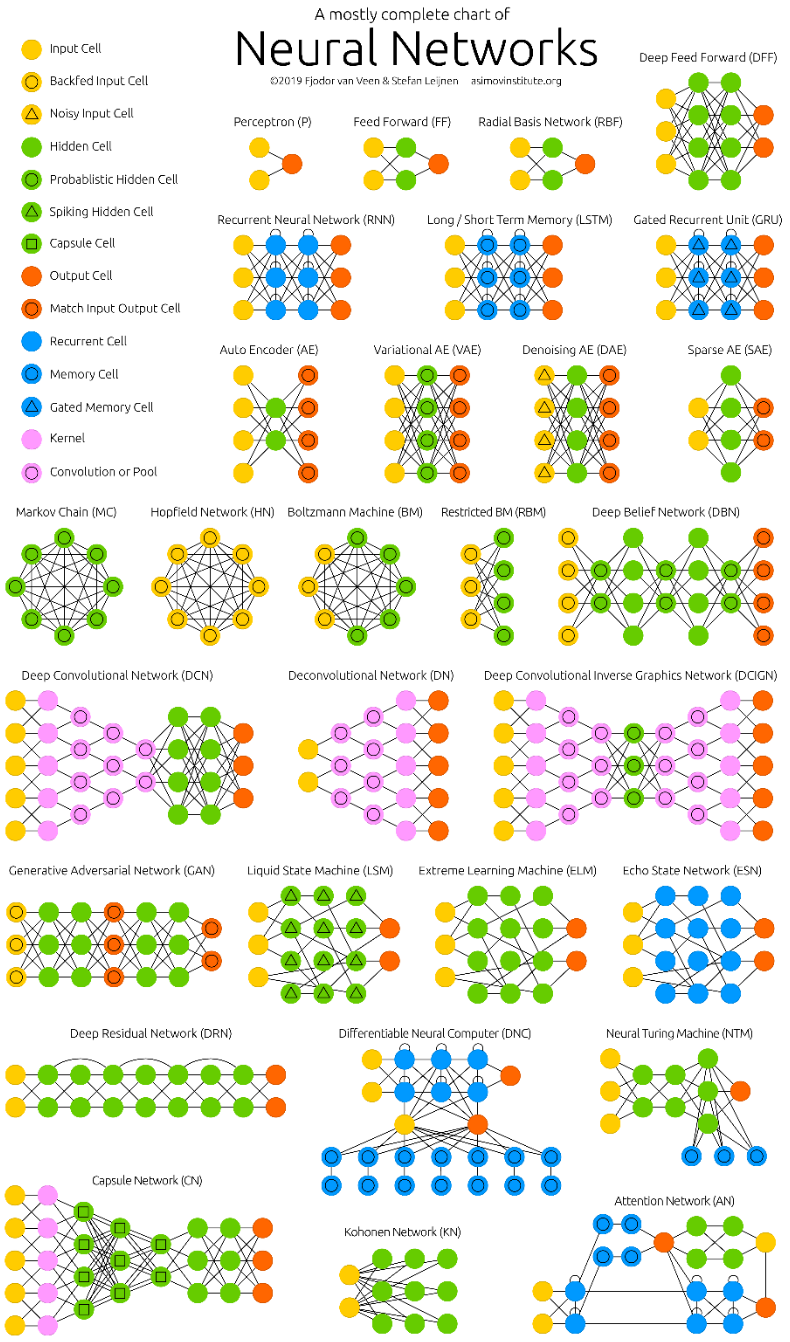
Neural Network Architectures

- There are many **rule-of-thumb methods** for determining the correct number of neurons to use in the hidden layers, such as the following:

  - The number of hidden neurons should be between the size of the input layer and the size of the output layer.
  - The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
  - The number of hidden neurons should be less than twice the size of the input layer

- The selection of an architecture for your neural network will come down to trial and error.

Source: Introduction to Neural Networks for Java *(second edition)* by **Jeff Heaton**



Source: https://www.asimovinstitute.org/neural-network-zoo/

Neural Network Callibration

- What needs to be calibrated in a Neural Network?
- Feature Selection
- Weights and Bias

Today's Class Example

- Number of neurons
- Number of hidden layers
- Neural network architecture
- Activation Function

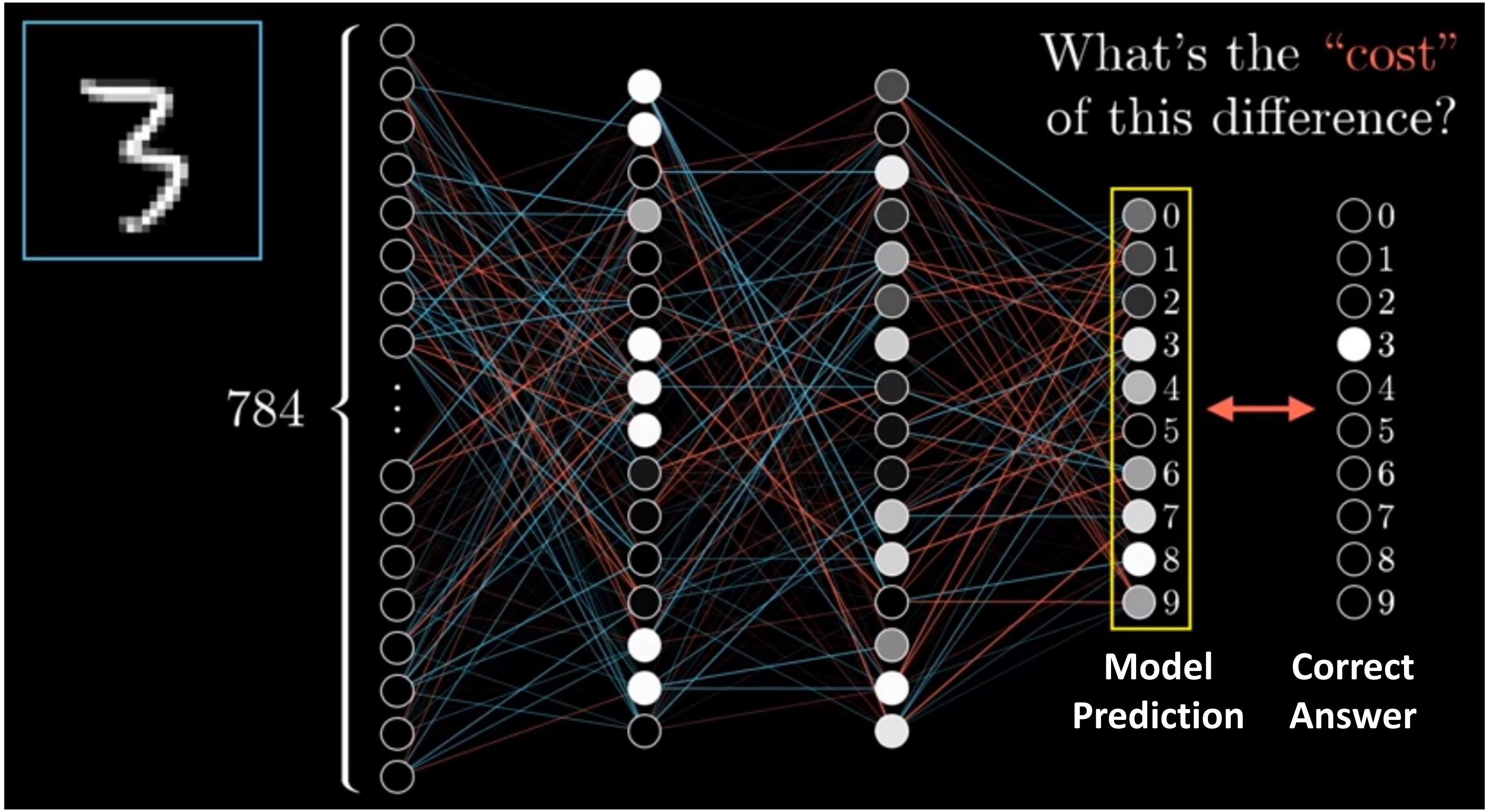
Calibrating Neural Networks

Nuno Antunes Ribeiro

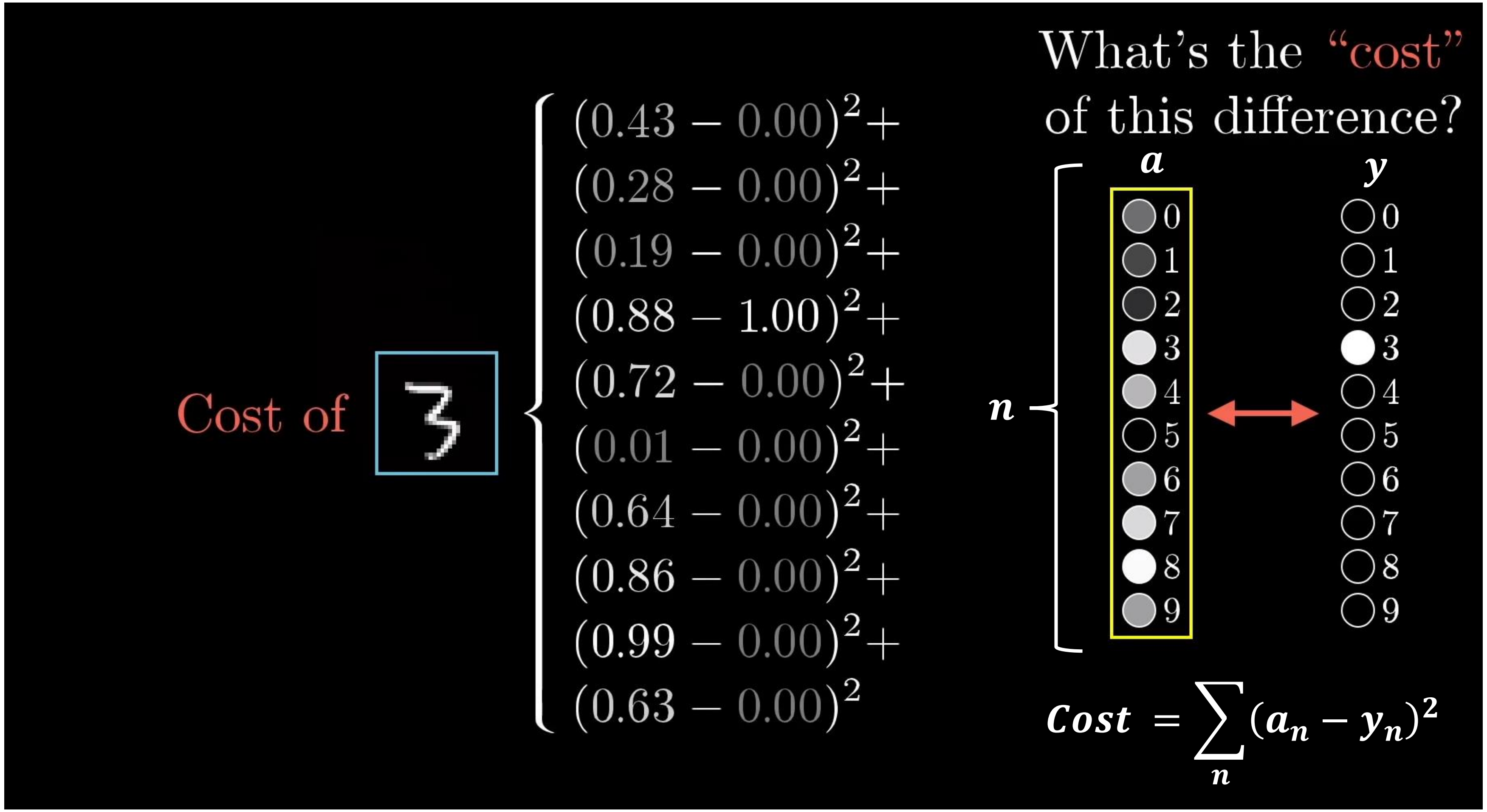
Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN | Engineering Systems and Design

Compute Cost

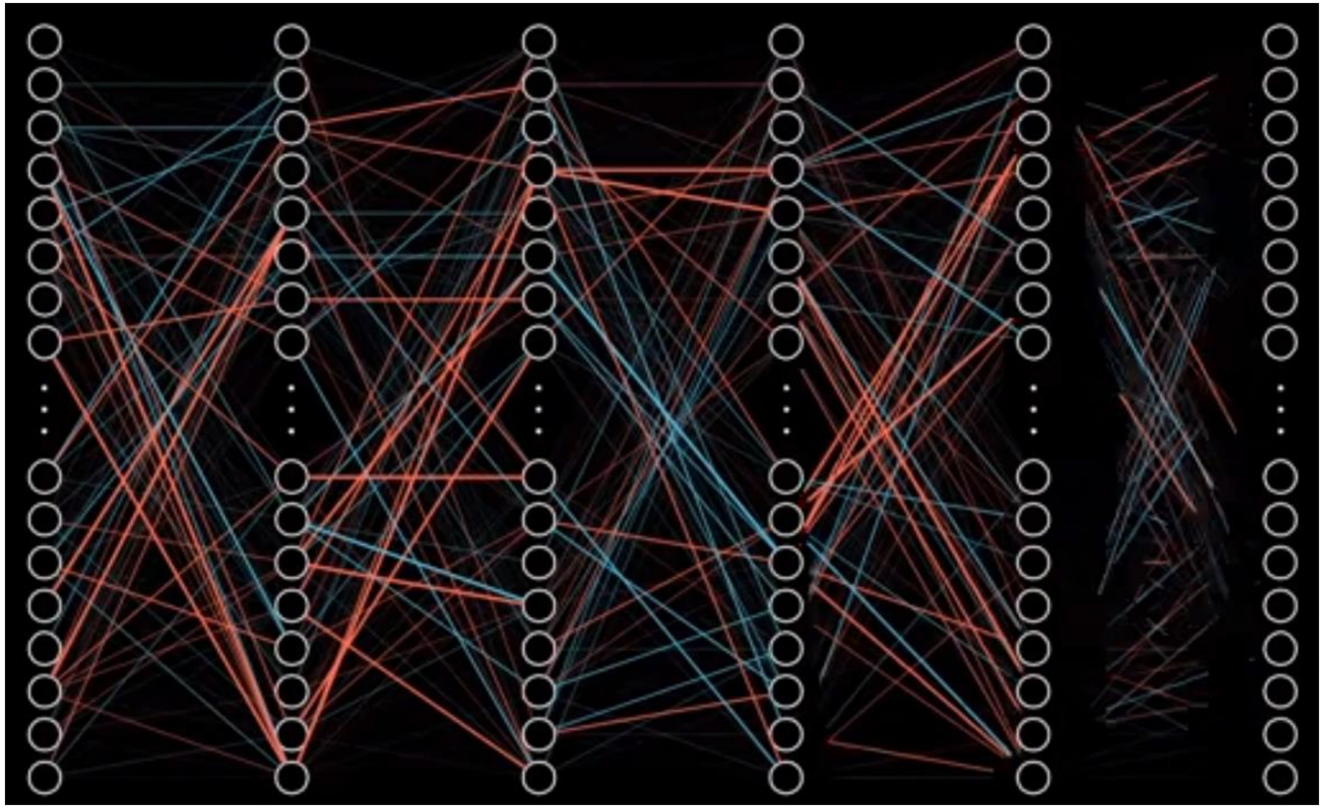


Compute Cost



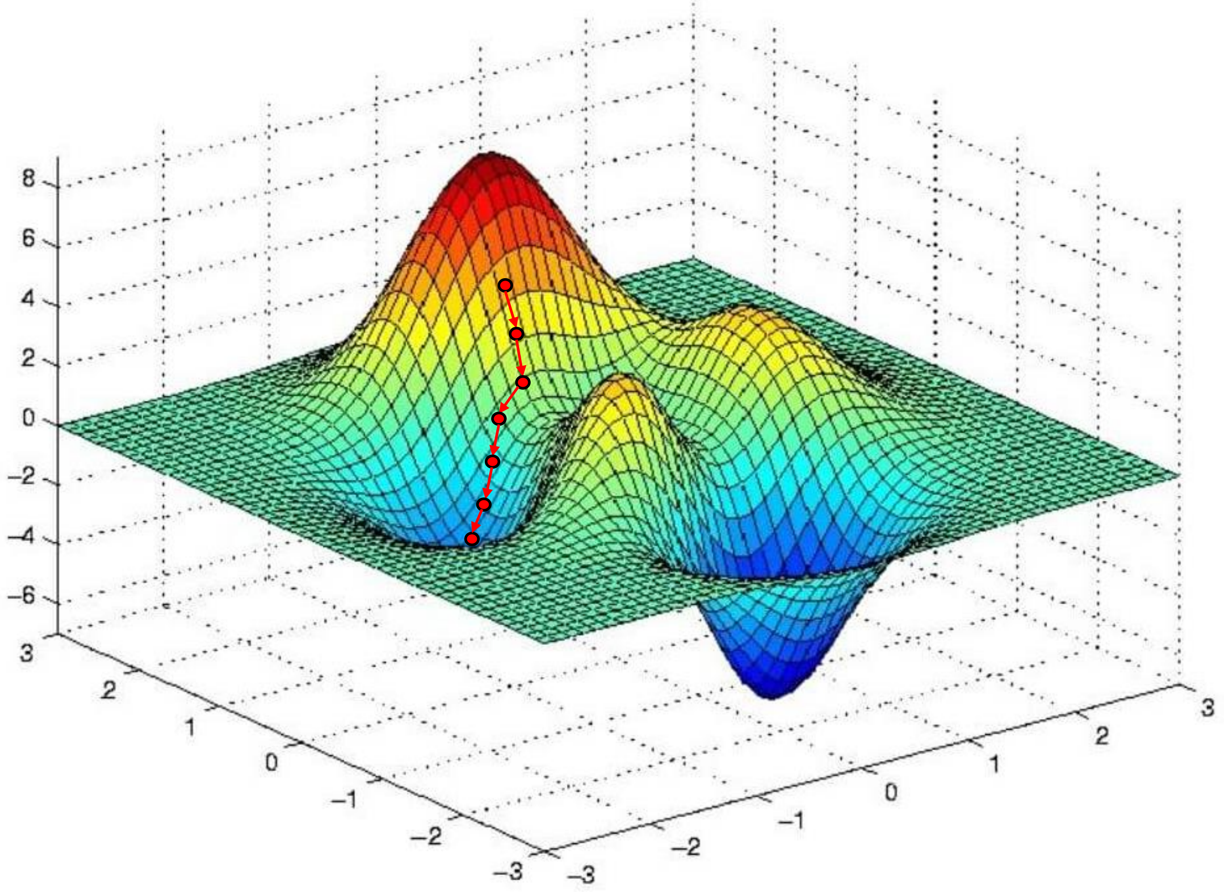
Random Sampling

- **Random sampling** is the most basic technique – it consists on randomly generating weights and bias for each node of the neural network –random sampling typically presents very poor performance



Backpropagation + Gradient Descent

- **Gradient descent + backpropagation** is the most popular algorithm for training neural networks. It is very fast and tend to provide high-quality solutions.
- Gradient descent behaves like a **local search heuristic**. At each iteration, the slope (gradient) of the cost function is computed and **learning step is applied**
- **Backpropagation** is the algorithm used to compute the slope (gradient) at each iteration

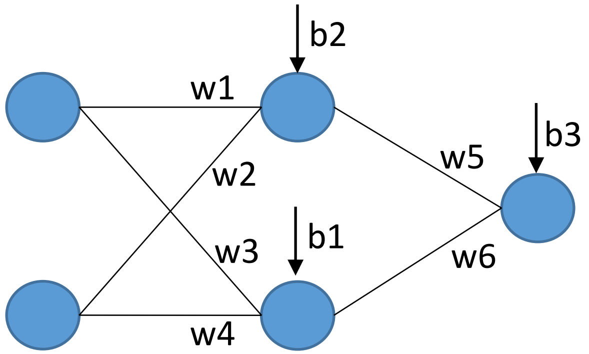


Evolutionary Algorithms

- **Representation:** Weights and Bias Matrices (Input, Hidden and Output Layers)
- **Selection Strategy: roulette wheel selection**
- **Reproduction Strategy:**
  - Crossover - **averaging technique for real operators** – i.e. the offspring weight and bias matrices will just be a linear combination of the parent's matrices
  - Mutation - **random value added** to each entry of the weight and bias matrix
  - To simplify, we will not apply probabilities for mutation and crossover. All offspring will be generated through crossover, only to the last 2 offspring will be subjected to mutation
- **Replacement strategy:** parents will create a set of four children, where the children will be pooled along with their parents and the individual with the best fitness will be chosen to survive.

Crossover

- **Real Operators**



Parents

	w1	w2	w3	w4	w5	w6	b1	b2	b3	
Weights – Solution 1	**0.10**	**0.23**	**0.41**	**0.13**	**0.46**	**0.21**	**0.66**	**0.22**	**0.19**	**P1**
Weights – Solution 2	**0.15**	**0.31**	**0.22**	**0.64**	**0.34**	**0.24**	**0.57**	**0.14**	**0.33**	**P2**

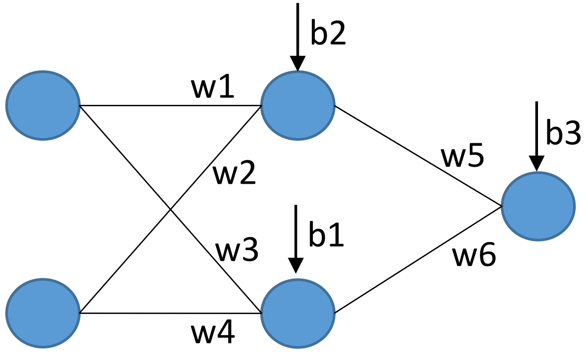
$\alpha P1_i + (1-\alpha)P2_i \longrightarrow \alpha$ generated randomly U(0,1)

Offspring

0.14	**0.29**	**0.28**	**0.49**	**0.38**	**0.23**	**0.60**	**0.16**	**0.29**	e.g. $\alpha = 0.3$

Mutation

- **Real Operators**



Parents

	w1	w2	w3	w4	w5	w6	b1	b2	b3
Weights – Solution	**0.10**	**0.23**	**0.41**	**0.13**	**0.46**	**0.21**	**0.66**	**0.22**	**0.19**
U(0,1)	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**
U(-0.2,0.2)	**0.05**	**0**	**0**	**0**	**-0.1**	**0**	**0**	**0**	**0**

Offspring

0.15	**0.23**	**0.41**	**0.13**	**0.36**	**0.21**	**0.66**	**0.22**	**0.19**

Calibrating Neural Networks in Python

Nuno Antunes Ribeiro

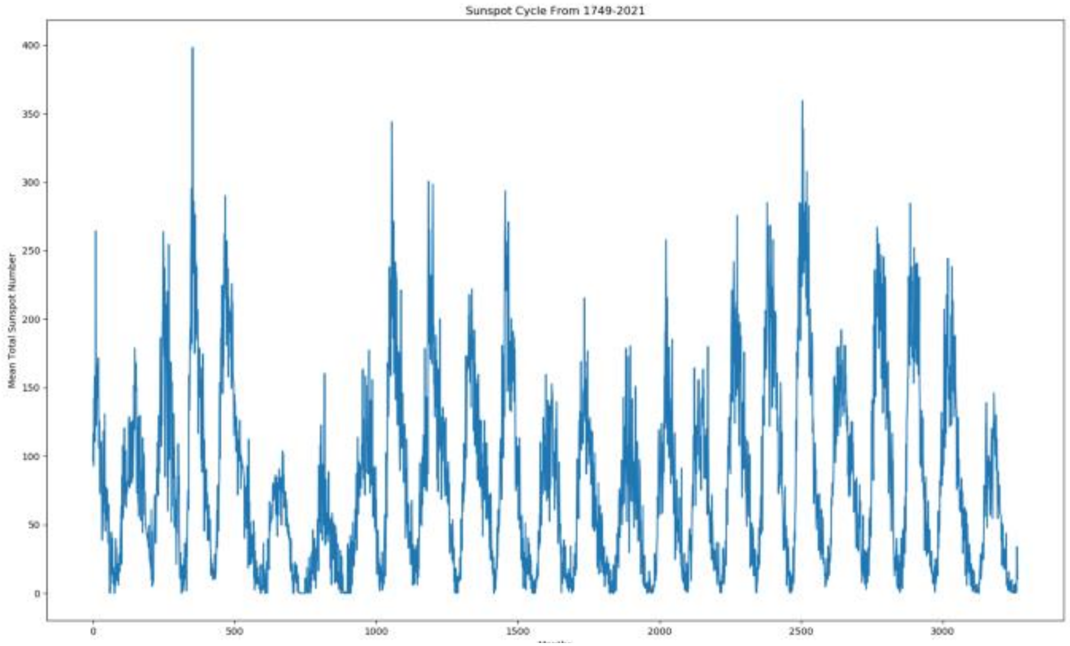
Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

Sunspot Cycle

- For **one dimensional time series problems**, you only have a single variable and a time index.
- The input layer is represented by a '**window**' that spans some value of indices before the current value.
- Example: For the first row, we want to predict the value 110.5, which we can do by feeding 141.7, 139.2, and 158 as the input to our neural network.
- Our **'window' size is three**.
- The optimal size of the window will be calibrated using the **genetic algorithm** (Feature Selection)

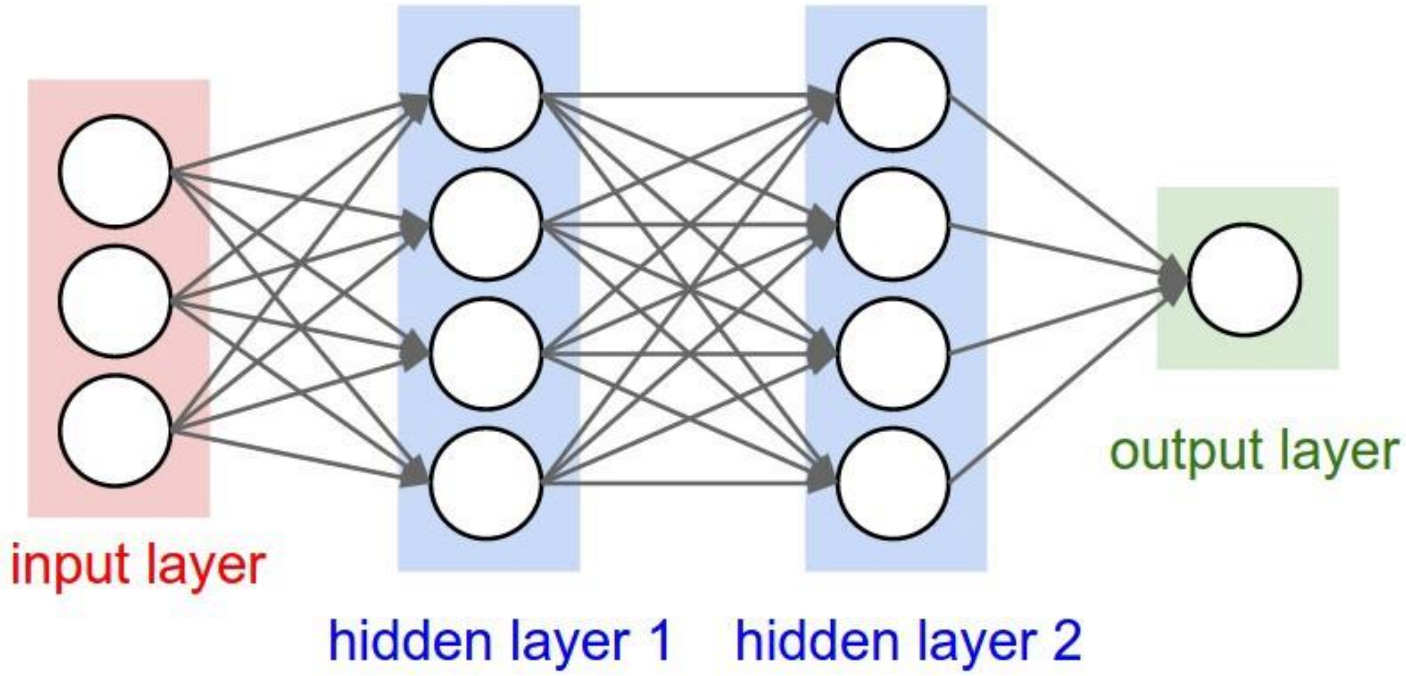


Gray Boxes Represent the Input values for predicting the value in the Black Box

141.7	139.2	158	110.5	126.5	264.8	142
141.7	139.2	158	110.5	126.5	264.8	142
141.7	139.2	158	110.5	126.5	264.8	142
141.7	139.2	158	110.5	126.5	264.8	142

Neural Network Arch. - Sunspot Cycle

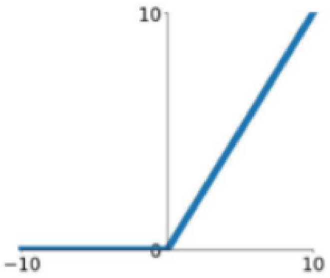
- To keep things simple, our neural network will have **two hidden layers**, each with **5 hidden nodes** and the **ReLU activation function**. However, the number of inputs will change dependent upon the ‘window’ size.
- If window size = 3 , then 3 input nodes. We will be testing window sizes that vary from 3 to 10.



ReLU activation function

ReLU

$$\max(0, x)$$




```python
# ReLU Activation Function
def relu(x):
    return np.maximum(x, 0)

```

NN Phyton

- Because **we will be evolving the weights of a neural network, we need to implement one from scratch** as it would require a lot of manoeuvring to do so in common Python implementations.

Neural Network Code

```python
class EvolvableNetwork:

    # Layer Nodes is a List of int values denoting the number of nodes per Layer
    # For example, if Layer_nodes = [3, 5, 3], we have three hidden Layers with a 3-5-3 node architecture
    # num_input and num_output refer to the number of input and output variables
    # I will explain the purpose of the Initialize boolean Later, but simply if False, do not create the weight
    # and bias matrices
    def __init__(self, layer_nodes, num_input, num_output, initialize=True):
        self.layer_count = len(layer_nodes)
        self.layer_nodes = layer_nodes
        self.num_input = num_input
        self.num_output = num_output
        self.activation_function = relu

        self.layer_weights = []
        self.layer_biases = []

        if not initialize:  # I will discuss the purpose of this Later
            return

        # create the NxM weight and bias matrices for input Layer
        self.layer_weights.append(
            np.random.uniform(-1, 1, num_input * layer_nodes[0]).reshape(num_input, layer_nodes[0]))
        self.layer_biases.append(np.random.uniform(-1, 1, layer_nodes[0]))

        # create the weight matrices for Hidden Layers
        for i in range(1, self.layer_count):
            self.layer_weights.append(
                np.random.uniform(-1, 1, layer_nodes[i-1]*layer_nodes[i]).reshape(layer_nodes[i-1], layer_nodes[i]))
            self.layer_biases.append(np.random.uniform(-1, 1, layer_nodes[i]).reshape(1, layer_nodes[i]))

        # Create the weight and bias matrices for output Layer
        self.layer_weights.append(
            np.random.uniform(-1, 1, layer_nodes[self.layer_count-1]*num_output).reshape(layer_nodes[self.layer_count-1],
                                                                                          num_output))
        self.layer_biases.append(np.random.uniform(-1, 1, num_output).reshape(1, num_output))

    def predict(self, x):  # same as forward pass, performs matrix multiplication of the weights
        output = self.activation_function(np.dot(x, self.layer_weights[0]) + self.layer_biases[0])
        for i in range(1, self.layer_count + 1):
            if i == self.layer_count:  # Last Layer so don't use activation function
                output = (np.dot(output, self.layer_weights[i]) + self.layer_biases[i])
            else:
                output = self.activation_function(
                    np.dot(output, self.layer_weights[i]) + self.layer_biases[i])
        if self.num_output == 1:  # if there is only one output variable then reshape
            return output.reshape(len(x), )
        return output
```

NN Phyton

In the first iteration we generate randomly the weights and the bias matrices

Once we have our population of solutions we no longer generate the weights and the bias matrices randomly but using the genetic algorithm (the code will appear later)

For a given row of data x, we predict the output by first computing v, and then applying the activation function

```
class EvolvableNetwork:

    # Layer Nodes is a list of int values denoting the number of nodes per layer
    # For example, if layer_nodes = [3, 5, 3], we have three hidden layers with a 3-5-3 node architecture
    # num_input and num_output refer to the number of input and output variables
    # I will explain the purpose of the initialize boolean later, but simply if False, do not create the weight
    # and bias matrices
    def __init__(self, layer_nodes, num_input, num_output, initialize=True):
        self.layer_count = len(layer_nodes)
        self.layer_nodes = layer_nodes
        self.num_input = num_input
        self.num_output = num_output
        self.activation_function = relu

        self.layer_weights = []
        self.layer_biases = []

        if not initialize:  # I will discuss the purpose of this later
            return

        # create the NxM weight and bias matrices for input layer
        self.layer_weights.append(
            np.random.uniform(-1, 1, num_input * layer_nodes[0]).reshape(num_input, layer_nodes[0]))
        self.layer_biases.append(np.random.uniform(-1, 1, layer_nodes[0]))

        # create the weight matrices for Hidden Layers
        for i in range(1, self.layer_count):
            self.layer_weights.append(
                np.random.uniform(-1, 1, layer_nodes[i-1]*layer_nodes[i]).reshape(layer_nodes[i-1], layer_nodes[i]))
            self.layer_biases.append(np.random.uniform(-1, 1, layer_nodes[i]).reshape(1, layer_nodes[i]))

        # Create the weight and bias matrices for output layer
        self.layer_weights.append(
            np.random.uniform(-1, 1, layer_nodes[self.layer_count-1]*num_output).reshape(layer_nodes[self.layer_count-1],
                                                                                          num_output))
        self.layer_biases.append(np.random.uniform(-1, 1, num_output).reshape(1, num_output))

    def predict(self, x):  # same as forward pass, performs matrix multiplication of the weights
        output = self.activation_function(np.dot(x, self.layer_weights[0]) + self.layer_biases[
        for i in range(1, self.layer_count + 1):
            if i == self.layer_count:  # last layer so don't use activation function
                output = (np.dot(output, self.layer_weights[i]) + self.layer_biases[i])
            else:
                output = self.activation_function(
                    np.dot(output, self.layer_weights[i]) + self.layer_biases[i])
        if self.num_output == 1:  # if there is only one output variable then reshape
            return output.reshape(len(x), )
        return output
```

$$z = \sum w_i x_i + b$$

GA Selection Strategy

- For selection, we will use roulette wheel selection, which works by creating a cumulative distribution from the proportion of an individual being chosen based off its fitness value --- the cumulative distribution is computed later

```
def roulette_wheel_selection(cumulative_sum, n):
    ind = []
    r = np.random.uniform(0, 1, n)
    for i in range(0, n):
        index = 0
        while cumulative_sum[index] < r[i]:
            index += 1
        ind.append(index)
    return ind
```

GA Crossover

- For crossover, we will implement the averaging technique, which takes a linear combination of the parent values. For our problem, the offspring weight and bias matrices will just be a linear combination of the parent's matrices. To do this, we first instantiate a new EvolvableNetwork, except this time with initialize equal to False, because we do not want the weight and bias matrices to be initialized to random values as we will create them from the parents:

```python
# p1 and p2 are parents
# const_cross is the coefficient for the linear combination, ranges between [0, 1]
# if near 0, it will favor p1; if near 1, it will favor p2; if equal to 0.5, it is the mean between the parents
def crossover(p1, p2, const_cross):
    # initialize new network with empty layer weights and biases
    child = EvolvableNetwork(layer_nodes=p1.layer_nodes, num_input=p1.num_input, num_output=p1.num_output,
                             initialize=False)
    # fill child weight and bias matrices from the parents
    for i in range(0, p1.layer_count+1):
        child.layer_weights.append( (1-const_cross)*p1.layer_weights[i]+const_cross*p2.layer_weights[i])
        child.layer_biases.append( (1-const_cross)*p1.layer_biases[i]+const_cross*p2.layer_biases[i])

    return child
```

GA Mutation

- For mutation, we will simply add some small random value to each entry of the weight and bias matrix for all matrices:

```python
# const_mutate is the max value to mutate by
def mutation(child, const_mutate):
    # loop over all layers
    for i in range(0, child.layer_count+1):
        n, c = child.layer_weights[i].shape
        # these are the random weights to add the current child
        r_w = np.random.uniform(-const_mutate, const_mutate, n*c)
        # loop over all rows and columns for the current layer
        for nr in range(0, n):
            for nc in range(0, c):
                child.layer_weights[i][nr, nc] += r_w[nr*c+nc]

    # loop over all layers
    for i in range(0, child.layer_count+1):
        c = child.layer_biases[i].shape[0]
        # these are the random weights to add the current child
        r_w = np.random.uniform(-const_mutate, const_mutate, c)
        # loop over all columns of the vector
        for nc in range(0, c):
            child.layer_biases[i][nc] += r_w[nc]
```

GA Reproduction

- To simplify, we will not apply probabilities for mutation and crossover. All offspring will be generated through crossover, only to the last 2 offspring will be subjected to mutation

```
# p1 and p2 are parents
# const_mutate is the max value to mutate by
def reproduce(p1, p2, const_mutate, fitness_function, train_data):
    # create a different gamma coefficient for averaging
    # crossover for each offspring
    c_cross = np.random.normal(0.5, 0.15, 4)
    ch1 = crossover(p1, p2, c_cross[0])
    ch2 = crossover(p1, p2, c_cross[1])
    ch3 = crossover(p1, p2, c_cross[2])
    ch4 = crossover(p1, p2, c_cross[3])
    # mutate only two of the individuals
    mutation(ch3, const_mutate)
    mutation(ch4, const_mutate)
    # pool offspring with parents
    all = [p1, p2, ch1, ch2, ch3, ch4]
    fit = fitness_function(all, train_data)
    # return the individual with the min fitness value
    return all[np.argmin(fit)]
```

Fitness Function

- The fitness function will take the Mean Sum of Square Errors (MSE) between the predicted and the actual values for our time series problem

$$MSE = \frac{\sum_i^n (y_i - \hat{y}_i)^2}{n}$$

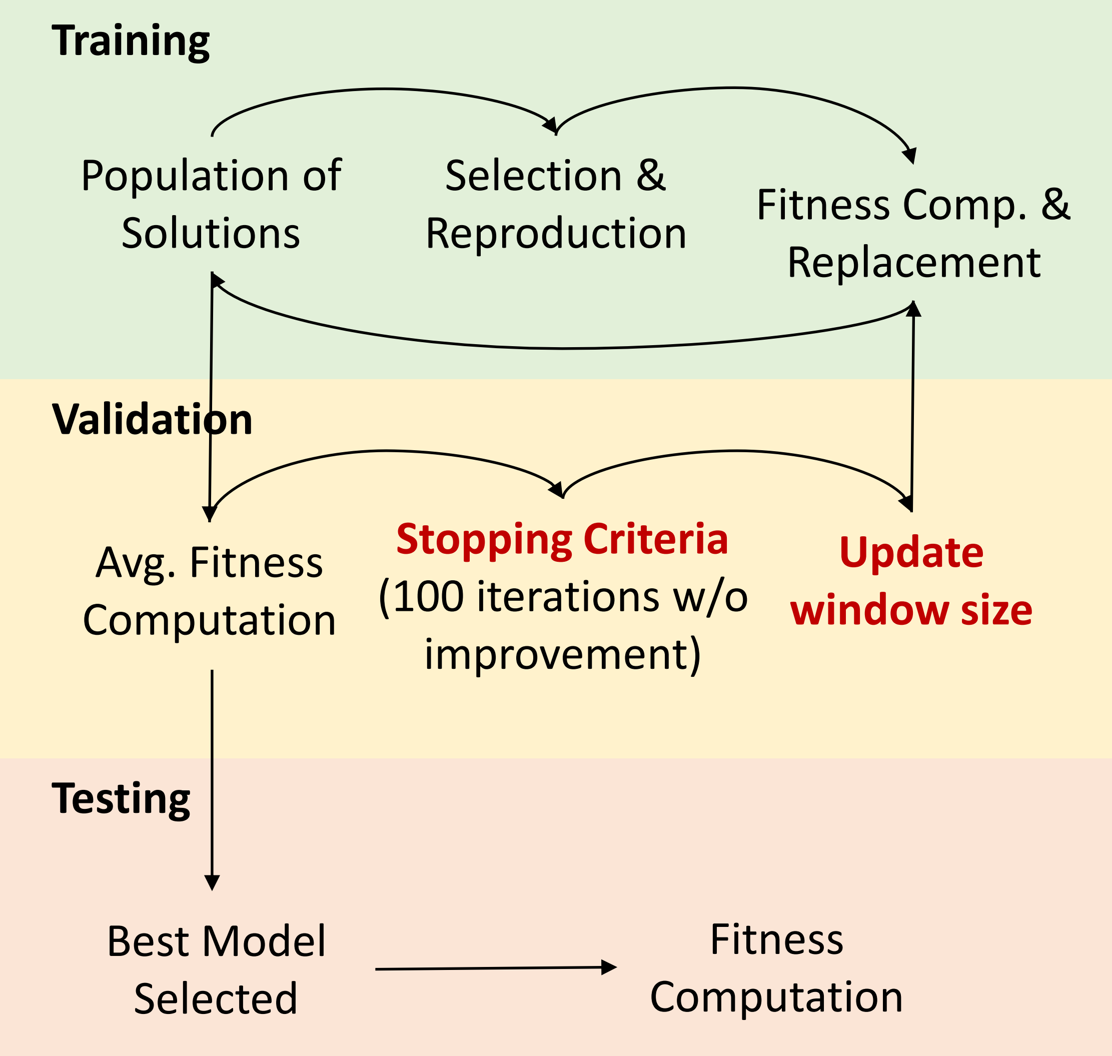
- Since we want to minimize the MSE error function, we need to scale our fitness values.

```python
# models represents the list of networks
# data is composed of the time series input and output
def fitness_function(models, data):
    mse_values = []
    x = data[0]
    y = data[1]
    for network in models:
        predictions = network.predict(x)
        mse = mean_squared_error(y, predictions)
        mse_values.append(mse)
    return np.asarray(mse_values)


def scale_fitness(x):
    return 1 / (1 + x)
```

Training, Testing, Valiation Datasets

- We split our data into the **training, validation, and testing** sets.
- Our algorithms will be trained using the training dataset.
- The validation will be used to compare our models.
- Lastly, after we've chosen our final model, we will evaluate it's accuracy through the test dataset.

```
# compile the data

df = pd.read_csv("Sunspots.csv")
y = np.asarray(df['Monthly Mean Total Sunspot Number'])
size = len(y)
# 50% of data for training
train_ind = int(size * 0.50)
# 25% of data for validation and other 25% for testing
val_ind = int(size * 0.75)
```



https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7

GA Routine

```python
# const_mutate in our example is the actual max value to mutate by, not percentage
# use train and val data to prevent overfitting - we early stop if the mean of
# validation data increases for three straight iterations
def evolve(init_gen, const_mutate, max_iter, train_data, val_data):
    gen = init_gen
    mean_fitness = []
    val_mean = []  # validation mean value
    best_fitness = []
    prev_val = 1000
    n = len(gen)
    val_index = 0
    for k in range(0, max_iter):
        fitness = fitness_function(gen, train_data)
        # scale so that large values -> small
        # and small values -> large
        scaled_fit = scale_fitness(fitness)

        # create distribution for proportional selection
        fit_sum = np.sum(scaled_fit)
        fit = scaled_fit / fit_sum
        cumulative_sum = np.cumsum(fit)

        selected = roulette_wheel_selection(cumulative_sum, n)
        mates = roulette_wheel_selection(cumulative_sum, n)

        children = []
        for i in range(0, n):
            children.append(reproduce(gen[selected[i]], gen[mates[i]], const_mutate, fitness_function, train_data))

        gen_next = children

        # evaluate training data
        fit = fitness_function(gen_next, train_data)
        fit_mean = np.mean(fit)
        fit_best = np.min(fit)
        mean_fitness.append(fit_mean)
        best_fitness.append(fit_best)

        # evaluate validation data
        val_fit = fitness_function(gen_next, val_data)
        val_fit_mean = np.mean(val_fit)
        val_mean.append(val_fit_mean)

        print("Generation: " + str(k))
        print(" Best: {}, Avg: {}".format(fit_best, fit_mean))
        print(" Validation: {}".format(val_fit_mean))

        gen = gen_next
```

Results

```
Best Validation Fitness Values Per Window Size:
Window Size: 3 - Validation MSE: 618.7214932489885
Window Size: 4 - Validation MSE: 581.9126893349862
Window Size: 5 - Validation MSE: 614.0682346045121
Window Size: 6 - Validation MSE: 591.9790816024835
Window Size: 7 - Validation MSE: 619.5943150956987
Window Size: 8 - Validation MSE: 586.8203769039181
Window Size: 9 - Validation MSE: 589.0110004213362
Window Size: 10 - Validation MSE: 561.972516929794
Validation Error: Mean w/ std: 595.5099635177146+-19.04960393916631
Best Model:
 Window Size : 10
 MSE for Test Data Set : 613.3878790323605
```

Evolutionary Algorithms vs (Backpropagation + Gradient Descent)

Nuno Antunes Ribeiro

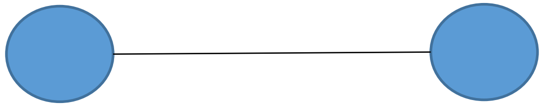
Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

Backpropagation Algorithm

- Using backpropagation to calibrate very simple neural network



Backpropagation Algorithm

- Using backpropagation to calibrate very simple neural network

$$\frac{\partial C}{\partial w^L}$$

We want to know how sensible the cost C_0 is to small changes in the weight w^L (gradient)

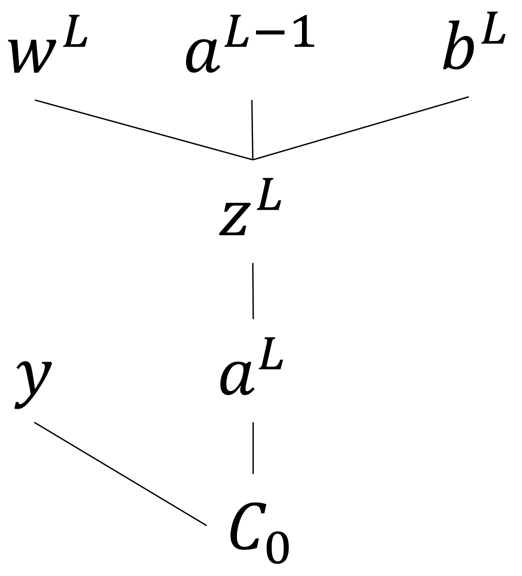
$$C_0 = (a^L - y)^2$$

$$z^L = w^L a^{L-1} + b^L$$

$$a^L = \sigma(z^L)$$

Chain Rule

$$\frac{\partial C}{\partial w^L} = \frac{\partial z^L}{\partial w^L} * \frac{\partial a^L}{\partial z^L} * \frac{\partial C}{\partial a^L}$$

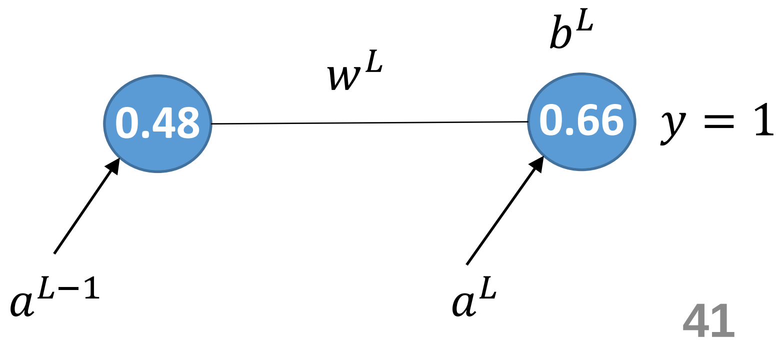


$$\frac{\partial C}{\partial a^L} = 2(a^L - y)$$

$$\frac{\partial a^L}{\partial z^L} = \sigma'(z^L)$$

$$\frac{\partial z^L}{\partial w^L} = a^{L-1}$$

$$\boldsymbol{\frac{\partial C}{\partial w^L} = 2(a^L - y)\sigma'(z^L)a^{L-1}}$$



Backpropagation Algorithm

- Using backpropagation to calibrate very simple neural network

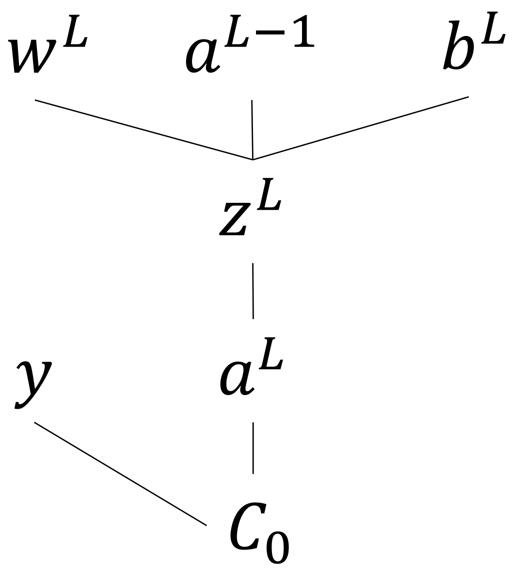
$$\frac{\partial C}{\partial b^L}$$

We want to know how sensible the cost C_0 is to small changes in the bias b^L (gradient)

$$C_0 = (a^L - y)^2$$

$$z^L = w^L a^{L-1} + b^L$$

$$a^L = \sigma(z^L)$$



Chain Rule

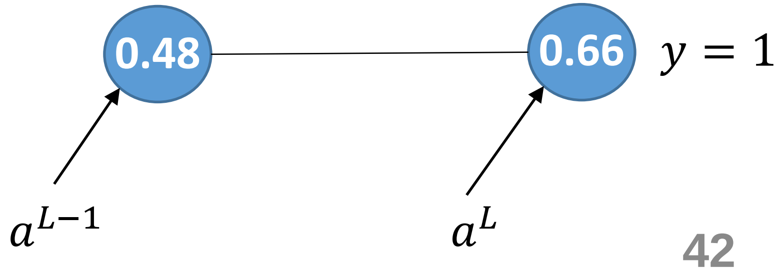
$$\frac{\partial C}{\partial b^L} = \frac{\partial z^L}{\partial b^L} * \frac{\partial a^L}{\partial z^L} * \frac{\partial C}{\partial a^L}$$

$$\frac{\partial C}{\partial a^L} = 2(a^L - y)$$

$$\frac{\partial a^L}{\partial z^L} = \sigma'(z^L)$$

$$\frac{\partial z^L}{\partial b^L} = 1$$

$$\boldsymbol{\frac{\partial C}{\partial b^L} = 2(a^L - y)\sigma'(z^L)}$$



Backpropagation Algorithm

- Using backpropagation to calibrate very simple neural network

$$\frac{\partial C}{\partial a^{L-1}}$$

We want to know how sensible the cost C_0 is to small changes in a^{L-1} (gradient)

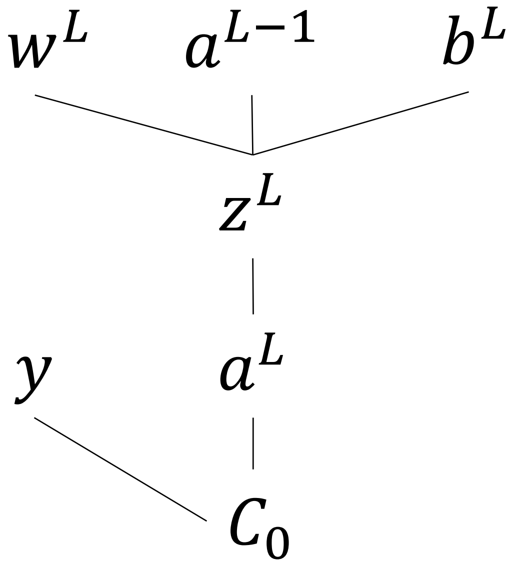
$$C_0 = (a^L - y)^2$$

$$z^L = w^L a^{L-1} + b^L$$

$$a^L = \sigma(z^L)$$

Chain Rule

$$\frac{\partial C}{\partial a^{L-1}} = \frac{\partial z^L}{\partial a^{L-1}} * \frac{\partial a^L}{\partial z^L} * \frac{\partial C}{\partial a^L}$$

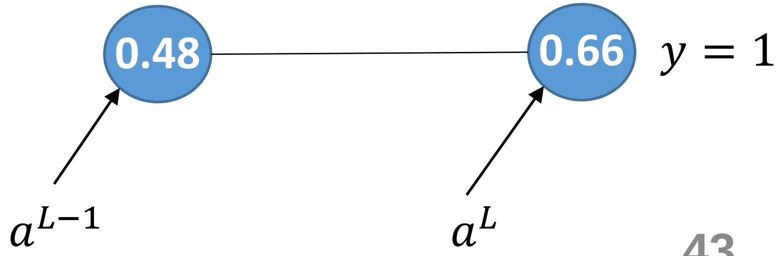


$$\frac{\partial C}{\partial a^L} = 2(a^L - y)$$

$$\frac{\partial a^L}{\partial z^L} = \sigma'(z^L)$$

$$\frac{\partial z^L}{\partial a^{L-1}} = w^L$$

$$\boldsymbol{\frac{\partial C}{\partial a^{L-1}} = 2(a^L - y)\sigma'(z^L)w^L}$$



Backpropagation Algorithm

- Using backpropagation to calibrate very simple neural network

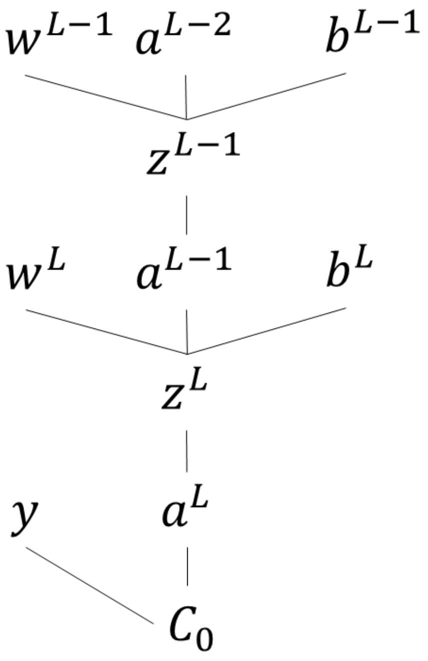
$$\frac{\partial C}{\partial w^{L-1}}$$

We want to know how sensible the cost C_0 is to small changes in a^{L-1} (gradient)

$$z^{L-1} = w^{L-1}a^{L-2} + b^{L-1}$$

$$a^{L} = \sigma(z^{L-1})$$

Chain Rule



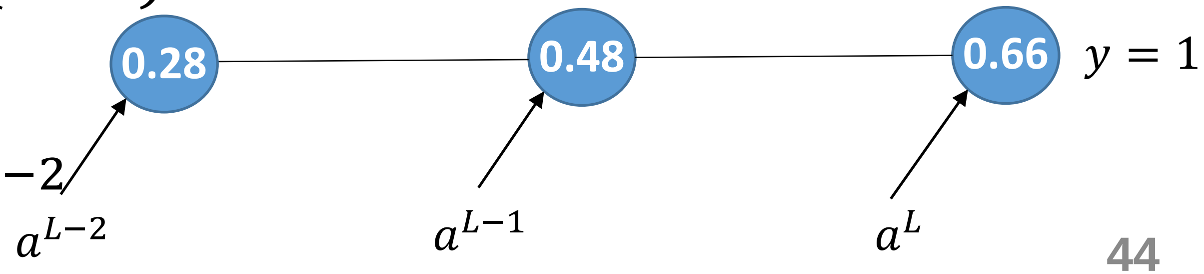
$$\frac{\partial C}{\partial w^{L-1}} = \frac{\partial z^{L-1}}{\partial w^{L-1}} * \frac{\partial a^{L-1}}{\partial z^{L-1}} * \boldsymbol{\frac{\partial C}{\partial a^{L-1}}}$$

$$\boldsymbol{\frac{\partial C}{\partial a^{L-1}} = 2(a^{L} - y)\sigma'(z^{L})w^{L}}$$

Obtained through backpropagation

$$\frac{\partial a^{L-1}}{\partial z^{L-1}} = \sigma'(z^{L-1})$$

$$\frac{\partial z^{L-1}}{\partial w^{L-1}} = a^{L-2}$$



Backpropagation Algorithm

- Using backpropagation to calibrate very simple neural network

$$\frac{\partial C}{\partial w^L} = 2(a^L - y)\sigma'(z^L)a^{L-1}$$

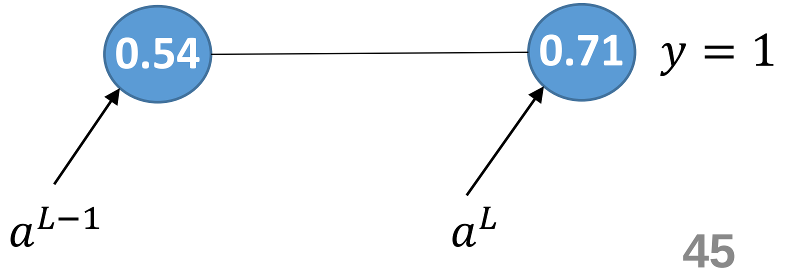
$$C_0 = (a^L - y)^2$$

$$z^L = w^L a^{L-1} + b^L$$

$$a^L = \sigma(z^L)$$

Average of all training examples

$$\frac{\partial C}{\partial w^L} = \frac{1}{n}\sum_{0}^{n-1} \frac{\partial C_k}{\partial w^L}$$



Backpropagation Algorithm

- Using backpropagation to calibrate very simple neural network

$$\frac{\partial C}{\partial w^L} = 2(a^L - y)\sigma'(z^L)a^{L-1}$$

Average of all training examples

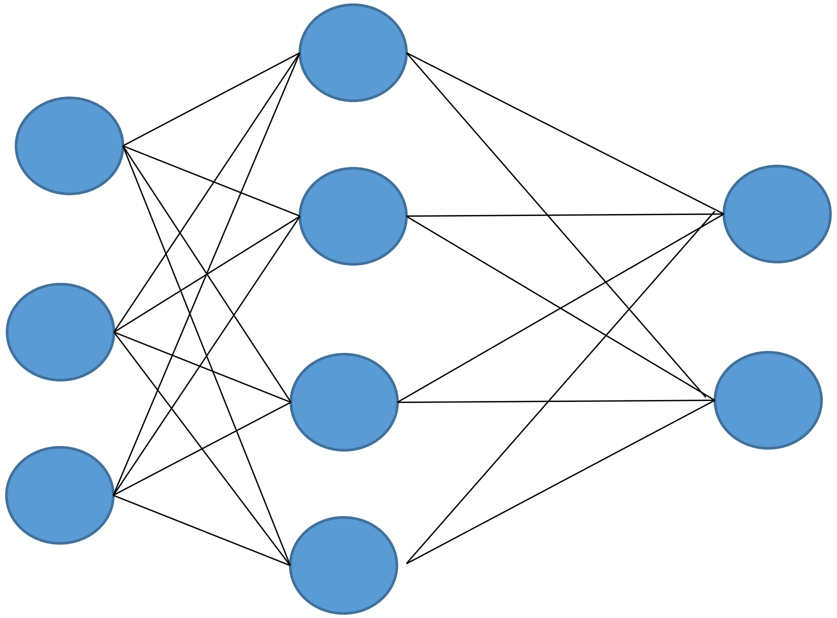
$$\boldsymbol{\frac{\partial C}{\partial w^L}} = \frac{1}{n}\sum_{0}^{n-1}\frac{\partial C_k}{\partial w^L}$$

$$\nabla C = \begin{pmatrix} \frac{\partial C}{\partial w^1} \\ \frac{\partial C}{\partial b^1} \\ \dots \\ \boldsymbol{\frac{\partial C}{\partial w^L}} \\ \frac{\partial C}{\partial b^1} \\ \dots \\ \frac{\partial C}{\partial w^n} \\ \frac{\partial C}{\partial w^n} \end{pmatrix}$$

$$C_0 = (a^L - y)^2$$

$$z^L = w^L a^{L-1} + b^L$$

$$a^L = \sigma(z^L)$$



Backpropagation Algorithm

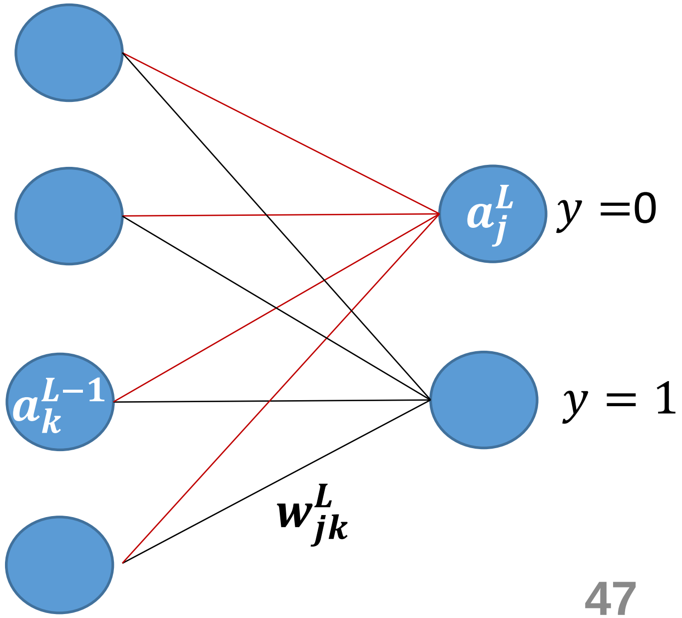
- Using backpropagation to calibrate very simple neural network

$$C_0 = \sum_{j=0}^{n_{L-1}} \left(a_j^L - y_j\right)^2$$

$$\boldsymbol{z_j^L = w_{j1}^L a_1^{L-1} + w_{j2}^L a_2^{L-1} + \cdots + b_j^L}$$

$$a^L = \sigma(z^L)$$

$$a_j^L = \sigma(z_j^L)$$

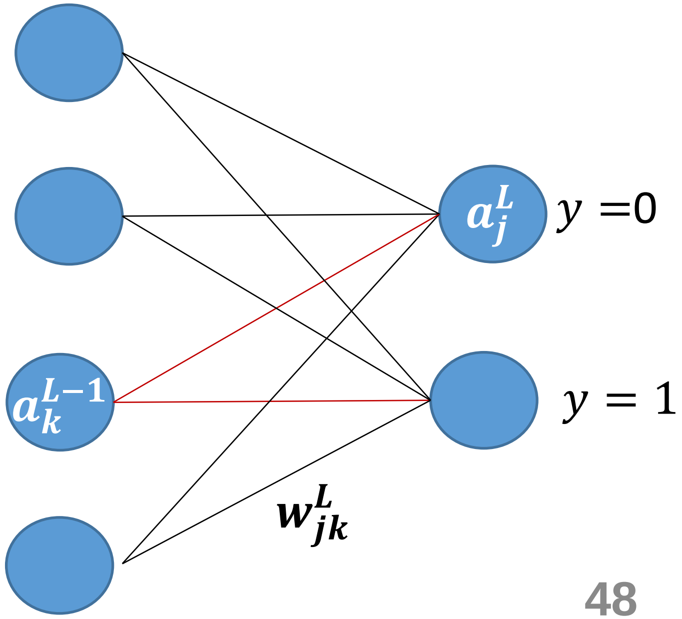


Backpropagation Algorithm

- Using backpropagation to calibrate very simple neural network

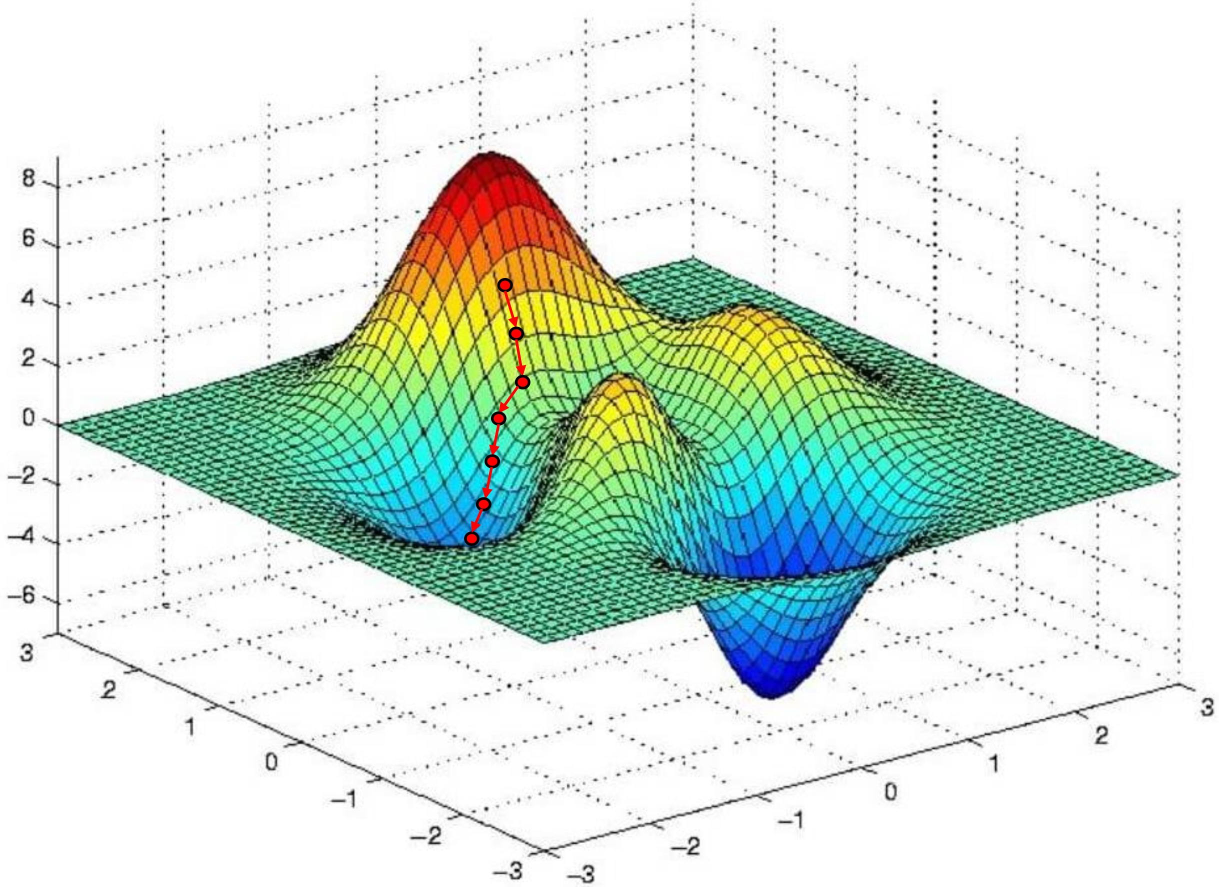
$$\frac{\partial C}{\partial a^{L-1}} = \sum_{j=1}^{n_L} \frac{\partial z_j^L}{\partial a_k^{L-1}} * \frac{\partial a_j^L}{\partial z_j^L} * \frac{\partial C}{\partial a_j^L}$$

$$a^L = \sigma(z^L)$$



Backpropagation + Gradient Descent

- **Gradient descent + backpropagation** is the most popular algorithm for training neural networks. It is very fast and tend to provide high-quality solutions.
- Gradient descent behaves like a **local search heuristic**. At each iteration, the slope (gradient) of the cost function is computed and **learning step is applied**
- **Backpropagation** is the algorithm used to compute the slope (gradient) at each iteration



Backpropagation in Python

- **Scikit-learn** is an open-source software library that provides a Python interface for many machine learning models including artificial neural networks. Scikit-learn does backpropagation automatically when estimating neural networks
- We should test our genetic algorithm against a neural network trained through back-propagation

Backpropagation Algorithm

```
for vision in range(min_window, max_window + 1):
    input = []
    output = []
    for j in range(vision, size):
        input.append(y[(j - vision):j].tolist())
        output.append(y[j])

    input = np.asarray(input)
    output = np.asarray(output)

    temp = np.column_stack((output, input))
    temp = temp[shuffled_indices]

    output = temp[:, 0]
    input = temp[:, 1:]

    y_train = output[0:train_ind]
    y_val = output[train_ind:val_ind]
    y_test = output[val_ind:size]
    x_train = input[0:train_ind]
    x_val = input[train_ind:val_ind]
    x_test = input[val_ind:size]

    mlp = MLPRegressor(hidden_layer_sizes=[5, 5], max_iter=1000, verbose=True,
                       learning_rate='adaptive', early_stopping=True)
    mlp.fit(x_train, y_train)
    predictions = mlp.predict(x_val)
    mse = mean_squared_error(y_val, predictions)
    best_models.append(mlp)
    best_fits.append(mse)
```

For loop through different window sizes

Scikit-learn Backpropagation Implementation

Backpropagation Algorithm

Neural Network Results

```
Best Validation Fitness Values Per Window Size:
Window Size: 3 - Validation MSE: 618.7214932489885
Window Size: 4 - Validation MSE: 581.9126893349862
Window Size: 5 - Validation MSE: 614.0682346045121
Window Size: 6 - Validation MSE: 591.9790816024835
Window Size: 7 - Validation MSE: 619.5943150956987
Window Size: 8 - Validation MSE: 586.8203769039181
Window Size: 9 - Validation MSE: 589.0110004213362
Window Size: 10 - Validation MSE: 561.972516929794
Validation Error: Mean w/ std: 595.5099635177146+-19.04960393916631
Best Model:
 Window Size : 10
 MSE for Test Data Set : 613.3878790323605
```

Backpropagation Results

```
Best Validation Fitness Values Per Window Size:
Window Size: 3 - Validation MSE: 624.4059389734603
Window Size: 4 - Validation MSE: 594.1869554289236
Window Size: 5 - Validation MSE: 620.6140923195566
Window Size: 6 - Validation MSE: 588.7357560279087
Window Size: 7 - Validation MSE: 653.5485403152843
Window Size: 8 - Validation MSE: 605.5655139477342
Window Size: 9 - Validation MSE: 570.7397867110802
Window Size: 10 - Validation MSE: 562.6523735119648
Validation Error: Mean w/ std: 602.5561196544891+-28.01433061551261
Best Model:
 Window Size : 10
 MSE for Test Data Set : 615.4081241153738
```

Not much difference

Comparing GA and Backpropagation

Genetic Algorithm Results		Backpropagation	
Validation	Testing	Validation	Testing
595	613	602	615
666	674	720	661
632	659	739	601
651	700	1896	734
646	626	1876	636
656	705	688	657
621	670	649	693
625	677	651	689
595	607	660	662
624	576	634	574