COMP 2211 Exploring Artificial Intelligence

Artificial Neural Network - Multilayer Perceptron

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

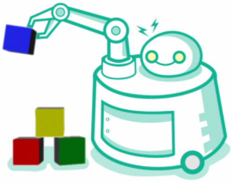
Perceptron

- Recall the perceptron is a simple biological neuron model in an artificial neural network.
- It has a couple of limitations:
  1. Can only represent a limited set of functions.
  2. Can only distinguish (by the value of its output) the sets of inputs that are linearly separable in the inputs.
     - One of the simplest examples of non-separable sets is logical function XOR

How to remedy these limitations?

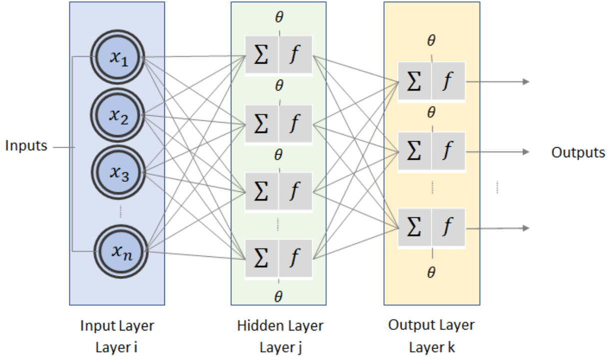
The output of one perceptron can be connected to the input of other perceptron(s). This makes it possible to extend the computational possibilities of a single perceptron.

$\Rightarrow$ Multi-layer Perception



Multi-Layer Perceptron Neural Network

- Multi-layer perceptron (MLP) neural network is a type of feed-forward neural network. (Feed-forward here means nodes in this network do not form a cycle.)
- It consists of three types of layers:
  - Input layer (also called layer i)
  - Hidden layer (also called layer j)
  - Output layer (also called layer k)



- $x_1, x_2, \ldots, x_n$ are the inputs
- $\sum$ is a summation
- w_{ab} is the weight connecting node a to node b
- f is an activation function
- θ is a bias

Questions

- How to initialize the weights and biases?
  Answer: Initialize them to some small random values.
- How to perform training?
  Answer:
  1. Let the network calculate the output with the given inputs (forward propagation)
  2. Calculate the error/loss function (i.e. the difference between the calculated outputs and the target outputs)
  3. Update the weights and biases between the hidden and output layer (backward propagation)
  4. Update the weights and biases between the input and hidden layer (backward propagation)
  5. Go back to step 1
- When to stop training?
  Answer:
  - After a fixed number of iterations through the loop.
  - Once the training error falls below some threshold.
  - Stop at a minimum of the error on the validation set.

Training can be very slow in networks with multiple hidden layers!

How to update the weights and biases?

- Assuming the activation function is the sigmoid function, $\sigma(x) = \frac{1}{1+e^{-x}}$. More details of the sigmoid function will be given later.
- The formula for updating weights and biases are derived by minimizing the error/loss function:

$$E = \frac{1}{2}\sum_{all\ k}(O_k - T_k)^2$$

  using gradient descent.

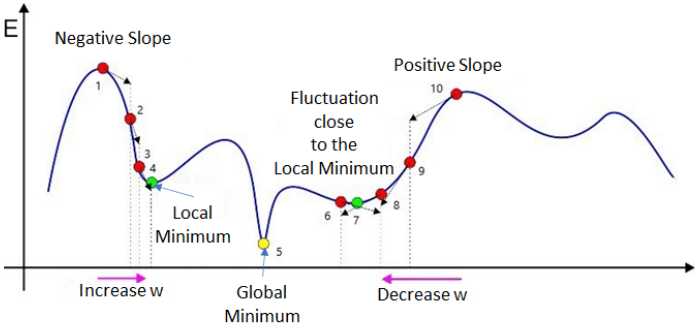
$$\delta_k = (O_k - T_k)O_k(1 - O_k) \qquad w_{jk} \leftarrow w_{jk} - \eta\delta_k O_j \qquad \theta_j \leftarrow \theta_j - \eta\delta_j$$

$$\delta_j = O_j(1 - O_j)\sum_{k\in K}\delta_k w_{jk} \qquad w_{ij} \leftarrow w_{ij} - \eta\delta_j O_i \qquad \theta_k \leftarrow \theta_k - \eta\delta_k$$

- O_i is the input in layer i
- O_j is the output of node in layer j
- O_k is the output of node in layer k
- T_k is the target output of node in layer k
- w_{ij} is the weight connecting node in layer i to node in layer j
- w_{jk} is the weight connecting node in layer j to node in layer k
- θ_j is the bias of node in layer j
- θ_k is the bias of node in layer k
- η is the learning rate.

Intuitive Idea of Gradient Descent

Consider the following error/loss function.



- It has many local minima.
- If we choose point #1 as the initial weight at certain learning rate, the slope at that point is negative. By moving towards the negative of the slope at that point, we converged to the minimum near point #4.
- But if we start at a different initial weight, we descend into a completely different local minimum.

Intuitive Idea of Gradient Descent (Cont'd)

- To minimize the function $E = \frac{1}{2}\sum_{all\ k}(O_k - T_k)^2$, we can follow the negative of the gradient (slope in 2D), and thus go in the direction of steepest descent. This is gradient descent.
- Formally, if we start at a point x_0 (x_0 can be a weight or a bias) and move a positive distance η in the direction of the negative gradient, then our new and improved x_1 (x_1 can be a weight or a bias) will look like this:

$$x_1 = x_0 - \eta \nabla E(x_0)$$

- More generally, we can write a formula for tuning x_n (x_n can be a weight or a bias) into x_{n+1} (x_{n+1} can be a weight or a bias):

$$x_{n+1} = x_n - \eta \nabla E(x_n)$$

- Starting from an initial guess x_0 (x_0 can be a weight or a bias), we keep improving little by little until we find a local minimum.
- This process may take thousands of iterations, so we typically implement gradient descent with a computer.

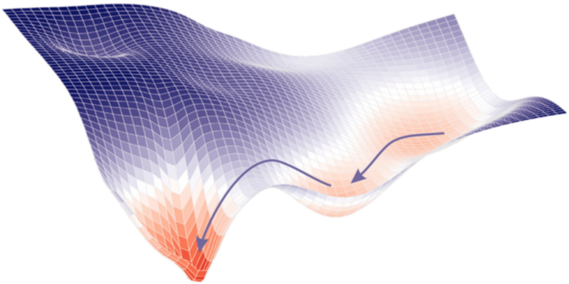
Gradient Descent

Question

Why is gradient descent used rather than directly to find a closed-form mathematics solution?

Answers:

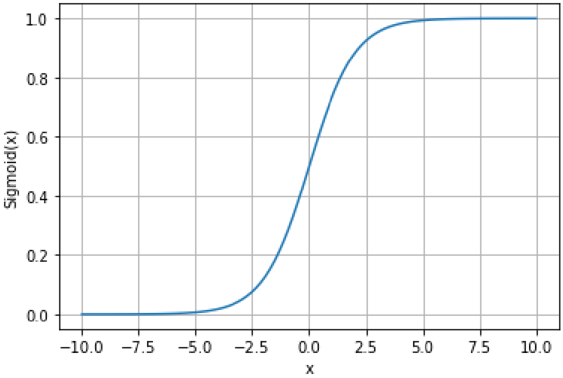
- For most non-linear regression problems, there is no closed-form solution.
- Even for those with a closed-form solution, gradient descent is computationally cheaper (faster) to find the solution.



Activation Function Again

- For multi-layer perceptron, the Sigmoid function is used as an activation function for neurons since it is continuous and differentiable (i.e. can be used to find the weights updating rules easily).

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



Learning Steps

1. Run the network forward with your input data to get the network output.
2. For each output node, compute

$$\delta_k = (O_k - T_k)O_k(1 - O_k)$$

3. For each hidden node, compute

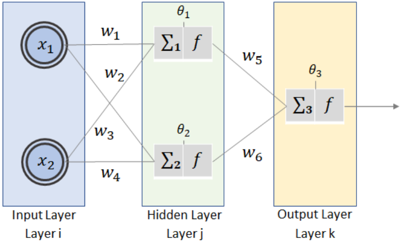
$$\delta_j = O_j(1 - O_j)\sum_{k \in K} \delta_k w_{jk}$$

4. Update the weights and biases as follows:

$$\begin{aligned} w_{jk} &\leftarrow w_{jk} - \eta\delta_k O_j \\ w_{ij} &\leftarrow w_{ij} - \eta\delta_j O_i \\ \theta_j &\leftarrow \theta_j - \eta\delta_j \\ \theta_k &\leftarrow \theta_k - \eta\delta_k \end{aligned}$$

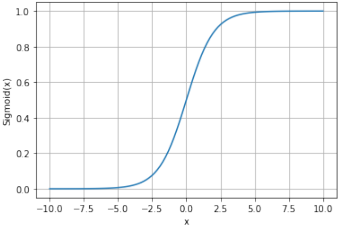
where η denotes the learning rate, typically, η is a value between 0 and 1.

Multi-Layer Perceptron Neural Network Example



- Suppose that we will work on a problem of XOR logical operation. The truth table of logical XOR is as follows.

x_1	x_2	T
0	0	0
0	1	1
1	0	1
1	1	0

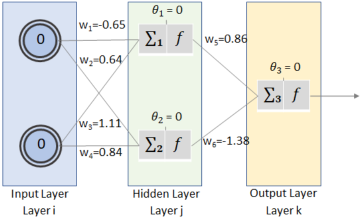


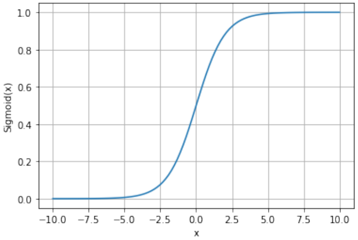
- Assume the weights are randomly generated, say $w_1 = -0.65$, $w_2 = 0.64$, $w_3 = 1.11$, $w_4 = 0.84$, $w_5 = 0.86$, and $w_6 = -1.38$. Also, assume the biases are randomly generated, say $\theta_1 = 0$, $\theta_2 = 0$, amd $\theta_3 = 0$. Finally, assume $\eta = 0.5$.
- Activation function is $f(x) = \frac{1}{1+e^{-x}}$

MLP Neural Network Example - Round 1 - Step 1, Forward Propagation

- Inputs: $x_1 = 0$, $x_2 = 0$
- Actual Output: $T = 0$
- Weights: $w_1 = -0.65$, $w_2 = 0.64$, $w_3 = 1.11$, $w_4 = 0.84$, $w_5 = 0.86$, and $w_6 = -1.38$
- Bias: $\theta_1 = 0$, $\theta_2 = 0$, $\theta_3 = 0$
- Calculations:
  - $\sum_1 = x_1 \cdot w_1 + x_2 \cdot w_2 = 0 \cdot (-0.65) + 0 \cdot (0.64) = 0$
    Output (O_{j1}): $f(\sum_1 + \theta_1) = \text{f}(0+0) = 0.5$
  - $\sum_2 = x_1 \cdot w_3 + x_2 \cdot w_4 = 0 \cdot 1.11 + 0 \cdot 0.84 = 0$
    Output (O_{j2}): $f(\sum_2 + \theta_2) = \text{f}(0+0) = 0.5$
  - $\sum_3 = O_{j1} \cdot w_5 + O_{j2} \cdot w_6 =$
    $0.5 \cdot (0.86) + 0.5 \cdot (-1.38) = -0.26$
    Output (O_k): $f(\sum_3 + \theta_3) = \text{f(-0.26+0)} = 0.435364$

$\eta = 0.5$

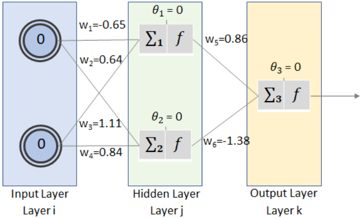


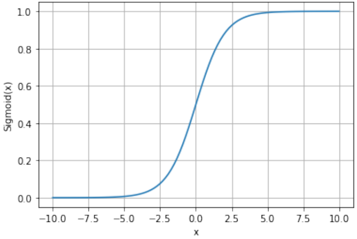


MLP Neural Network Example - Round 1 - Step 1, Backward Propagation

- Calculations:
  - Output (O_{j1}): $f(\sum_1 + \theta_1)$ = f(0+0) = 0.5
  - Output (O_{j2}): $f(\sum_2 + \theta_2)$ = f(0+0) = 0.5
  - Output (O_k): $f(\sum_3 + \theta_3)$ = f(0.26+0) = 0.435364
  - $\delta_k = (O_k - T_k)O_k(1- O_k) = (0.435364 - 0)(0.435364)(1-0.435364) = 0.107022$
  - New w_5 = Old w_5 - $\eta \delta_k O_{j1}$ = 0.86 - (0.5)(0.107022)(0.5) = 0.833245
  - New w_6 = Old w_6 - $\eta \delta_k O_{j2}$ = -1.38 - (0.5)(0.107022)(0.5) = -1.406756
  - New θ_3 = Old θ_3 - $\eta \delta_k$ = 0 - (0.5)(0.107022) = -0.053511
  - $\delta_{j1} = O_{j1}(1-O_{j1})\sum_{k \in K} \delta_k w_{jk} = 0.5(1-0.5)(0.107022)(0.86) = 0.023010$
  - $\delta_{j2} = O_{j2}(1-O_{j2})\sum_{k \in K} \delta_k w_{jk} = 0.5(1-0.5)(0.107022)(-1.38) = -0.036923$
  - New w_1 = Old w_1 - $\eta \delta_{j1} x_1$ = -0.65 - 0.5(0.023010)(0) = -0.65
  - New w_2 = Old w_2 - $\eta \delta_{j1} x_2$ = 0.64 - 0.5(0.023010)(0) = 0.64
  - New w_3 = Old w_3 - $\eta \delta_{j2} x_1$ = 1.11 - 0.5(-0.036923)(0) = 1.11
  - New w_4 = Old w_4 - $\eta \delta_{j2} x_2$ = 0.84 - 0.5(-0.036923)(0) = 0.84
  - New θ_1 = Old θ_1 - $\eta \delta_{j1}$ = 0 - (0.5)(0.023010) = -0.011505
  - New θ_2 = Old θ_2 - $\eta \delta_{j2}$ = 0 - (0.5)(-0.036923) = 0.018462

$\eta = 0.5$

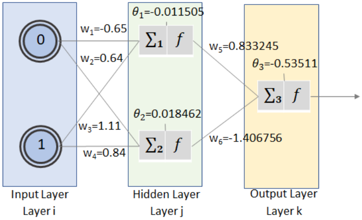


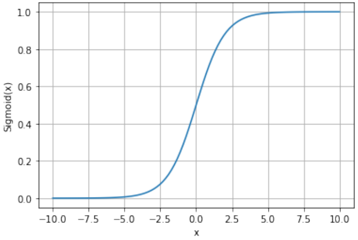


MLP Neural Network Example - Round 1 - Step 2, Forward Propagation

- Inputs: $x_1 = 0$, $x_2 = 1$
- Actual Output: $T = 1$
- Weights: $w_1 = -0.65$, $w_2 = 0.64$, $w_3 = 1.11$, $w_4 = 0.84$, $w_5 = 0.833245$, and $w_6 = -1.406756$
- Bias: $\theta_1 = -0.011505$, $\theta_2 = 0.018462$, $\theta_3 = -0.053511$
- Calculations:
  - $\sum_1 = x_1 \cdot w_1 + x_2 \cdot w_2 = 0 \cdot (-0.65) + 1 \cdot (0.64) = 0.64$
  - Output (O_{j1}): $f(\sum_1 + \theta_1)$ = f(0.64+(-0.011505)) = 0.652148
  - $\sum_2 = x_1 \cdot w_3 + x_2 \cdot w_4 = 0 \cdot 1.11 + 1 \cdot 0.84 = 0.84$
  - Output (O_{j2}): $f(\sum_2 + \theta_2)$ = f(0.84+(0.018462)) = 0.702339
  - $\sum_3 = O_{j1} \cdot w_5 + O_{j2} \cdot w_6 = 0.652148 \cdot 0.833245 + 0.702339 \cdot (-1.406756) =$ -0.444621
  - Output (O_k): $f(\sum_3 + \theta_3)$ = f(-0.444621-0.053511) = 0.377980

$\eta = 0.5$

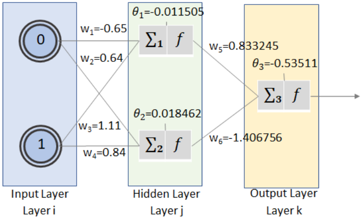


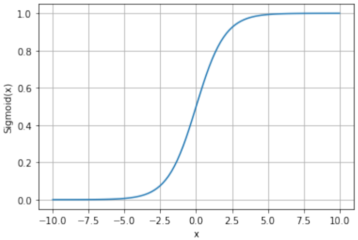


MLP Neural Network Example - Round 1 - Step 2, Backward Propagation

- Calculations:
  - Output (O_{j1}): $f(\sum_1 + \theta_1)$ = f(0.64+(-0.011505)) = 0.652148
  - Output (O_{j2}): $f(\sum_2 + \theta_2)$ = f(0.84+(0.018462)) = 0.702339
  - Output (O_k): $f(\sum_3 + \theta_3)$ = f(-0.444621-0.053511) = 0.377980
  - $\delta_k = (O_k - T_k)O_k(1 - O_k)$ = (0.377980 - 1)(0.377980)(1-0.377980) = -0.146244
  - New w_5 = Old w_5 - $\eta\delta_k O_{j1}$ = 0.833245 - (0.5)(-0.146244)(0.652148) = 0.880931
  - New w_6 = Old w_6 - $\eta\delta_k O_{j2}$ = -1.406756 - (0.5)(-0.146244)(0.702339) = -1.355400
  - New θ_3 = Old θ_3 - $\eta\delta_k$ = -0.053511 - (0.5)(-0.146244) = 0.019611
  - $\delta_{j1} = O_{j1}(1 - O_{j1})\sum_{k\in K}\delta_k w_{jk}$ = 0.652148(1-0.652148)(-0.146244)(0.833245) = -0.027463
  - $\delta_{j2} = O_{j2}(1 - O_{j2})\sum_{k\in K}\delta_k w_{jk}$ = 0.702339(1-0.702339)(-0.146244)(-1.406756) = 0.043010
  - New w_1 = Old w_1 - $\eta\delta_{j1}x_1$ = -0.65 - 0.5(-0.027463)(0) = -0.65
  - New w_2 = Old w_2 - $\eta\delta_{j1}x_2$ = 0.64 - 0.5(-0.027463)(1) = 0.653732
  - New w_3 = Old w_3 - $\eta\delta_{j2}x_1$ = 1.11 - 0.5(0.043010)(0) = 1.11
  - New w_4 = Old w_4 - $\eta\delta_{j2}x_2$ = 0.84 - 0.5(0.043010)(1) = 0.818495
  - New θ_1 = Old θ_1 - $\eta\delta_{j1}$ = -0.011505 - (0.5)(-0.027463) = 0.002227
  - New θ_2 = Old θ_2 - $\eta\delta_{j2}$ = 0.018462 - (0.5)(0.043010) = -0.003043

$\eta = 0.5$

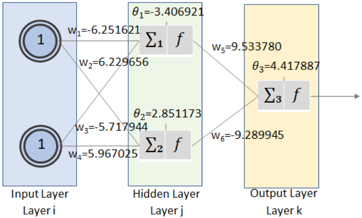


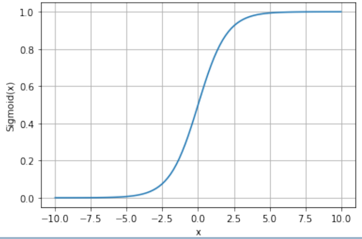


MLP Neural Network Example - Round 10000 - Step 4, Backward Propagation

- Calculations:
  - New $w_5 = 9.533777$
  - New $w_6 =$ -9.290053
  - New $w_1 =$ -6.251654
  - New $w_2 = 6.229623$
  - New $w_3 =$ -5.717901
  - New $w_4 = 5.967069$
  - New $\theta_3 = 4.417774$
  - New $\theta_1 =$ -3.406953
  - New $\theta_2 = 2.851217$

$\eta = 0.5$





- Round 10001, Forward Propagation
  - Input: $(x_1 = 0,\ x_2 = 0)$, Output: $y = 0.016973$
  - Input: $(x_1 = 0,\ x_2 = 1)$, Output: $y = 0.984139$
  - Input: $(x_1 = 1,\ x_2 = 0)$, Output: $y = 0.980513$
  - Input: $(x_1 = 1,\ x_2 = 1)$, Output: $y = 0.015179$

Multi-layer Perceptron Implementation from Scratch I

```
import numpy as np   # Import NumPy

class MultiLayerPerceptron:
   def __init__(self):
       """ Multi-layer perceptron initialization """
       self.wij = np.array([       # Weights between input and hidden layer
          [-0.65, 0.64],           # w1, w2
          [1.11, 0.84]             # w3, w4
       ])
       self.wjk = np.array([       # Weights between hidden and output layer
          [0.86],                  # w5
          [-1.38]                  # w6
       ])
       self.tj = np.array([        # Biases of nodes in the hidden layer
          [0.0],                   # Theta 1
          [0.0]                    # Theta 2
       ])
       self.tk = np.array([[0.0]]) # Bias of node in the output layer, Theta 3
       self.learning_rate = 0.5    # Eta
       self.max_round = 10000      # Number of rounds
```

Multi-layer Perceptron Implementation from Scratch II

```
def sigmoid(self, z, sig_cal=False):
    """ Sigmoid function and the calculation of z * (1-z) """
    if sig_cal: return 1 / (1 + np.exp(-z))  # If sig_cal is True, return sigmoid
    return z * (1-z) # If sig_cal is False, return z * (1-z)

def forward(self, x, predict=False):
    """ Forward propagation """
    # Get the training example as a column vector
    sample = x.reshape(len(x), 1)                                         # Shape (2,1)
    # Compute the hidden node outputs
    yj = self.sigmoid(self.wij.dot(sample) + self.tj, sig_cal=True)       # Shape (2,1)
    # Compute the output of node in the output layer
    yk = self.sigmoid(self.wjk.transpose().dot(yj) + self.tk, sig_cal=True) # Shape (1,1)
    # If predict is True, return the output of node in the layer node
    if predict: return yk
    # Otherwise, return (data sample, hidden node outputs, predicted output)
    return (sample, yj, yk)
```

Multi-layer Perceptron Implementation from Scratch III

```
def backpropagation(self, values, tk):
    Oi = values[0]  # Input sample
    Oj = values[1]  # Hidden node outputs
    Ok = values[2]  # Predicted output
    """ back propagation """
    # deltak = (Ok-tk)Ok(1-Ok)
    deltaK = np.multiply((Ok - tk), self.sigmoid(Ok))                  # Shape (1,1)
    # deltaj = Oj(1-Oj)(deltak)(Wjk)
    deltaJ = np.multiply(self.sigmoid(Oj), deltaK[0][0] * self.wjk)  # Shape (2,1)
    # wjk = wjk - eta(deltak)(Oj)
    self.wjk -= self.learning_rate * deltaK[0][0] * Oj               # Shape (2,1)
    # wij = wij - eta(deltaj)(Oi)
    s = self.learning_rate * deltaJ.dot(Oi.T)                        # Shape (2,2)
    # Alternative for the above: s = self.learning_rate * deltaJ * Oi.T
    self.wij -= s
    # thetaj = thetaj - eta(deltaj)
    self.tj -= self.learning_rate * deltaJ # Shape (2,1)
    # thetak = thetak - eta(deltak)
    self.tk -= self.learning_rate * deltaK # Shape (1,1)
```

Multi-layer Perceptron Implementation from Scratch IV

```
    def train(self, X, T):
        """ Training """
        for i in range(self.max_round):                # Train max_round number of rounds
            for j in range(m):                         # Use all the samples in the data set
                print(f'Iteration: {i+1} and {j+1}')
                values = self.forward(X[j])            # Forward propagation
                self.backpropagation(values, T[j]) # Back propagation

    def print(self):
        print(f'wij: {self.wij}')
        print(f'wjk: {self.wjk}')
        print(f'tj:  {self.tj}')
        print(f'tk:  {self.tk}')

m = 4 # Number of training samples
```

Multi-layer Perceptron Implementation from Scratch V

```
X = np.array([ # Input data
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

T = np.array([ # Target values
    [0],
    [1],
    [1],
    [0]
])

mlp = MultiLayerPerceptron()     # Create an object
mlp.train(X, T)                  # Call train function
mlp.print()                      # Print all the parameter values
for k in range(m):               # Testing
    Ok = mlp.forward(X[k],True)
    print(f'y{k}: {Ok}')
```

Note

- The following formulas only apply for the error/loss function
$$E = \frac{1}{2}\sum_{all\ k}(O_k - T_k)^2$$
and use the Sigmoid function as the activation.

$$\delta_k = (O_k - T_k)O_k(1 - O_k) \qquad w_{jk} \leftarrow w_{jk} - \eta\delta_k O_j \qquad \theta_j \leftarrow \theta_j - \eta\delta_j$$

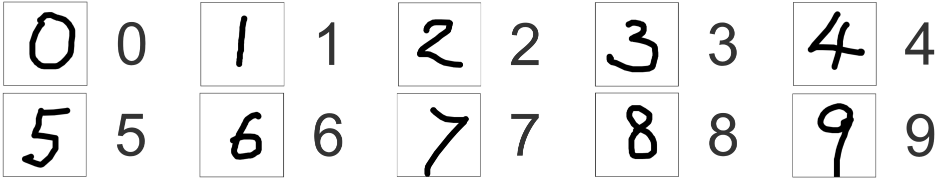
$$\delta_j = O_j(1 - O_j)\sum_{k \in K}\delta_k w_{jk} \qquad w_{ij} \leftarrow w_{ij} - \eta\delta_j O_i \qquad \theta_k \leftarrow \theta_k - \eta\delta_k$$

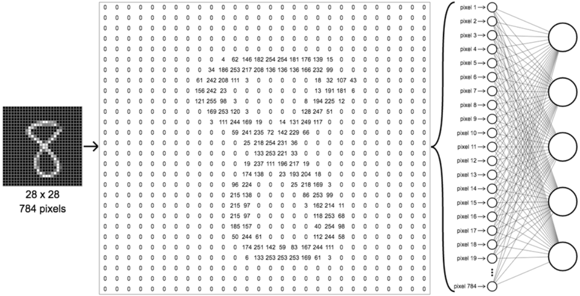
Question and Answer

- Question: What if a different error/loss function or activation function is used in an MLP?
- Answer: We need to differentiate the error/loss function, i.e., something similar to what we demonstrated in the supplementary notes.

Handwritten Digits Recognition using MLP

We will build a MLP Artificial Neural Network to recognize/classify handwritten digits.

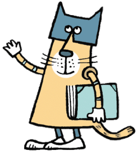
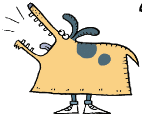




Terminologies

- Training data
  - The data our model learn from
- Testing data
  - The data is kept secret from the model until after it has been trained. Testing data is used to evaluate our model.
- Loss function
  - A function used to quantify how accurate a model's predictions were.
- Optimization algorithm
  - It controls exactly how the weights are adjusted during training.



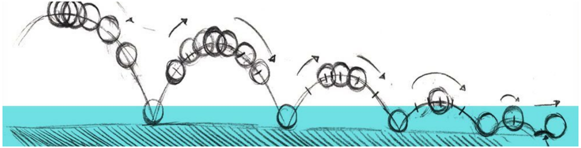



Dataset

- We use the Modified National Institute of Standards and Technology (MNIST) dataset.
- This dataset contains two sets of samples:
  - Training data: 60000 28 pixel $\times$28 pixel images of handwritten digits from 0 to 9.
  - Testing data: 10000 28 pixel $\times$ 28 pixel images.

Procedures

1. Import the required libraries and define a global variable
2. Load the data
3. Explore the data
4. Build the model
5. Compile the model
6. Train the model
7. Evaluate the model accuracy
8. Save the model
9. Use the model
10. Plotting the confusion matrix



1. Import the Required Libraries and Define a Global Variable

```
import numpy as np                              # Import numpy library
import matplotlib.pyplot as plt                 # Import matplot library
import seaborn as sn                            # Import seaborn library
import pandas as pd                             # Import pandas library
import math                                     # Import math library
import datetime                                 # Import datetime library
from keras.datasets import mnist                # Import MNIST dataset
from keras.models import Sequential             # Import Sequential class
from keras.layers import Dense, Flatten         # Import Dense, Flatten class
from keras import regularizers                  # Import regularizers
from tensorflow.keras.optimizers import Adam    # Import Adam optimizer
# Import spare categorical crossentroy loss
from keras.metrics import sparse_categorical_crossentropy
from keras.callbacks import TensorBoard         # Import TensorBoard class
from keras.models import load_model             # Import load_model method
from tensorflow.keras.utils import plot_model   # Import plot_model method
from tensorflow.math import confusion_matrix    # Import confusion_matrix method
from tensorflow.keras import activations        # Import activations module
epochs = 1                                      # Number of epochs to train the model
```

2. Load the Data

```
# x_train is a NumPy array of grayscale image data with shapes (60000, 28, 28)
# y_train is a NumPy array of digit labels (in range 0-9) with shape (60000,)
# x_test is a NumPy array of grayscale image data with shapes (10000, 28, 28)
# y_test is a NumPy array of digit labels (in range 0-9) with shape (10000,)
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Print the data shape
print('x_train:', x_train.shape)
print('y_train:', y_train.shape)
print('x_test:', x_test.shape)
print('y_test:', y_test.shape)
```

```
x_train: (60000, 28, 28)
y_train: (60000,)
x_test: (10000, 28, 28)
y_test: (10000,)
```

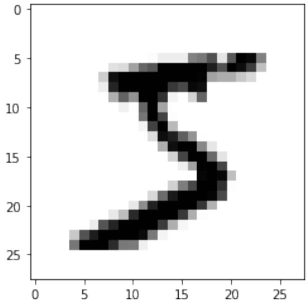
3. Explore the Data

```
# Show the pixel values (from 0 255) of
# the first image
pd.DataFrame(x_train[0])
```

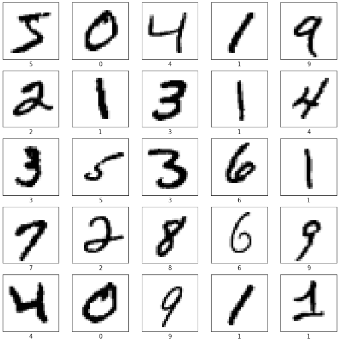
	0	1	2	3	4	5	6	7	8	9	...	18	19	20	21	22	23	24	25	26	27
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	...	175	26	166	255	247	127	0	0	0	0
6	0	0	0	0	0	0	0	0	30	36	...	225	172	253	242	195	64	0	0	0	0
7	0	0	0	0	0	0	0	49	238	253	...	93	82	82	56	39	0	0	0	0	0
8	0	0	0	0	0	0	0	18	219	253	...	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	80	156	...	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	14	...	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	...	25	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	...	150	27	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	...	253	187	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	...	253	249	64	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	...	253	207	2	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	...	250	182	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	...	78	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	23	66	...	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	18	171	219	253	...	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	55	172	226	253	253	253	...	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	136	253	253	253	212	135	...	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

28 rows × 28 columns

```
# Show the image in binary form
plt.imshow(x_train[0], cmap=plt.cm.binary)
plt.show()
```

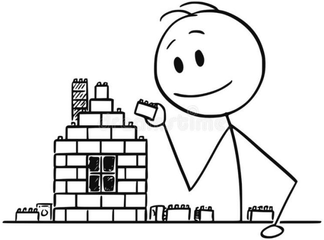


```python
numbers_to_display = 25     # Display 25 images
# Compute number of images per row
num_cells = math.ceil(math.sqrt(numbers_to_display))
plt.figure(figsize=(10,10)) # Each image is in 10x10 inches
# Show all the images
for i in range(numbers_to_display):
    # number of rows, number of columns, index (start from 1)
    plt.subplot(num_cells, num_cells, i+1)
    plt.xticks([])              # Remove all xticks
    plt.yticks([])              # Remove all yticks
    plt.grid(False)             # No grid lines
    # Display data as a binary image
    plt.imshow(x_train[i], cmap=plt.cm.binary)
    # Show training image labels
    plt.xlabel(y_train[i])
plt.show() # Show the figure
```



4. Build the Model

- Instead of building the model from scratch, we will use the software library, Keras, instead.
- Layers
  - Layer 1: Flatten layer that will flatten 2D image into 1D
  - Layer 2: Hidden Dense layer 1 with 128 neurons and ReLU activation
  - Layer 3: Hidden Dense layer 2 with 128 neurons and ReLU activation
  - Layer 4: Output Dense layer with 10 Softmax outputs. This layer represents the guess, i.e., the 0th output represents the probability that the input digit is 0, the 1st output represents a probability that the input digit is 1, etc.



```
model = Sequential() # Create a Sequential object
# Input layer
# Add a flatten layer to convert the image data to a single column
model.add(Flatten(input_shape=x_train.shape[1:]))
# Hidden layer 1
# Add a dense layer (fully-connected layer) and use ReLU activation function.
# This layer uses L2 loss, computed as l2 * reduce_sum(square(x)), where l2 is 0.002
model.add(Dense(units=128, activation='relu',
                kernel_regularizer=regularizers.l2(0.002)
))
# Hidden layer 2
# Add a dense layer (fully-connected layer) and use ReLU activation function.
# This layer uses L2 loss, computed as l2 * reduce_sum(square(x)), where l2 is 0.002
model.add(Dense(units=128, activation=activations.relu,
                kernel_regularizer=regularizers.l2(0.002)
))
# Output layer
# Add a dense layer (fully-connected layer) and use softmax activation function.
model.add(Dense(units=10, activation='softmax'))

# We apply kernel_regularizer to penalize the weights which are very large causing the
# network to overfit, after applying kernel_regularizer the weights will become smaller.
```

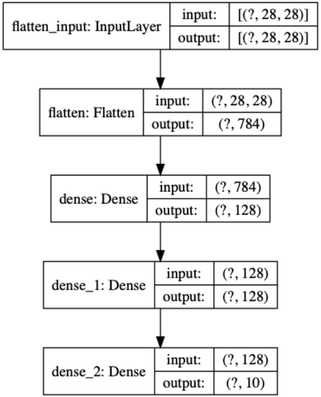
- Print model summary

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dense_1 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 118,282
Trainable params: 118,282
Non-trainable params: 0
_________________________________________________________________
```

- Plot the model

```
plot_model(model,
           show_shapes=True,
           show_layer_names=True
)
```



5. Compile the Model

```
# Create an Adam optimizer by creating an object
# Set learning rate to 0.001
# Note: Optimizers are Classes or methods used to change the attributes
# of your machine/deep learning model such as weights and learning rate
# in order to reduce the losses.
adam_optimizer = Adam(learning_rate=0.001)

# Compile the model, i.e., configures the model for training
# Use crossentropy loss function since there are two or more label classes.
# Use adam algorithm (a stochastic gradient descent method)
# Use accuracy as metric, i.e., report on accuracy
model.compile(
    optimizer=adam_optimizer,
    loss=sparse_categorical_crossentropy,
    metrics=['accuracy']
)
```

6. Train the Model

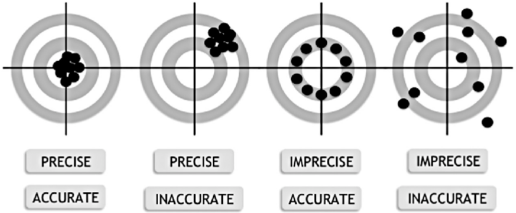
```
# TensorBoard is a visualization tool, enabling us to track metrics like
# loss and accuracy, visualize the model graph, view histograms of weights, etc.
# Create TensorBoard object to track experiment metrics like loss and
# accuracy, visualizing the model graph, etc.
log_dir=".logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

# log_dir: the path of the directory where to save the log files
# histogram_freq: frequency (in epochs) at which to compute activation and weight
# histograms for the layers of the model
tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

# Fit the model, i.e., train the model
# Specify training data and labels, number of epochs to train the model,
# validation data, i.e., data on which to evaluate the loss
# Write TensorBoard logs after every batch of training to monitor our metrices
training_history = model.fit(x_train, y_train, epochs=epochs,
                             validation_data=(x_test, y_test),
                             callbacks=[tensorboard_callback]
)
```

7. Evaluate Model Accuracy

```python
# Evaluate the model
# Specify testing data and labels
validation_loss, validation_accuracy = model.evaluate(x_test, y_test)
# Print loss and accuracy
print('Validation loss: ', validation_loss)
print('Validation accuracy: ', validation_accuracy)
```

Output

```
Validation loss:  0.2004156953573227
Validation accuracy:  0.9646
```



8. Save the Model

- Save the entire model to an HDFS (Hadoop Distributed File System) file.
- The .h5 extension of the file indicates that the model should be saved in Keras format as an HDFS file.

```
model_name = 'digits_recognition_mlp.h5'
model.save(model_name, save_format='h5')

loaded_model = load_model(model_name)
```

9. Use the Model

- To use the model, we call predict() function

```python
# Use the model to do prediction by specifying the image(s).
# Get back a NumPy array of prediction
predictions = loaded_model.predict([x_test])
print('predictions:', predictions.shape)
```

Output

```
predictions: (10000, 10)
```

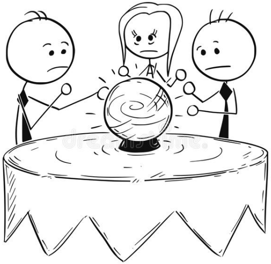
- Each prediction consists of 10 probabilities (one for each number from 0 to 9). The digit with the highest probability is chosen as that would be a digit that our model is most confident with.

```
# Predictions in form of one-hot vectors (arrays of probabilities).
pd.DataFrame(predictions)
```

	0	1	2	3	4	5	6	7	8	9
0	7.774682e-07	1.361266e-05	9.182121e-05	1.480533e-04	3.271606e-08	2.764984e-06	5.903113e-11	9.996371e-01	1.666906e-06	1.042360e-04
1	1.198126e-03	1.047034e-04	9.888538e-01	3.167473e-03	2.532723e-08	8.854911e-04	6.828848e-04	5.048555e-06	5.102141e-03	2.777219e-07
2	8.876222e-07	9.985157e-01	7.702865e-05	2.815677e-05	5.406159e-04	2.707353e-05	2.035172e-04	9.576474e-05	5.053744e-04	5.977653e-06
3	9.990014e-01	4.625264e-06	5.582303e-04	5.484722e-06	3.299095e-05	2.761683e-05	1.418936e-04	1.374896e-04	1.264711e-06	8.899846e-05
4	1.575061e-04	3.707617e-06	9.205778e-06	3.638557e-07	9.973990e-01	1.538193e-06	3.079933e-05	4.155232e-05	4.639028e-06	2.351647e-03
...	...	...	...	...	...	...	...	...	...	...
9995	1.657425e-07	8.666945e-04	9.987835e-01	2.244577e-04	6.904386e-12	1.850143e-07	4.015289e-08	9.534077e-05	2.960863e-05	4.335253e-10
9996	6.585806e-09	6.717554e-06	6.165197e-06	9.982822e-01	2.519031e-09	1.577783e-03	5.583775e-11	2.066899e-06	1.257137e-05	1.125286e-04
9997	3.056851e-08	6.843247e-06	3.161353e-09	6.484316e-08	9.989114e-01	2.373860e-07	1.930965e-08	1.753431e-05	5.452521e-06	1.058474e-03
9998	7.249156e-06	5.103301e-07	2.712475e-08	1.025373e-04	5.019490e-08	9.996431e-01	9.364716e-05	1.444746e-07	1.520906e-04	6.703385e-07
9999	2.355737e-06	4.141651e-07	4.489176e-06	1.321389e-07	2.956528e-05	3.940167e-05	9.999231e-01	7.314535e-08	2.270459e-07	1.044889e-07

10000 rows × 10 columns

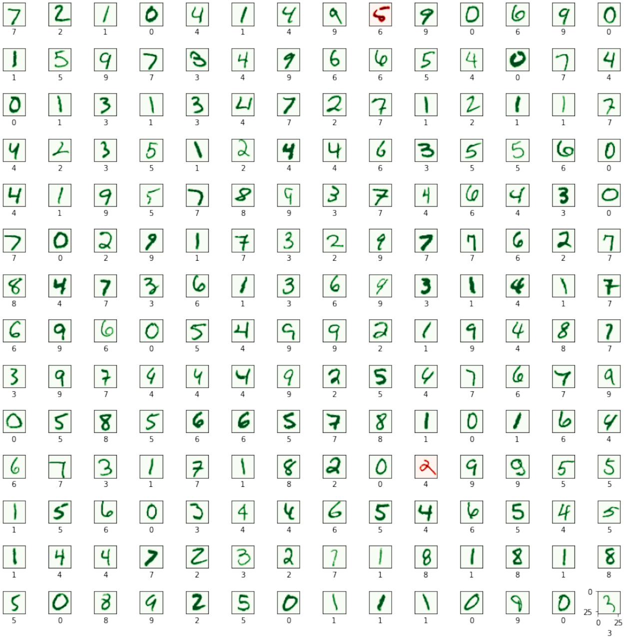
```python
# Let's extract predictions with highest probabilites and
# detect what digits have been actually recognized.
prediction_results = np.argmax(predictions, axis=1)

pd.DataFrame(prediction_results)
```



	0
0	7
1	2
2	1
3	0
4	4
...	...
9995	2
9996	3
9997	4
9998	5
9999	6

10000 rows × 1 columns

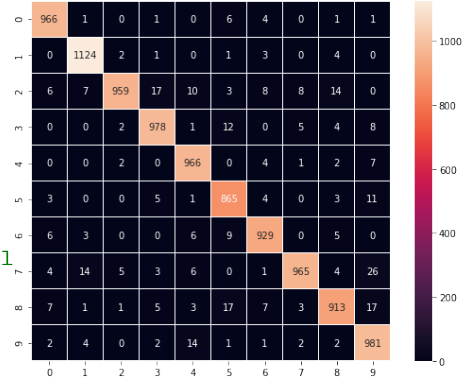
```python
numbers_to_display = 196     # Display 196 images
# Compute number of images per row
num_cells = math.ceil(math.sqrt(numbers_to_display))
plt.figure(figsize=(10, 10)) # Each image is in size 10x10 inches

# Show all the images
for i in range(numbers_to_display):
    # Number of rows, number of columns, index (start from 1)
    plt.subplot(num_cells, num_cells, i + 1)
    plt.xticks([])           # Remove all xticks
    plt.yticks([])           # Remove all yticks
    plt.grid(False)          # No grid lines
    # Check if the prediction is correct. If so, display in green. Otherwise in red.
    color_map = 'Greens' if prediction_results[i] == y_test[i] else 'Reds'
    plt.imshow(x_test[i], cmap=color_map) # Display data as a color image
    plt.xlabel(prediction_results[i])     # Show predicted image labels

# Adjust the height of the padding between subplots to 1
# Adjust the width of the padding between subplots to 0.5
plt.subplots_adjust(hspace=1, wspace=0.5)
plt.show() # Show the figure
```


7 2 1 0 4 1 4 9 6 9 0 6 9 0
1 5 9 7 3 4 9 6 6 5 4 0 7 4
0 1 3 1 3 4 7 2 7 1 2 1 1 7
4 2 3 5 1 2 4 4 6 3 5 5 6 0
4 1 9 5 7 8 9 3 7 4 6 4 3 0
7 0 2 9 1 7 3 2 9 7 7 6 2 7
8 4 7 3 6 1 3 6 9 3 1 4 1 7
6 9 6 0 5 4 9 9 2 1 9 4 8 7
3 9 7 4 4 4 9 2 5 4 7 6 7 9
0 5 8 5 6 6 5 7 8 1 0 1 6 4
6 7 3 1 7 1 8 2 0 4 9 9 5 5
1 5 6 0 3 4 4 6 5 4 6 5 4 5
1 4 4 7 2 3 2 7 1 8 1 8 1 8
5 0 8 9 2 5 0 1 1 1 0 9 0 3
0
25
0 25

10. Plotting a confusion matrix

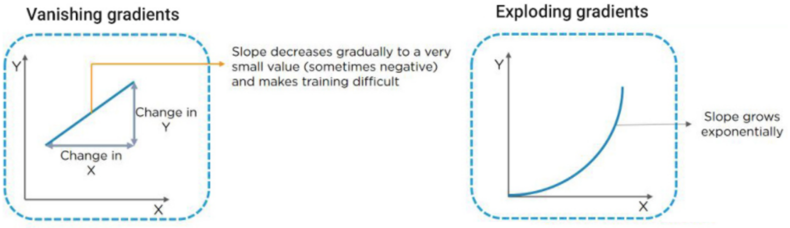
The confusion matrix shows what numbers are recognized well by the model and what numbers the model usually confuses to recognize correctly.

```
# Compute confusion matrix to evaluate the accuracy of a classification
# by creating a confusion_matrix object.
# Specify true labels and prediction results
cm = confusion_matrix(y_test, prediction_results)
# Each image is in size 9x9 inches
f, ax = plt.subplots(figsize=(9, 9))
# Draw heat map to show the magnitude in color
sn.heatmap(
    cm,              # data
    annot=True,      # True (write the data in each cell)
    linewidths=.5,   # Width of line that divides each cell
    fmt="d",         # Format of the data, decimal
    square=True,     # Make cell as square-shaped
    ax=ax            # Draw it on ax
)
plt.show()           # Show the figure
```



Problem: Vanishing Gradient and Exploding Gradient

- One of the problems of training a neural network (especially with many hidden layers) is the vanishing and exploding gradient.
- When we train a neural network, the gradient or the slope can get very big or very small or exponentially small, which makes training difficult.
- As a consequence, the weights are not updated anymore, and learning stalls.

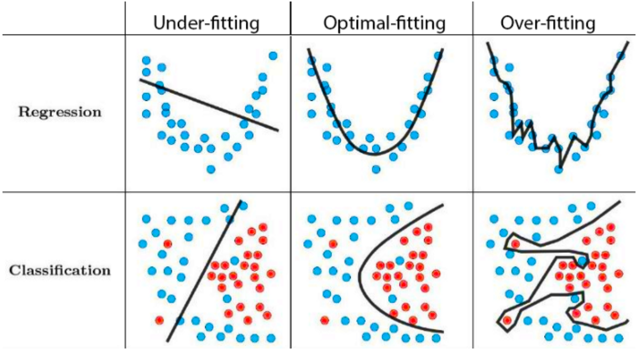


How to Know Whether Model is Suffering from Vanishing/Exploding Gradient?

- For vanishing gradient
  - The parameters of the higher layers vary dramatically, whereas the parameters of the lower levels do not change significantly for vanishing (or not at all).
  - During training, the model weights may become zero.
  - The model learns slowly, and after a few cycles, the training may become stagnant.
- For exploding gradient
  - The model parameters are growing exponentially.
  - During training, the model weights may become NaN.
  - The model goes through an avalanche learning process.

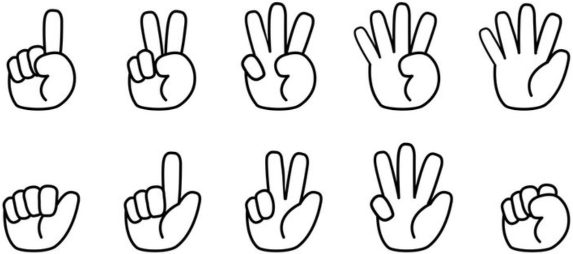
Problem: Overfitting and Underfitting

- Overfitting
  - It refers to a model that models the training data too well. It happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on the new data.
- Underfitting
  - It refers to a model that can neither model the training data nor generalize to new data.



How Many Layers and Number of Neurons in Each of These Layers?

- The input layer
  - Number of layers = 1
  - Number of neurons = Number of features (i.e., columns) in our data (e.g., for XOR, the number of neurons in the input layer is 2)
- The output layer
  - Number of layers = 1
  - Number of neurons = Mostly 1, unless softmax is used (Just like the handwritten digits example)

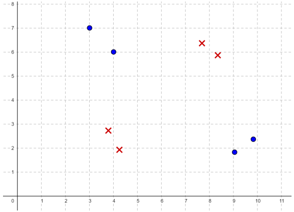


How Many Layers and Number of Neurons in Each of These Layers?

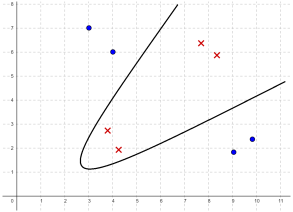
- The hidden layers
  - Number of layers
    - If our data is linearly separable, NO hidden layer at all.
    - If data is less complex and has few dimensions or features, neural networks with 1 to 2 hidden layers would work.
    - If data has large dimensions or features, 3 to 5 hidden layers can be used to get an optimum solution.
  - Number of neurons:
    - The number of hidden neurons should be between the size of the input layer and the output layer.
    - The most appropriate number of hidden neurons is
$$\sqrt{\mathit{input\ layer\ nodes} \times \mathit{output\ layer\ nodes}}$$
    - The number of hidden neurons should keep decreasing in subsequent layers to get closer to pattern and feature extraction and identify the target class.

The above algorithms are only a general use case, and they can be moulded according to the use case. Sometimes the number of nodes in hidden layers can also increase in subsequent layers, and the number of hidden layers can also be more than the ideal case. This depends on the use case and problem statement that we are dealing with.

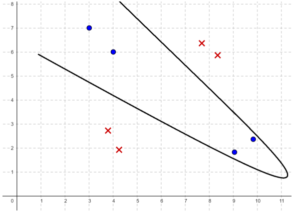
Example



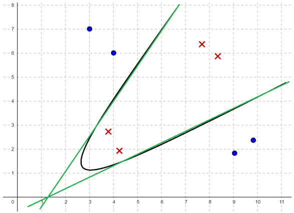
Each sample has two inputs and one output presenting the class label



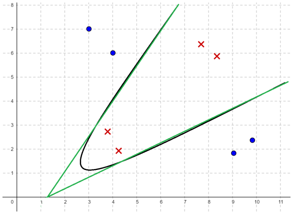
One possible decision boundary separates the data correctly



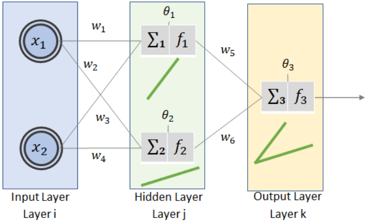
Another possible decision boundary separates the data correctly



Two lines are required to represent the decision boundary, which tells us the first hidden layer will have two hidden neurons

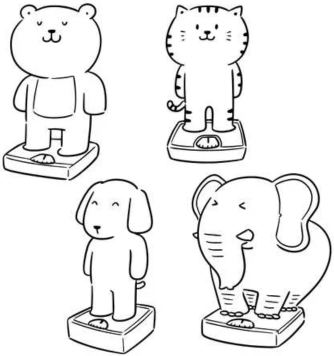


The two lines are to be connected by another neuron, which is in the output layer



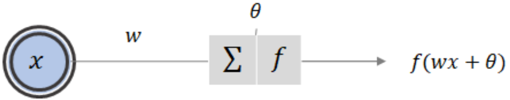
The Role of Weights

- Weights are the real values associated with each feature which tells the importance of that feature in predicting the final value.



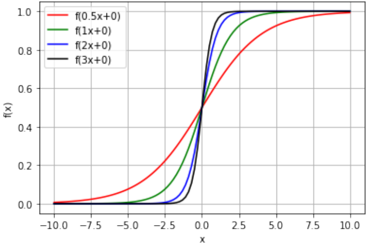
The Effect for the Change of Weights

- Suppose we have the following perceptron:



- Let's get the output functions by setting w to 1, 2, 3, θ to 0, and using sigmoid activation function. Now, plot the output functions and figure out the use of weights.

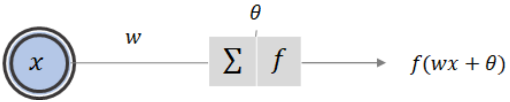
$$y = f(0.5x + 0)$$
$$y = f(1x + 0)$$
$$y = f(2x + 0)$$
$$y = f(3x + 0)$$



According to the example, we can see that weights control the steepness of the activation function.

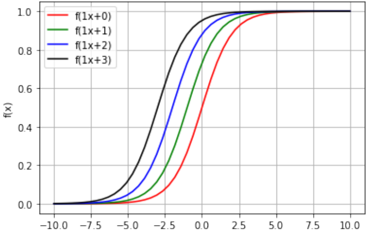
What is the Role of Biases?

- Suppose we have the following perceptron:



- Now, let's get another set of output functions by setting w to 1, θ to 0, 1, 2, and 3, and using the sigmoid activation function. Plot the output functions and try to figure out the use of biases.

$$y = f(1x + 0)$$
$$y = f(1x + 1)$$
$$y = f(1x + 2)$$
$$y = f(1x + 3)$$



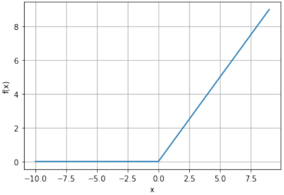
According to the example, we can see that bias is used for shifting the activation function towards the left or right.

Activation Functions

- The choice of activation function in the hidden layer has a large impact on the capability and performance of the neural network.
- An activation function in a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network.
- All hidden layers typically use the same activation function. The output layer will typically use a different activation function from the hidden layers and is dependent upon the type of prediction required by the model.
- Typically, a differentiable non-linear activation function is used in the hidden layers of a neural network. This allows the model to learn more complex functions.
- Three most commonly used activation functions in hidden layers:
  - Rectified Linear Activation (ReLU)
  - Logistic (Sigmoid)
  - Hyperbolic Tangent (Tanh)

ReLU Hidden Layer Activation Function

- ReLU function is defined as $$f(x) = max(0, x)$$

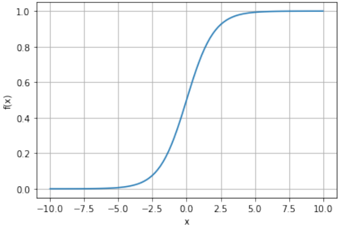


- It is the most common function used for hidden layers.
- It is simple to implement and effectively overcome the limitations of other previously popular activation functions, such as Sigmoid and Tanh. Specifically, it is less susceptible to vanishing gradients.
- When using the ReLU function for hidden layers, it is good to use "He Normal" or "He Uniform" weight initialization and scale input data to the range of 0-1 before training.

```
initializer = tf.keras.initializers.HeNormal()
# initializer = tf.keras.initializers.HeUniform()
layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)
```

Sigmoid Hidden Layer Activation Function

- Sigmoid function is defined as $$f(x) = \frac{1}{1+e^{-x}}$$



- The sigmoid activation function is also called the logistic function.
- When using the sigmoid function for hidden layers, it is good to use a "Glorot Normal" or "Glorot Uniform" weight initialization and scale input data to the range 0-1 before training.

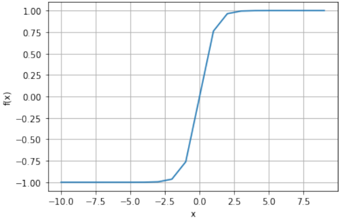
  Note: Glorot initializer is also called Xavier initializer.

```
initializer = tf.keras.initializers.GlorotNormal()
# initializer = tf.keras.initializers.GlorotUniform()
layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)
```

Tanh Hidden Layer Activation Function

- Tanh function is defined as

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



- When using the Tanh function for hidden layers, it is good to use a "Glorot Normal" or "Glorot Uniform" weight initialization and scale input data to the range -1 to 1 before training.

  Note: Glorot initializer is also called Xavier initializer.

```
initializer = tf.keras.initializers.GlorotNormal()
# initializer = tf.keras.initializers.GlorotUniform()
layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)
```

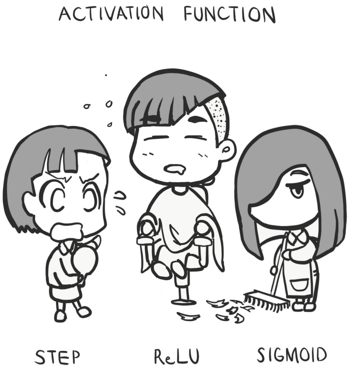
How to Choose a Hidden Layer Activation Function?

- Both the Sigmoid and Tanh functions can make the model more susceptible to problems during training via the so-called vanishing gradients problem.
- The activation function used in hidden layers is typically chosen based on the type of neural network architecture.
- Modern neural network models with common architectures, such as MLP and CNN (will be mentioned soon), will use the ReLU activation function or extensions.
- Recurrent networks (a type of neural network that has at least one loop) still commonly use Tanh or sigmoid activation functions, or even both.
- Summary

Neural Network	Commonly Used Activation Function
Multi-layer Perceptron (MLP)	ReLU activation function
Convolutional Neural Network (CNN)	ReLU activation function
Recurrent Neural Network (RNN)	Tanh and/or Sigmoid activation function

Activation Function for Output Layers

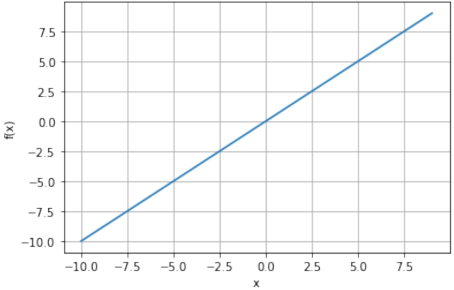
- The output layer is the layer in a neural network model that directly outputs a prediction.
- There are three commonly used activation functions for use in the output layer.
    - Linear
    - Logistic (Sigmoid)
    - Softmax



Linear Output Activation Function

- The linear output activation function is defined as:

$$f(x) = x$$



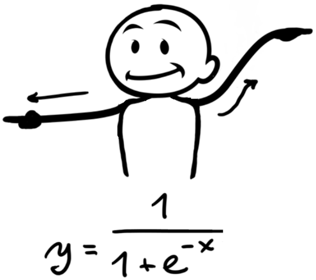
- The linear activation function is also called "identity" (multiplied by 1.0) or "no activation".
- Target values used to train a model with a linear activation function in the output layer are typically scaled before modeling using normalization or standardization transforms.

Sigmoid Output Activation Function

- Recall, the Sigmoid activation function is defined as:

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

- Target labels used to train a model with a Sigmoid activation function in the output layer will have the values 0 or 1.



Softmax Output Activation Function

- Softmax is a mathematical function that converts an array (or vector) of numbers into an array (or vector) of probabilities.
- The softmax function is defined as:

$$\mathbf{x} = [x_0, x_1, \ldots, x_{n-1}]$$
$$f(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{n-1} e^{x_j}}$$

- The softmax function is used as the activation function for multi-class classification problems where class membership is required on more than two class labels.

How to Choose an Output Activation Function

- We choose the activation function for your output layer based on the prediction problem we are solving.
- If a problem is a regression problem, we should use a linear activation function.
- If a problem is a classification problem, then there are three main types of classification problems, and each may use a different activation function.
  1. Binary classification: One node, sigmoid activation.
  2. Multiclass classification: One node per class, softmax activation
  3. Multilabel classification: One node per class, sigmoid activation

Note: Multiclass classification makes the assumption that each sample is assigned to one and only one label. Multilabel classification assigns to each sample a set of target labels.

When to Use Multi-layer Perceptrons?

- Multi-layer perceptrons are suitable for classification prediction problems where inputs are assigned a class or label.
- They are also suitable for regression prediction problems where a real-valued quantity is predicted given a set of inputs. Data is often provided in a tabular format, such as we would see in a CSV file or a spreadsheet.

Practice Problem

- Given a multilayer perceptron with two inputs x_1, x_2, one hidden unit and one output unit. Both the hidden unit and output use sigmoid activation function. Altogether, the network has 3 weights, w_1, w_2, w_3, and 2 biases, θ_1, θ_2.
- All weights are initialized with 0.1, and all the biases are initialized with -0.1.
- Use sigmoid as the activation function for all units, i.e.

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

- Let the training set be as follows:

x1	x2	T
1	0	1
0	1	0

Determine the weights after the first epoch two iterations of the backpropagation algorithm, given a learning rate of $\eta = 0.3$.

Multilayer Perceptron - Round 1 - Step 1, Forward Propagation

- Inputs: $x_1 = 1, x_2 = 0$
- Actual Output: $T = 1$
- Weights: $w_1 = 0.1,\ w_2 = 0.1,\ w_3 = 0.1$
- Biases: $\theta_1 = -0.1,\ \theta_2 = -0.1.$
- Calculations:
  - $\sum_1 = x_1 \cdot w_1 + x_2 \cdot w_2 = 1 \cdot (0.1) + 0 \cdot (0.1) = 0.1$
    Output (O_j): $f(\sum_1 + \theta_1) = f(0.1 - 0.1) = 0.5$
  - $\sum_2 = O_j \cdot w_3 = 0.5 \cdot (0.1) = 0.05$
    Output (O_k): $f(\sum_2 + \theta_2) = f(0.05 - 0.1) = 0.487503$

Multilayer Perceptron - Round 1 - Step 1, Backward Propagation

- Calculations:
  - Output (O_j): $f(\sum_1 + \theta_1) = f(0.1 - 0.1) = 0.5$
  - Output (O_k): $f(\sum_2 + \theta_2) = f(0.05 - 0.1) = 0.487503$
  - $\delta_k = (O_k - T_k)O_k(1 - O_k) = (0.487503 - 1)(0.487503)(1 - 0.487503) = -0.128044$
  - New w_3 = Old w_3 - $\eta\delta_k O_j = 0.1 - 0.3(-0.128044)(0.5) = 0.119207$
  - New θ_2 = Old $\theta_2 - \eta\delta_k = -0.1 - (0.3)(-0.128044) = -0.061587$
  - $\delta_j = O_j(1 - O_j)\delta_k w_{jk} = 0.5(1 - 0.5)(-0.128044)(0.1) = -0.003201$
  - New w_1 = Old $w_1 - \eta\delta_j x_1 = 0.1 - (0.3)(-0.003201)(1) = 0.100960$
  - New w_2 = Old $w_2 - \eta\delta_j x_2 = 0.1 - (0.3)(-0.003201)(0) = 0.1$
  - New θ_1 = Old $\theta_1 - \eta\delta_j = -0.1 - (0.3)(-0.003201) = -0.099040$

Multilayer Perceptron - Round 1 - Step 2, Forward Propagation

- Inputs: $x_1 = 0, x_2 = 1$
- Actual Output: $T = 0$
- Weights: $w_1 = 0.100960,\ w_2 = 0.1,\ w_3 = 0.119207$
- Biases: $\theta_1 = -0.099040,\ \theta_2 = -0.061587.$
- Calculations:
  - $\sum_1 = x_1 \cdot w_1 + x_2 \cdot w_2 = 0 \cdot (0.100960) + 1 \cdot (0.1) = 0.1$
    Output (O_j): $f(\sum_1 + \theta_1) = f(0.1 - 0.099040) = 0.50024$
  - $\sum_2 = O_j \cdot w_3 = 0.50024 \cdot (0.119207) = 0.059632$
    Output (O_k): $f(\sum_2 + \theta_2) = f(0.059632 - 0.061587) = 0.499511$

Multilayer Perceptron - Round 1 - Step 2, Backward Propagation

- Calculations:
  - Output (O_j): $f(\sum_1 + \theta_1) = f(0.1 - 0.099040) = 0.50024$
  - Output (O_k): $f(\sum_2 + \theta_2) = f(0.059632 - 0.061587) = 0.499511$
  - $\delta_k = (O_k - T_k)O_k(1 - O_k) = (0.499511 - 0)(0.499511)(1 - 0.499511) = 0.124878$
  - New w_3 = Old w_3 - $\eta \delta_k O_j = 0.119207 - 0.3(0.124878)(0.50024) = 0.100466$
  - New θ_2 = Old $\theta_2 - \eta \delta_k = -0.061587 - (0.3)(0.124878) = -0.09905$
  - $\delta_j = O_j(1 - O_j)\delta_k w_{jk} = 0.50024(1 - 0.50024)(0.124878)(0.119207) = 0.003722$
  - New w_1 = Old $w_1 - \eta \delta_j x_1 = 0.100960 - (0.3)(0.003722)(0) = 0.100960$
  - New w_2 = Old $w_2 - \eta \delta_j x_2 = 0.1 - (0.3)(0.003722)(1) = 0.098883$
  - New θ_1 = Old $\theta_1 - \eta \delta_j = -0.099040 - (0.3)(0.003722) = -0.100157$

That's all!

Any questions?