The purpose of this assignment is to get familiar with the basic concept of deep networks for classification tasks. You will start by implementing a simple perceptron and will continue to implement some of the most well-known architectures for the image classification task.

Learning the logic operators

The single perceptron is the simplest unit of an artificial neural network. Although a single-layer perceptron works as a simple linear classifier, the extension to multilayer perceptrons would increase the discrimination power of the model that would be helpful to classify the data with nonlinear distributions. In brief, the architecture of each perceptron (neuron) is as follow:

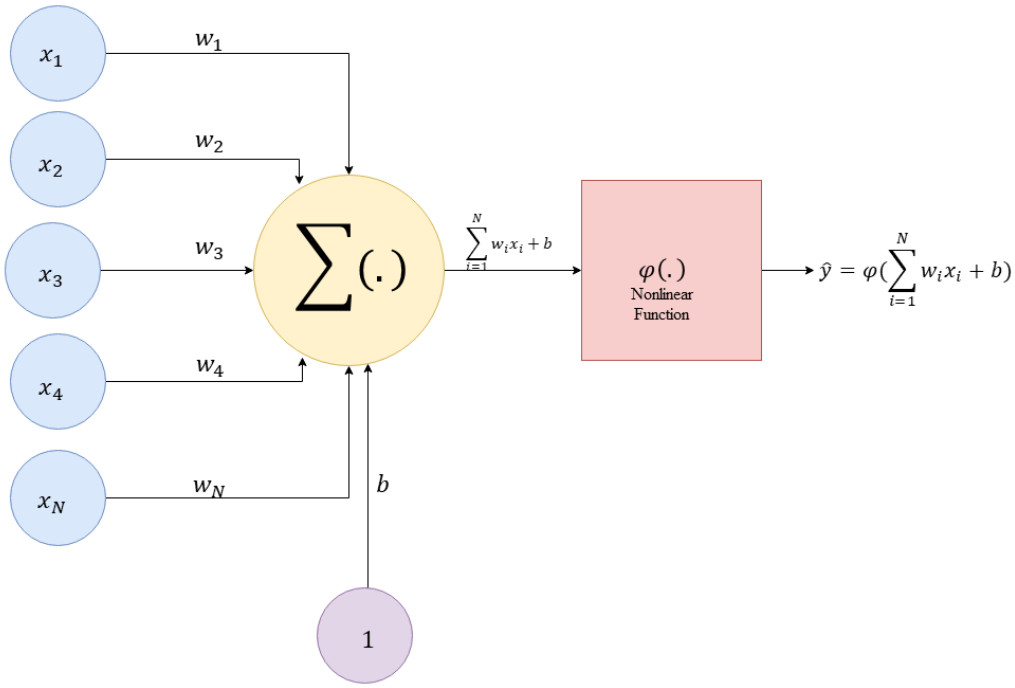

Fig.1 Perceptron architecture.

Where x_i refers to i dimensional input data, w_i refers to weights, and b represents the bias. In this tutorial, a Sigmoid activation function (nonlinear function), $\varphi(x) = \frac{1}{1+e^{-x}}$, will be used.

If you connect two perceptrons sequentially in a way that the output of the first neuron is set as the input of the second neuron, then, you will have a 2-layers neural network:

$$\hat{y} = \varphi\big(w_2.\big(\varphi(w_1 x + b_1)\big) + b_2\big)$$

The only variables of this formula are weights and biases (W, b). In general, these values will be initiated randomly, and during a proper iterative learning procedure of the network, they would converge to their optimum values. Each iteration contains two steps: during the *feedforward* step, the estimated model output $\hat{y}$ will be calculated, and during the *backpropagation*, based on the difference (error) between the target (y) and estimated output ($\hat{y}$), model variables will be updated.

The following forgue shows an example of a 2-layers neural network:

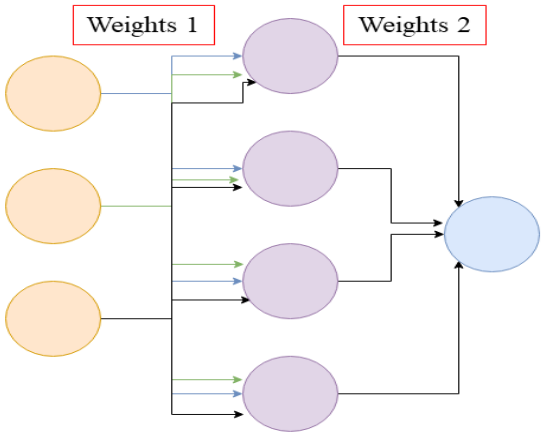

Fig2. Two layers network.

As illustrated in Fig.2, the dimension of the input layers is 3, and there are 4 neurons in the second layer. Now, we are going to implement a 2-layers model step by step. To update the model variables (weights and biases) by minimizing the differences between the target and the estimated output, an objective function should be defined. In this exercise, the Sum of Square Error (SSD) will be used as the objective function. Please note, loss/cost function are the other names of the objective function.

$$Loss(y, \hat{y}) = SSD = \sum_{i=1}^{n} (y - \hat{y})^2$$

The goal is to employ the gradient descent algorithm to update the model variables which will result in minimizing the loss function:

$$\frac{\partial Loss(y, \hat{y})}{\partial w} = \frac{\frac{\partial Loss(y, \hat{y})}{\partial \hat{y}} * \partial \hat{y}}{\frac{\partial z}{\partial w}} * \partial z = 2(y - \hat{y}) * z(1 - z) * x$$

$$z = wx + b$$

For simplicity, we begin by applying this basic model to solve the “AND” logic operators:

Table1. AND operator

x_1	x_2	**y**
0	0	0
0	1	0
1	0	0
1	1	1

If you consider x_1 and x_2 as the two dimensions of the input into the AND space, **y** will be the target, or visually:

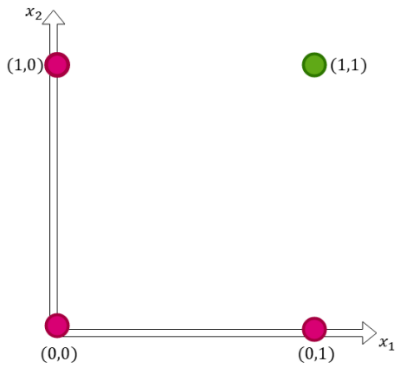

Fig.3 Illustration of AND logic operator in 2D space.

This means the dimension of the input data is 2, and based on the target (red and green circles), the task will be a binary classification between these two classes by drawing a line. The simple step by step implementation is:

```
# Logic Operator

import numpy as np
import matplotlib.pyplot as plt

# Sigmoid function
def sigmoid(x):
    return 1.0/(1+ np.exp(-x))

# derivative of Sigmoid function for backprop.
def sigmoid_derivative(x):
    return x * (1.0 - x)

class NeuralNetwork:
    def __init__(self, x, y, N):
        self.input    = x
        self.neuron   = N
        self.weights1  =  np.random.rand(self.input.shape[1],  self.neuron)  #  X  dimension  input
connected to N neurons
        self.weights2 = np.random.rand(self.neuron, 1)                  # N neurons connected to
output
        self.y        = y
        self.output   = np.zeros(self.y.shape)               # instantiating the output

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.output = sigmoid(np.dot(self.layer1, self.weights2))

    def backprop(self):
        # Chain rule to calculate derivative of the loss function with respect to weights2 and
weights1
        d_weights2 = np.dot(self.layer1.T,
                            (2*(self.y - self.output)
                            * sigmoid_derivative(self.output)))

        d_weights1 = np.dot(self.input.T,
                            (np.dot(2*(self.y - self.output)
                            * sigmoid_derivative(self.output),
                            self.weights2.T) * sigmoid_derivative(self.layer1)))
```

```
        # weights updating
        self.weights1 += d_weights1
        self.weights2 += d_weights2


iterations = 10
n_unit = 1

if __name__ == "__main__":

    Input = np.array([[0,0,1],
                      [0,1,1],
                      [1,0,1],
                      [1,1,1]])

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

    model = NeuralNetwork(Input, Target, n_unit)

    SSD = []
    for i in range(iterations):
        model.feedforward()
        model.backprop()
        errors = (Target - model.output)**2
        SSD.append(np.sum(errors))            # Objective(loss) function


    Itr = np.linspace(1,len(SSD),len(SSD))
    plt.plot(Itr, SSD)
    plt.xlabel('Iterations')
    plt.ylabel('SSD')

    print("The target values are:", Target)
    print("The predicted values are:", model.output)
```

Task1) Run the above code and interpret the results. Please note the output of the model is the predicted of class lables of each of the four points. If you run the code several times, will you observe the same results? Why? Keep the parameter "n_unit=1" and increase the number of iterations starting from 10, 50, 100, 500, 2000, and compare the loss values. What can you conclude from increasing the number of iterations? Now, with a fixed value of "iterations = 1000", increase the parameter "n_unit" to 2, 5, 10 and interpret the results.

Task2) Repeat task1 for XOR logic operator. For fixed values of parameters (iterations=2000, and n_unit=1), which of the AND or XOR operators has lower loss values? Why? Increase the number of neurons in the hidden layer (n_unit) to 2, 5, 10, 50. Does increasing the number of neurons improve the results? Why?

Table2. XOR operator

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

The previous model will be completed by adding the learning rate parameter. The same 2-layers model is now implemented more efficiently in Tensorflow platform:

```
# Logic operator with Tensorflow Keras
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD

Input = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
Target = np.array([[0],[1],[1],[0]], "float32")
n_unit = 50

model = Sequential()
model.add(Dense(n_unit, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='mean_squared_error',
              optimizer = SGD(),
              metrics=['binary_accuracy'])

model.fit(Input, Target, epochs = 5000, verbose=0)

print("The predicted class labels are:", model.predict(Input))
```

Task3) In the above code, change the parameter "n_unit" as 1, 2, 4, 16, 25, 50, and interpret the observed results.

Multilayer Perceptron for Image Classification

In general, reading the image data and their corresponding labels(targets) is the first step in any image-based deep learning tasks. For instance, for image classification tasks, each image belongs to a specific class; therefore, it is important to read the images along with their corresponding class labels properly. Although during this course, you will be familiar with more efficient data loading approaches, for this exercise, we will use a simple method.

For this task, you will use skin cancer images as the dataset which belong to two groups named as "Melanoma" and "Nevi". For simplicity, the class labels are adopted from the image names. If the image names contain a pattern of "Mel," it will be marked as class 0, and if it entails a pattern of "Nev" it will be marked as class 1.

Task) Review the following data loader code and find out how it works. Run it to load the training and test data.

```
# Data Loader
import os
import numpy as np
from random import shuffle
from skimage.io import imread
from skimage.transform import resize



def gen_labels(im_name, pat1, pat2):
    '''

    Parameters
    ----------
    im_name : Str
        The image file name.
    pat1 : Str
```

```
        A string pattern in the filename for 1st class, e.g "Mel"
    pat2 : Str
        A string pattern in the filename 2nd class, e.g, "Nev"

    Returns
    -------
    Label : Numpy array
        Class label of the filename name based on its pattern.
    '''
    if pat1 in im_name:
        label = np.array([0])
    elif pat2 in im_name:
        label = np.array([1])
    return label


def get_data(data_path, data_list, img_h, img_w):
    """

    Parameters
    ----------
    train_data_path : Str
        Path to the data directory
    train_list : List
        A list containing the name of the images.
    img_h : Int
        image height to be resized to.
    img_w : Int
        image width to be resized to.

    Returns
    -------
    img_labels : Nested List
        A nested list containing the loaded images along with their
        correcponding labels.
    """
    img_labels = []
    for item in enumerate(data_list):
        img = imread(os.path.join(data_path, item[1]), as_gray = True) # "as_grey"
        img = resize(img, (img_h, img_w), anti_aliasing = True).astype('float32')
        img_labels.append([np.array(img), gen_labels(item[1], 'Mel', 'Nev')])


        if item[0] % 100 == 0:
             print('Reading: {0}/{1}  of train images'.format(item[0], len(data_list)))


    shuffle(img_labels)
    return img_labels

def get_data_arrays(nested_list, img_h, img_w):
    """
    Parameters
    ----------
    nested_list : nested list
        nested list of image arrays with corresponding class labels.
    img_h : Int
        Image height.
    img_w : Int
        Image width.

    Returns
    -------
    img_arrays : Numpy array
        4D Array with the size of (n_data,img_h,img_w, 1)
    label_arrays : Numpy array
```

```
        1D array with the size (n_data).

    """
    img_arrays = np.zeros((len(nested_list), img_h, img_w), dtype = np.float32)
    label_arrays = np.zeros((len(nested_list)), dtype = np.int32)
    for ind in range(len(nested_list)):
        img_arrays[ind] = nested_list[ind][0]
        label_arrays[ind] = nested_list[ind][1]
    img_arrays = np.expand_dims(img_arrays, axis =3)
    return img_arrays, label_arrays



def get_train_test_arrays(train_data_path, test_data_path, train_list,
                          test_list, img_h, img_w):

    """
    Get the directory to the train and test sets, the files names and
    the size of the image and return the image and label arrays for
    train and test sets.
    """


    train_data = get_data(train_data_path, train_list, img_h, img_w)
    test_data = get_data(test_data_path, test_list, img_h, img_w)


    train_img, train_label =  get_data_arrays(train_data, img_h, img_w)
    test_img, test_label = get_data_arrays(test_data, img_h, img_w)
    del(train_data)
    del(test_data)
    return train_img, test_img, train_label, test_label



img_w, img_h = 128, 128      # Setting the width and heights of the images.
data_path = '/DL_course_data/Lab1/Skin/'     # Path to data root with two subdirs.


train_data_path = os.path.join(data_path, 'train')
test_data_path = os.path.join(data_path, 'test')

train_list = os.listdir(train_data_path)
test_list = os.listdir(test_data_path)


x_train, x_test, y_train, y_test = get_train_test_arrays(
        train_data_path, test_data_path,
        train_list, test_list, img_h, img_w)
```

As you are familiar with the functionality of the perceptrons, now, you will develop a multilayer perceptron (MLP) by adding a number of perceptron in sequential layers. Then the developed MLP should be examined to classify the two classes of skin cancer images.

Task4) Develop a 4-layers MLP. If you call the number of neurons in the first fully-connected layer as “base_dense”, this 4-layers MLP should contain “base_dense”, “base_dense//2”, and “base_dense//4” as the number of neurons in the first 3 layers respectively. The activation function of all those neurons should set as “Relu”. However, in the last layer (4th layer), choose a proper number of neurons as well as activation function(s) that fit the binary classification task. Develop your model as a function, like:

```
def model(img_width, img_height, img_ch, base_dense):
    """
    Functional API model.
    name the last layer as "out"; e.g., out = ....

    """
    input_size = (img_width, img_height, img_ch)
    inputs_layer = Input(shape=input_size, name='input_layer')
    # TODO
    clf = Model(inputs=inputs_layer, outputs=out)
    clf.summary()


    return clf
```

Then compile and train the model for the following parameters: `n_epochs = 150, Batch_Size = 16, base_dense = 64, LR = 0.0001` by using "clf_hist = clf.fit(...)"

Task4) The values of loss and accuracy metrics are saved within the variable "clf_hist". You can use the following code to visualize the loss curves:

```
import matplotlib.pyplot as plt
import numpy as np
plt.figure(figsize=(4, 4))
plt.title("Learning curve")
plt.plot(clf_hist.history["loss"], label="loss")
plt.plot(clf_hist.history["val_loss"], label="val_loss")
plt.plot( np.argmin(clf_hist.history["val_loss"]),
          np.min(clf_hist.history["val_loss"]),
          marker="x", color="r", label="best model")

plt.xlabel("Epochs")
plt.ylabel("Loss Value")
plt.legend(); ht, img_ch, base_dense)          # Model Instantiating
clf.compile(loss = 'binary_crossentropy', optimizer = Adam(lr = LR), metrics = ['accuracy'])
```

How do you interpret the observed values of loss and accuracy values? Is the number of epochs enough to make a good decision about model performance? For the same number of epochs, reduce the learning rate parameter to 0.1 and interpret the results.
Now increase the number of epochs to 150 with LR=0.0001. Does this model have enough capacity to yield acceptable results? Increase the "base_dense" parameter to 256 and compare the results with the case of "base_dense=64". Is increasing the model capacity helpful to improve the model performance? Why?

Convolutional Neural Network

In this lab, the concept of Convolutional Neural Networks (CNNs) starts by implementing the LeNet model. The architecture of this model contains two convolutional layers, two max-pooling layers, followed by two dense layers as well.

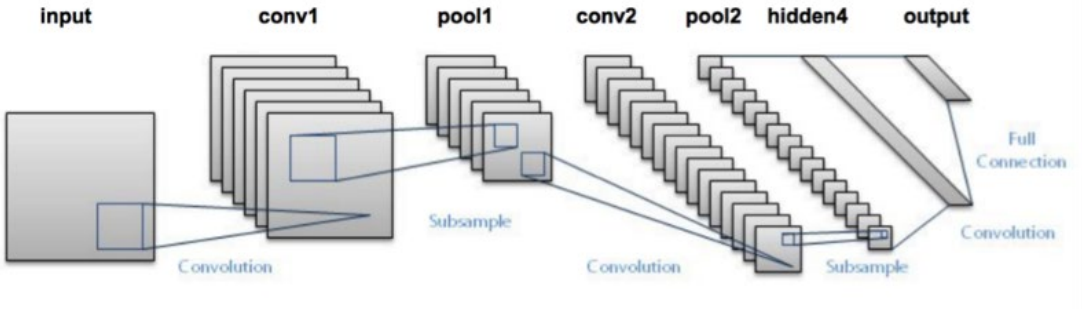

Fig.4 LeNeT architecture.

The following code is an implementation of the LeNet model in sequential format. Use the framework of task4 to load the data, compile the model, and fit the complied model with the loaded skin cancer dataset. Then, answer the questions in the following tasks. (Please note you have to import all the required layers similar to the last exercise.)

```
def model(img_ch, img_width, img_height):
    model = Sequential()
    model.add(Conv2D(base, kernel_size = (3, 3), activation='relu',
                     strides=1, padding='same',
                     input_shape = (img_width, img_height, img_ch)))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(base*2, kernel_size = (3, 3), activation='relu',
                     strides=1, padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())
    model.add(Dense(base*2, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()

    return model
```

Task5A) Set the following parameters: # of epochs = 20; batch size = 8; number of feature maps at the first convolutional layer = 32 and learning rate = 0.00001 and then run the experiment. What are the value training and validation accuracies? What can you infer from the learning curves? Is it reasonable to make a decision based on this set-up of the parameters?

Task5B) Leave all the parameters from the previous task unchanged except for the n_epoch = 200. Compare the results with task 5A.

Task5C) Keep all parameters from the last step except for the LR = 0.0001 . Run the experiment with new LR and interpret the results. How do you evaluate the generalization power of the model? What are the values of training and validation accuracies?

Task5D) What is the role of the first two convolutional layers?

Task5E) What is the role of the last two dense layers?

Task5F) What is the major difference between the LeNet and MLP?

Task5G) Look at the last layer of the network. How should we choose the number of neurons and the activation function of the last layer?

Good luck.