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 image classification task.

Learning the logic operators

The single layer perceptron is the simplest artificial neural network. Although the perceptron is a simple linear classifier, extension to multi-layer perceptron will be useful for more complicated nonlinear dataset. In brief, the architecture of each perceptron (neuron) is as follow:

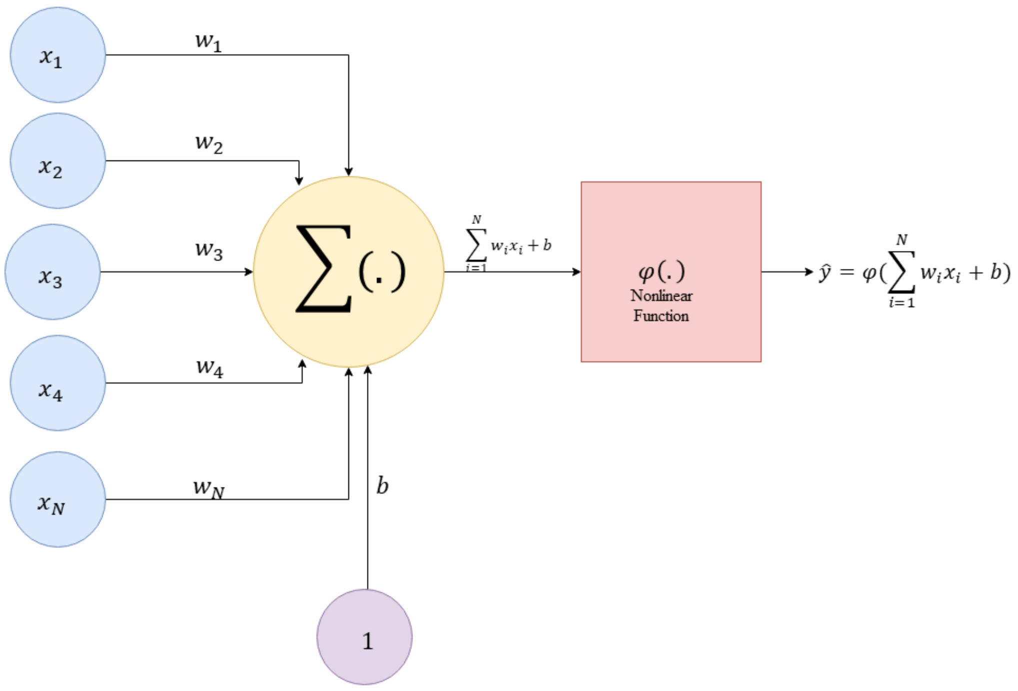

Fig.1 Perceptron architecture.

Where x_i refers to i dimensional input data, w_i refers to weights and b represent 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 perceptron sequentially in a way that the output of the first neuron is set the input of the second neuron, then, you will have a 2-layers neural network:

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

Weights and biases (W, b) are the only variables of this formula. In general, these values will be initiated randomly, and during a proper iterative training of the network, they are supposed to 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. In the following, an example of a 2-layers network will is presented:

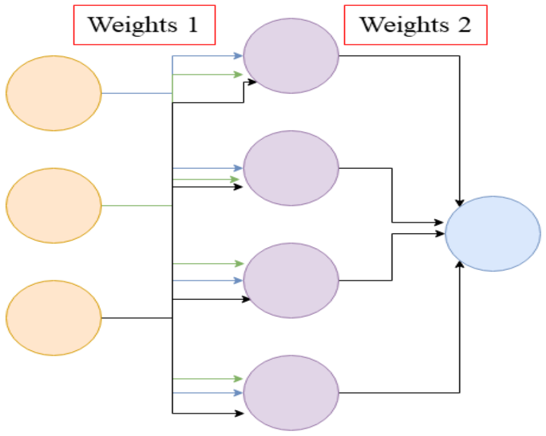

Fig2. Two layers network.

As illustrated in Fig.1 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. For this purpose, Sum of Square Error (SSD) will be set as objective (loss) 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 results in minimizing the loss function:

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

$$z = wx + b$$

Now, you will apply this basic model on the AND logic operators:

Table1. AND operator

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

Or graphically:

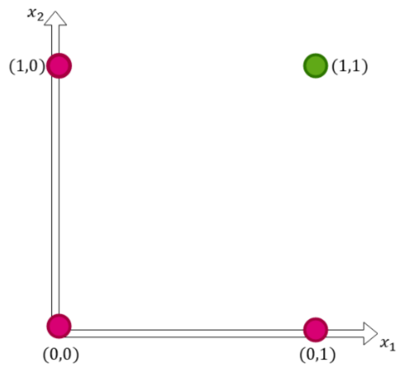

Fig.3 Illustration of AND logic operator in 2D space.

This means the dimension of the input data is 2 and the task will be a binary classification between the red and green circles by simply drawing a line. The simple step by step implementation in Python will include the following parameters:
Random initial weights, and variable iteration.

```
# 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 = 1000
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. If you run the code several times, will you observe the same results? Why? Increase the number of iterations starting from 10, 50, 100, 500, 2000 and compare the loss values.

Task2) Repeat task2 for XOR operator. For the same number of iterations, which operator has lower loss values? Why? Increase the number of neurons in hidden layer (n_unit) from 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 effectively 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 n_unit as 1, 2, 4, 16, 25, 50 and interpret the observed resutls.

Multilayer Perceptron for Image Classification

In general, reading the image data and the corresponding labels 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 very important to read the images along with their corresponding class labels properly. Although during this course, you will be familiar with advanced data loading, for this exercise we will use a simple approach:

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


img_w, img_h = 100, 100      # Setting the width and heights of the images
data_path = '/Lab1/Skin/'           # Path to data root. Inside this path,
                                                        #two subfolder are placed one for train
data and one for test data.


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)

# Assigning labels two images; those images contains pattern1 in their filenames
# will be labeled as class 0 and those with pattern2 will be labeled as class 1.
def gen_labels(im_name, pat1, pat2):
        if pat1 in im_name:
            Label = np.array([0])
        elif pat2 in im_name:
            Label = np.array([1])
        return Label

# reading and resizing the training images with their corresponding labels
def train_data(train_data_path, train_list):
    train_img = []
    for i in range(len(train_list)):
        image_name = train_list[i]
        img = imread(os.path.join(train_data_path, image_name), as_grey=True)
        img = resize(img, (img_h, img_w), anti_aliasing = True).astype('float32')
        train_img.append([np.array(img), gen_labels(image_name, 'Mel', 'Nev')])

        if i % 200 == 0:
             print('Reading: {0}/{1}  of train images'.format(i, len(train_list)))

    shuffle(train_img)
    return train_img

# reading and resizing the testing images with their corresponding labels
def test_data(test_data_path, test_list):
    test_img = []
    for i in range(len(test_list)):
        image_name = test_list[i]
        img = imread(os.path.join(test_data_path, image_name), as_grey=True)
        img = resize(img, (img_h, img_w), anti_aliasing = True).astype('float32')
        test_img.append([np.array(img), gen_labels(image_name, 'Mel', 'Nev')])

        if i % 100 == 0:
            print('Reading: {0}/{1} of test images'.format(i, len(test_list)))

    shuffle(test_img)
    return test_img

# Instantiating images and labels for the model.
def get_train_test_data(train_data_path, test_data_path, train_list, test_list):

    Train_data = train_data(train_data_path, train_list)
    Test_data = test_data(test_data_path, test_list)

    Train_Img = np.zeros((len(train_list), img_h, img_w), dtype = np.float32)
    Test_Img = np.zeros((len(test_list), img_h, img_w), dtype = np.float32)

    Train_Label = np.zeros((len(train_list)), dtype = np.int32)
    Test_Label = np.zeros((len(test_list)), dtype = np.int32)

    for i in range(len(train_list)):
        Train_Img[i] = Train_data[i][0]
        Train_Label[i] = Train_data[i][1]

    Train_Img = np.expand_dims(Train_Img, axis = 3)

    for j in range(len(test_list)):
        Test_Img[j] = Test_data[j][0]
        Test_Label[j] = Test_data[j][1]

    Test_Img = np.expand_dims(Test_Img, axis = 3)

    return Train_Img, Test_Img, Train_Label, Test_Label

x_train, x_test, y_train, y_test = get_train_test_data(
        train_data_path, test_data_path,
        train_list, test_list)
```

For this task, you will use skin cancer images 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.

As you are familiar with the function of perceptron, now, you will develop a multilayer perceptron (MLP) by adding a number of perceptron in sequential layers. Then by using the skin cancer images, you will try to classify them into two groups.

```
# MLP Example; Complete the code
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

???

plt.figure(figsize=(4, 4))
plt.title("Learning curve")
plt.plot(History.history["loss"], label="loss")
plt.plot(History.history["val_loss"], label="val_loss")
plt.plot( np.argmin(History.history["val_loss"]),
         np.min(History.history["val_loss"]),
         marker="x", color="r", label="best model")

plt.xlabel("Epochs")
plt.ylabel("Loss Value")
plt.legend();
```

Task4) Develop a 4-Layers MLP with the following setting: 4 fully-connected layers with Base, Base//2 and Base//4 as the number of neurons at the first 3 layers and Relu activation function. For the last layer choose a proper number of neuron as well as activation function that fits the binary classification task. Write the model as function like:

```
def model(img_ch, img_width, img_height):
…
return model
```

Then compile and train the model for the following parameters: `n_epochs = 60, Batch_Size = 16, Base = 64, LR = 0.0001`

Fit the developed model to the provided sample code to visualize the learning curve.

How do you interpret the observed values of loss and accuracy? Set the learning rate parameters as 0.01, 0.001 and 0.0001 and try again. Which learning rate produced more stable results? Which learning rated yielded over-fitted results?

Convolutional Neural Network

To begin with Convolutional Neural Networks (CNNs), you will do some experiments with LeNet model. The model architecture includes two convolutional layers, two max-pooling layers, and two dense layers.

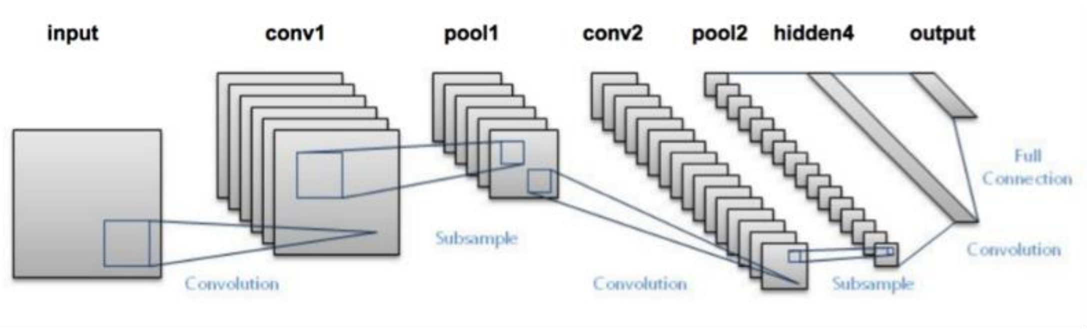

Fig.4 LeNeT architecture.

The following code is an implementation of the LeNet model. Use the framework of exercise 4 and answer the questions in the following tasks. (Please note to import all the required layers similar to 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 = 0.0001 and then run the experiment. What are the training and validation accuracies? What can you infer from the learning curves?

Task5B) Leave all the parameters from previous step unchanged except the # of epochs = 80. What is the major difference between the observed results and what you achieved from previous step?

Task5C) Reduce the number of feature maps at the first convolutional layer to 8 and let the model run for 80 epochs. Compare your results with the steps 5B. How can you interpret the changes?

Task5D) Keep the setting of step 5C but increase the learning rate and set it as 0.01; What do you conclude from task5? How should you evaluate the generalization of your model?

Task5E) What is the role of the first two convolutional layers?

Task5F) What is the role of the last two dense layers?

Task5G) What is the major difference between the LeNet and MLP?

Task5H) Look at the last layer of the network. How should we choose the number of neurons and activation function of last layer?

Deeper Networks

Use again the same framework of previous task and replace the LeNet architecture by implementing the following figure (AlexNet architecture):

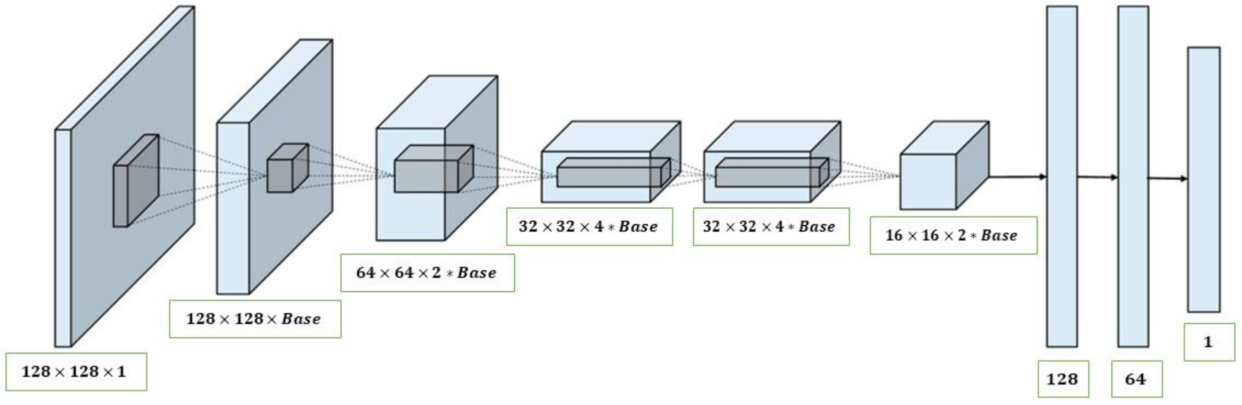
$128 \times 128 \times 1$
$128 \times 128 \times Base$
$64 \times 64 \times 2 * Base$
$32 \times 32 \times 4 * Base$
$32 \times 32 \times 4 * Base$
$16 \times 16 \times 2 * Base$
128
64
1

Fig.5 AlexNet Architecture.

```
def model(img_ch, img_width, img_height):

    model = Sequential()

    model.add(Conv2D(filters=Base, input_shape=(img_width, img_height, img_ch),
                     kernel_size=(3,3), strides=(1,1), padding='same'))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))

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

    model.add(Conv2D(filters=Base*4, kernel_size=(3,3), strides=(1,1), padding='same'))
    model.add(Activation('relu'))

    model.add(Conv2D(filters=Base*4, kernel_size=(3,3), strides=(1,1), padding='same'))
    model.add(Activation('relu'))

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

    model.add(Flatten())
    model.add(Dense(128))
    model.add(Activation('relu'))

    model.add(Dense(64))
    model.add(Activation('relu'))

    model.add(Dense(1))
    model.add(Activation('sigmoid'))

    model.summary()
    return model
```

Taks6A) Read the skin images with the size of 128*128 and train the AlexNet model with the following parameter: Batch_size = 8; Epochs = 50; Base = 32; learning rate = 0.0001; Evaluate the model performance.

Taks6B) Change the Base parameter as 16 and 8 and try again.

Taks6C) Set the Base = 8 and Batch size = 16. How do you interpret the observed results?

Task6D) By finding the optimum values of batch_size, learning rate, and base parameters, train the model for 100 epochs and make sure it is not overfitted. Report the classification accuracy of the best model. Then, for this model, only change the optimizer algorithm from Adam to SGD and RMSprop and compare the observed results.

Task6E) "Binary cross entropy" is not the only loss function for a binary classification task. Run the code again by changing the optimizer into "hinge". Please note, you need to have class labels as [-1, 1] therefore you need to change the labels as :

```
y_test[y_test == 0] = -1
y_train[y_train == 0] = -1
```

Taks7) Read the skin images with the size of 128*128 and repeat the Task6D for the VGG16. For this exercise, you need to implement the model first. Use the framework from previous exercise and just replace the model function with your implementation.

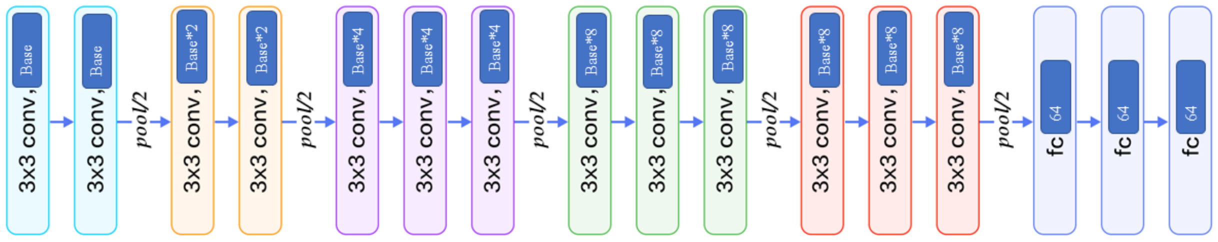

Fig.6 VGG16 Architecture.

A simple implementation of VGG16 model can be found at https://engmrk.com/vgg16-implementation-using-keras/

Task7) For the skin images, compare the results you achieved from the 3 implemented models: LeNet, AlexNet, and VGG.

Task8) Repeat Task7 for the fractured bone image classification. Please note, in order to read the bone images, you need to apply some changes to the data loader:

```
data_path = '/Lab1/Bone/'
img_w, img_h = 128, 128
replace the pattern1 and pattern2 in train_data and test_data functions with 'AFF' and 'NFF'
```

Task9) With the implemented models, which dataset could be better classified? Skin or bone images? How do you make sure that achieved results are reliable?

Bonus Task) In previous tasks, you experienced binary classification tasks by implementing three deep networks. In this bonus exercise, the data set includes X-ray images of 9 different organs. Therefore, you are expected to extend the implemented models into a multi-class classification task. Modify the data loader to properly load the images along with their class labels. Extend the LeNet, AlexNet, and VGG models for multi-class classification tasks. Tune the models by finding the optimum values for hyperparameters and compare the achieved results from the three models. Report the learning curves for both of the loss and accuracy values (for train and test data).

`Data path is: '/Lab1/X_ray/'`

==

Although residual networks are among the powerful classification models, their applications are not restricted only to image classification. In fact, for many different architectures, it would be useful to replace the conventional convolutional layers with residual blocks.

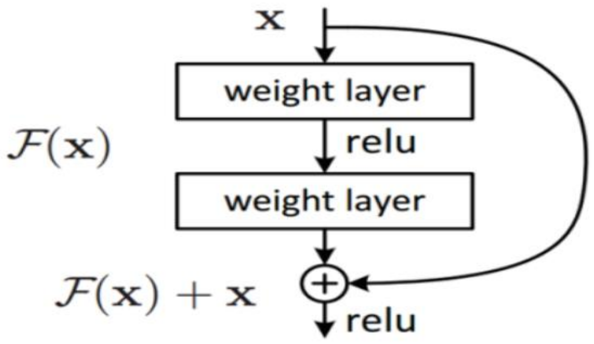
x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Fig.7 Schematic view of a residual block.

During this course, you will be familiar with this approach. A simple yet effective implementation of such models can be found at https://www.kaggle.com/meownoid/tiny-resnet-with-keras-99-314 .

Good luck.