The purpose of this laboratory is to tune the networks you already worked with in the last session in order to speed up the training, improve the performance as well as minimizing the risk of overfitting. Employing batch normalization and drop out techniques are two conventional ways to regularize the training procedure that you will apply them on AlexNet and VGG16 networks. Another approach is to synthetically increase the number of training data which is known as data augmentation. As the second step of this lab, you will experiment some image augmentation techniques. Afterwards, you will also be familiar with class imbalance problems, transfer learning technique, and visualization of the feature maps.

Regularization Techniques

Task1a) Employ the AlexNet model with the following architecture: five convolutional layers (base=8), followed by three dense layers (64,64,1), three max-pooling layers after 1st, 2nd, and 5th blocks; LR=0.0001, batch size=8, Adam optimizer, and image size=(128,128,1). Train this model for 50 epochs on skin images. What is the final training accuracy?

Task1b) With the same model and same settings, now, insert a batch normalization layer at each convolutional blocks (right after convolution layer and before activation function). At which epoch do you observe the same training accuracy as task1a? What is the final training accuracy? What is the effect of batch normalization layer?

Task1c) Train again the same model with exact same parameters except LR = 0.00001 and epochs = 80 with and without batch normalization layers. Focus on validation accuracy. Which model resulted in higher validation accuracy? Which model resulted in higher generalization power? How do you justify the effect of batch normalization?

Task2) Use the same model as task1c with the exact same parameters. Add drop out layers after the first two fully connected layers (right after activation layers with drop out rate of 0.4). Run the model with/without batch normalization layers and discuss the observed results. How does drop out layer help your model?

Task3) Use the same model as task1c but set the Base parameters as 64 and remove the batch normalization layers. Instead, insert spatial dropout layers at each convolutional blocks (after activation function, and before max-pooling). Set the dropout rate of spatial drop out layers as 0.1 and the rate of 0.4 for the drop out layers after the first fully connected layers. Then let the model runs for 150 epochs with LR=0.00001. Save the loss and accuracy values for the validation data. Then, run the same model with the same settings but remove all the drop out layers. Which models perform better? Which models resulted in higher generalization power? In general, discuss how the drop out layers would be helpful?

Task4) Try to improve the performance of VGG16 model while prevent it from overfitting by using batch normalization, spatial dropout, and/or drop out techniques. Tune the model parameters to achieve the best classification accuracy over the validation set and save the observed results for both skin and bone data.

Data Augmentation

Having a large number of training data is crucial for any deep learning tasks; however, most often, the number of available data is not that large. Augmenting the available data by applying geometrical and/or intensity transformations on them is an efficient strategy to tackle the unavailability of such a large dataset.

Task5a) Read the following codes to find out how they work. Then, change the following parameters scale_factor, Angle, Min and Max percentile to see their effects.

```
import numpy as np
from skimage.io import imread
from skimage.transform import rescale
from skimage.transform import rotate
from skimage import exposure
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")

Sample = '/Lab1/X_ray/train/C4_4662.jpg'
Img = imread(Sample)
row, col = Img.shape

def show_paired(Original, Transform, Operation):
    fig, axes = plt.subplots(nrows=1, ncols=2)
    ax = axes.ravel()
    ax[0].imshow(Original, cmap='gray')
    ax[0].set_title("Original image")

    ax[1].imshow(Transform, cmap='gray')
    ax[1].set_title(Operation + " image")
    if Operation == "Rescaled":
        ax[0].set_xlim(0, col)
        ax[0].set_ylim(row, 0)
    else:
        ax[0].axis('off')
        ax[1].axis('off')
    plt.tight_layout()

# Scaling
scale_factor = 0.2
image_rescaled = rescale(Img, scale_factor)
show_paired(Img, image_rescaled, "Rescaled")

# Roation
Angle = 30
image_rotated = rotate(Img, Angle)
show_paired(Img, image_rotated, "Rotated")

# Horizontal Flip
horizontal_flip = Img[:, ::-1]
show_paired(Img, horizontal_flip, 'Horizontal Flip')

# Vertical Flip
vertical_flip = Img[::-1, :]
show_paired(Img, vertical_flip, 'vertical Flip')


# Intensity rescaling
Min_Per, Max_Per = 5, 95
min_val, max_val = np.percentile(Img, (Min_Per, Max_Per))
```

```
better_contrast = exposure.rescale_intensity(Img, in_range=(min_val, max_val))
show_paired(Img, better_contrast, 'Intensity Rescaling')
```

Task5b) A practical way to perform data augmentation is to develop a generator. The following code is an example of how you can generate random augmented images with tensorflow built-in generator. Search for other options of the ImageDataGenerator and apply them in code and try to find out how it works.

```
import numpy as np
from skimage.io import imread
import matplotlib.pyplot as plt
%matplotlib inline
from tensorflow.keras.preprocessing.image import ImageDataGenerator, array_to_img

Sample = '/Lab1/X_ray/train/C4_4662.jpg'
Img = imread(Sample)
Img = np.expand_dims(Img, axis = 2)
Img = np.expand_dims(Img, axis = 0)


count = 5
MyGen = ImageDataGenerator(rotation_range = 20,
                           width_shift_range = 0.2,
                           horizontal_flip = True)


fix, ax = plt.subplots(1,count+1, figsize=(14,2))
images_flow = MyGen.flow(Img, batch_size=1)
for i, new_images in enumerate(images_flow):
    new_image = array_to_img(new_images[0], scale=True)
    ax[i].imshow(new_image,cmap="gray")
    if i >= count:
        break
```

Task6) Develop a framework for training the models with augmenting the training data as:

```
# Import required libraries …
# Data and model parameters
# TRAIN_DIR = '/Lab1/Lab2/Skin/train/' and VAL_DIR = '/Lab1/Lab2/Skin/validation/'
# Def model() …
# train_datagen = ImageDataGenerator(…)
# train_generator = train_datagen.flow_from_directory(…)
# val_datagen = ImageDataGenerator(…)
# val_generator = val_datagen.flow_from_directory(…)
# MyModel = model(…)
# MyModel.compile(…)
# History = MyModel.fit_generator(…)
# Learning curve plots
```

Apply image augmentations on training data including rotation_range=10, width_shift_range=0.1, height_shift_range=0.1, rescale=1./255, and horizontal_flip=True. However, for validation data you just need to apply rescale=1./255.

Use the AlexNet model with the batch normalization layers, and drop out layers for the first two dense layers(rate=0.4). Set the Base parameter as 64, and assign 128 neurons for the first dense layer and 64 for the second one. LR=0.00001, batch-size=8 and train the model on skin images for for 80 epochs. Does data augmentation help to improve the model performance? Why?

(Please note since you will use the ImageDataGenerator, then you do not need to load the image data independently like what you already done.)

Task7) Repeat task6 for VGG model for both of skin and bone data set.

```
# TRAIN_DIR = '/Lab1/Lab2/Bone/train/' and VAL_DIR = '/Lab1/Lab2/ Bone /validation/'
Base=64;LR=0.00001;batch size=8;3 dense layers each 64 neurons, epochs=80.
```

Transfer Learning

Transfer learning is a method that an already trained model can be reused as the starting point of another similar task. It often starts by training a model on a large dataset such as ImageNet. After the model learns the task, model weights will be saved. The saved weights will be loaded on the new similar model that is called pre-trained network. If the new data set is very small, it's better to only train the final layers or the network. In that case, from the pretrained model all the layers will be hold except the last layer which will be replaced by new layer(s). The training process is, then, applied only on the new added layer(s). This procedure is called fine tuning.

Task8) In this task, you will employ a pretrained VGG16 model which was trained on ImageNet database. By fine tuning this model, you will classify the skin and bone data set again. To do so, as the first step, the pretrained model should be loaded. Training and validation data should passed through the layers and the model output should be saved to be fed, later, to the new layer. Finally, only the new added layers will be trained. Follow the code and complete it as:

```
Image size = 224*224; epochs = 150; LR=0.00001; batch size=8; and the MLP model should contains
Dense(128), drop(0.5) and a another dense layer for classification.

import os
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout, ZeroPadding2D
from tensorflow.keras.layers import Convolution2D, MaxPooling2D
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import applications
import numpy as np


def get_length(Path, Pattern):
    Length =  len(os.listdir(os.path.join(Path, Pattern)))
    return Length

def VGG_16(weights_path=None):
    model = Sequential()
    model.add(ZeroPadding2D((1,1),input_shape=(3,224,224)))
    model.add(Convolution2D(64, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(64, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(128, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(128, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
```

```
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(Flatten())
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1000, activation='softmax'))
    model.summary()



# parameters (TODO)
train_data_dir = '/Lab1/Lab2/Bone/train/'
validation_data_dir = '/Lab1/Lab2/Bone/validation/'
img_width, img_height = ???
epochs = ???
batch_size = ???
LR = ???
# number of data for each class
Len_C1_Train = get_length(train_data_dir,'AFF')
Len_C2_Train = ???
Len_C1_Val = ???
Len_C2_Val = ???

# loading the pre-trained model
model = applications.VGG16(include_top=False, weights='imagenet')
model.summary()


# Feature extraction from pretrained VGG (training data)
datagen = ImageDataGenerator(rescale=1. / 255)

train_generator = datagen.flow_from_directory(
        train_data_dir,
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode=None,
        shuffle=False)

features_train = model.predict_generator(
        train_generator,
        (Len_C1_Train+Len_C2_Train) // batch_size, max_queue_size=1)


# To DO: Feature extraction from pretrained VGG (validation data)
???
```

```
# training a small MLP with extracted features from the pre-trained model
train_data = features_train
train_labels = np.array([0] * int(Len_C1_Train) + [1] * int(Len_C2_Train))

validation_data = features_validation
validation_labels = np.array([0] * int(Len_C1_Val) + [1] * int(Len_C2_Val))

# TODO: Building the MLP model
???

# TODO: Compile and train the model, plot learning curves
???
```

Task9) Train the fine-tuned model with skin and bone images. Do you observe some changes in model performance? Describe how transfer learning would be useful for training? How can you make sure that the results are reliable?

Visualizing Activation Maps

In classification tasks, regardless of the model performance, it is important to find out where the model focuses to make the classification decision. One strategy is to visualize class activation maps. Simply speaking, it will let us know which region of each image belongs to each class.

Task10) Back to bone image classification. Design a VGG16 with 2 neurons at the last year so that the activation function should be set as “softmax” and categorical cross entropy as loss function. (Remember to name the last convolutional layer as name = 'Last_ConvLayer'). Train the model with data augmentation techniques and after the model learned, follow the implementation below and interpret the observed results:

```
from tensorflow.keras import backend as K
from skimage.io import imread
from skimage.transform import resize
import cv2

Sample = '/Lab1/Lab2/Bone/train/AFF/14.jpg'
Img = imread(Sample)
Img = Img[:,:,0]
Img = Img/255
Img = resize(Img, (img_height, img_width), anti_aliasing = True).astype('float32')
Img = np.expand_dims(Img, axis = 2)
Img = np.expand_dims(Img, axis = 0)
preds = model.predict(Img)
class_idx = np.argmax(preds[0])
print(class_idx)
class_output = model.output[:, class_idx]
last_conv_layer = model.get_layer("Last_ConvLayer")

grads = K.gradients(class_output, last_conv_layer.output)[0]
pooled_grads = K.mean(grads, axis=(0, 1, 2))
iterate = K.function([model.input], [pooled_grads, last_conv_layer.output[0]])
pooled_grads_value, conv_layer_output_value = iterate([Img])
for i in range(Base*8):
    conv_layer_output_value[:, :, i] *= pooled_grads_value[i]

heatmap = np.mean(conv_layer_output_value, axis=-1)
heatmap = np.maximum(heatmap, 0)
heatmap /= np.max(heatmap)
# For visualization
img = cv2.imread(Sample)
```

```
img = cv2.resize(img, (512, 512), interpolation = cv2.INTER_AREA)
heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))
heatmap = np.uint8(255 * heatmap)
heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)
superimposed_img = cv2.addWeighted(img, 0.6, heatmap, 0.4, 0)
plt.figure()
plt.imshow(img)
plt.figure()
plt.imshow(superimposed_img)
```

Bonus Task) Design a *N*-Layer Residual network for classification task. Try to find the optimum hyperparameters such as batch size, depth of model, learning rate etc. Train your model with data augmentation for bone and X-ray images. Apply the followings in your implementation:

B1) As you might noticed, the number of image data in each class of X-ray images are not equal. Therefore, you need to handle this class imbalanced problem.

B2) In previous tasks, for simplicity, you performed all the experiments on randomly selected train and validation data. Split your data through 3-fold cross validation approach and report the classification accuracy of your model as mean$\mp$variatoin scores over the validation set.

B3) Optimize your implementations by adding early stopping criteria and learning rate schedule. You can also save the whole model or model weights only. This is a very useful strategy by which you can train your model for few epochs and later, you can load the weights and continue the training process.

Good luck.