The purpose of this laboratory is to tune the networks you already worked with the last session 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. In this lab, you will apply such regularization techniques on AlexNet and VGG16 networks. In the second step of this lab, you will experiment with some image augmentation technique to synthetically increase the number of training data. Finally, you will be familiar with class imbalance problems, transfer learning techniques, and visualization of the feature maps. Please remember to save the observed results at each task as you need to make some comparisons with the subsequent tasks. Please note that the data required for this lab can be found within the data directory …/Lab2/…

Regularization Techniques

Task1a) Employ the AlexNet model with the following architecture: five convolutional blocks [with feature maps of the size of base, base*2, base*4, base*4, base*2 where base=8], followed by three dense layers with 64, 64, and 1 neuron, respectively. Add three max-pooling layers after 1st, 2nd, and 5th convolutional blocks. Set the following parameters: learning rate=0.0001, batch size=8, 'relu' as activation function, 'Adam' as optimizer, and image size=(128,128,1). Train this model for 50 epochs on skin images. What are the values of the train and validation accuracy? How do you interpret the learning curves? Add two drop out layers after the first two dense layers with the dropout rate of 0.4 and repeat the experiments and compare the results. What is the effect of adding drop out layers?

Task1b) With the same model and same settings, now insert a batch normalization layer at each convolutional block (right after convolution layer and before activation function). At which epoch do you observe the same training accuracy as task1a? What is the value of final training accuracy? What is the effect of the batch normalization layer? Similar to task1a, do this task with and without drop out layers.

Task1c) Train again the same model with precisely the same parameters except learning rate = 0.00001 and epochs = 80 with and without batch normalization layers (in bose cases, use the drop out layers). Focus on validation loss & accuracy. Which model resulted in higher validation accuracy? How do you explain the effect of batch normalization?

Task1d) Keep the settings from the task1c unchanged and train the model for 150 epochs with and without batch normalization layers (in bose cases, use the drop out layers). Which model yields more accurate results? Which of them has more generalization power?

Task2a) Use the same model as task1d but set the "base" parameter as 32 and replace the batch normalization layers with 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 normal drop out layers after the first two 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 spatial drop out layers. Which of them converges faster? Why?

Task2b) Repeat the task2a for 250 epochs with and without spatial dropout layers. In general, discuss how the drop out technique (spatial and normal one) would help the learning procedure.

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.

Task3a) Read the following set of codes and find out how they work. Then, change the following parameters: scale_factor, Angle, low/high bounds 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
import warnings
warnings.filterwarnings("ignore")

sample_dir = '../Lab1/X_ray/train/C4_4662.jpg'
img = imread(sample_dir)
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
low_bound, high_boud = 5, 95
min_val, max_val = np.percentile(img, (low_bound, high_boud))
better_contrast = exposure.rescale_intensity(img, in_range=(min_val, max_val))
show_paired(img, better_contrast, 'Intensity Rescaling')
```

Task3b) A practical way to perform the data augmentation technique is to develop a generator. The following code is an example of how you can generate augmented images randomly with a TensorFlow built-in generator.

```
import numpy as np
from skimage.io import imread
import matplotlib.pyplot as plt
from 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
my_gen = ImageDataGenerator(rotation_range = 20,
                            width_shift_range = 0.2,
                            horizontal_flip = True)


fix, ax = plt.subplots(1,count+1, figsize=(14,2))
images_flow = my_gen.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
```

Task4) Develop a framework for training the models with augmenting the training data as:

```
# 1)Import required libraries …
# 2) set the directory to the data and model parameters like:
#       train_dir = '/Lab2/Skin/train/' and val_dir = '/Lab2/Skin/validation/'
# 3)Model architecture like:
        Def model(…)
# 4)Set train data generator like:
       train_datagen = ImageDataGenerator(types of augmentation)
# 5)Compile the train generator over the training directory …:
       train_generator = train_datagen.flow_from_directory(…)
# 6)Set and compile the validation generator like:
       val_datagen = ImageDataGenerator(…)
       val_generator = val_datagen.flow_from_directory(…)
# 7)Compile the model like:
       my_model = model(…)
       my_model.compile(…)
# 8)Training the model like:
       model_hist = my_model.fit_generator(…)
# 9)Learning curve plots
```

The augmentations that are going to applied over the training data include: 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 rescale=1./255 the images.

Please note that, so far, we have used 8-bit images means that the intensity value of each pixel lies in the range of [0,255]. This means that we need to first normalize the intensity range of the loaded images. In the previous exercises, we used 'skimage' library to load the images. The built-in function within 'skimage' automatically read the images and normalized their arrays in the range of [0,1] so that it did not require the intensity normalization step.

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. Set the optimizer=Adam, LR=0.00001, batch-size=8, and train the model on skin images for 80 epochs. How the data augmentation impact model training? Why?

(Please note since you will use the ImageDataGenerator, then you do not need to load all the data into the memory as it loads the data batch by batch.)

Task5) Repeat task6 for VGG model for both of skin and bone data set.

```
# TRAIN_DIR = '../Lab2/Bone/train/' and VAL_DIR = '../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 for a certain task can be reused as the starting point of another task. It often starts by training a model on a large dataset such as ImageNet. This trained model is called pre-trained network. After the model learns the primary task, model weights will be saved to be used as initial weights of the new model with similar (not necessirily) architecture to train another dataset.

If the number of available data for a certain task is not enough to train a deep network, sometimes we can use a pretarined model and keep the weights of some of the layers of the trained model but only train few layers of this network (or maybe adding some new layers on the top of the pretarined network and only train and update these new layers). In another term, only the weights of the new layers will be updated, and the rest will not be updated during backpropagation. The training process is then applied only to the new added layer(s). This procedure is called fine-tuning.

Task6) In this task, you will employ a pre-trained VGG16 model that was trained on the 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 pre-trained model should be loaded. This pretarined model will be used as a feature extractor model. In other words, training and validation data should be passed through the layers of the pre-trained VGG network, and the output of last convolutional block will be used as the extracted features. These features, then, are fed, to the new layer(s) in a way that only the new added layer(s) 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 the 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):
    # Pattern: name of the subdirectory
    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 = '…Lab2/Bone/train/'
validation_data_dir = '…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
# include top: false means that the dense layers at the top of the network will not be used.
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)
# Extracting the features from the loaded images
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
# In fact this MLP will be used instead of the dense layers of the VGG model
# and only this MLP will be trained on the dataset.
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
???
```

Task7) Train the fine-tuned model with skin and bone images. Compare the observed results from transfer learning against the ones you already achieved by training the models from scratch. 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 that the model focused on which region of the to make the classification decision. To better understand which region of the image play more important role in the decision making, one strategy is to visualize class activation maps.

Task8) Design a VGG16 model similar to what you already implemented in the lab 2 task7a. Please note the model should contain 3 dense layers at the top each of which contain 64 neurons as well

as dropout layer (rate=0.4). Finally, a dense layer with 2 neurons at the last layer 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 = 'Last_ConvLayer') [model.add(Conv2D(filters=Base*8, kernel_size=(3,3), strides=(1,1), padding='same', name = 'Last_ConvLayer'))] Train the model with data augmentation techniques with parameters as : base=8; batch_s = 8; LR=1e-5; img_size = 128*128, epoch = 80. After the training process, follow the implementation below and interpret the observed results. What can you infer from visualization of the class activation maps?

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

sample_dir = '../../Lab2_Preparation/Data/Lab2/Bone/train/AFF/14.jpg'
img = imread(sample_dir)
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('the predicted class label is {}'.format(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)


img_ = cv2.imread(sample_dir)
img_ = cv2.resize(img_, (512, 512), interpolation = cv2.INTER_AREA)
#img = img/255
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 the 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 X-ray images. Apply the followings in your implementation:

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

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

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 a few epochs, and later, you can load the weights and continue the training process.

Good luck.