Healthy-Defective Metal Object Classification

Goal of this assignment is to build an AI model which can accurately detect the crack defect in metal objects. Based on the presence of the crack defect, we aim to build a classification model.

Since this is an image classification problem, we will be leveraging CNN models or ConvNets to solve this problem for us. We will start by building simple CNN models from scratch, then try to improve using techniques such as regularization and image augmentation. Then, we will try and leverage pre-trained models to unleash the latest powerful transfer learning technique.

Modelling Overview

We have stored our Defective and Healthy images in two separate folders. We load them using getLabel function and store their paths in 'images' and their labels in 'labels'.

Next, we split our dataset in two parts training and validation using split_trainTest function. Our dataset contains 111 Defective images and 139 OK/Health images. For modelling purpose, we will split our dataset into two parts – Training and Validation. For the split we collect all the images in a python list and shuffle it to keep the randomness. Then we do a good 0.25 split of the final combination.

```
def getLabel(filePaths):
    labels = []
    for img in filePaths:
        if 'Healthy' in img:
            labels.append(0)
        elif 'defects' in img:
            labels.append(1)

    dataZip = list(zip(filePaths, labels))
    shuffle(dataZip)
    filePaths, labels = zip(*dataZip)
    return filePaths, labels


def split_trainTest(imgsAll, labelAll, splitRatio = 0.20):
    dataZip = list(zip(imgsAll, labelAll))
    shuffle(dataZip)
    imgsAll, labelAll = zip(*dataZip)
    splitPoint = int(len(imgsAll)*splitRatio)

    trainImgs = imgsAll[:int(len(imgsAll) - splitPoint)]
    trainLabel = labelAll[:int(len(imgsAll) - splitPoint)]
    testImgs = imgsAll[int(len(imgsAll)-splitPoint):]
    testLabel = labelAll[int(len(imgsAll)-splitPoint):]

    return trainImgs, testImgs, trainLabel, testLabel

images, labels = getLabel(glob.glob('/content/JBMClassification/*/*.jpg'))
# split the images into train and test sets.
X_train, X_test, y_train, y_test = split_trainTest(images, labels,  splitRatio=0.25)
```

Next, we will covert out large pixel images into (224, 224) images as later Inception model used for transfer learning accepts input images in this size.

```
display(print(f'SHape of training dataset {len(X_train)} and shape of validation set is {len(X_test)}'))

train_imgs = [img_to_array(load_img(img, target_size=IMAGE_SIZE)) for img in X_train]
train_imgs = np.array(train_imgs)
train_labels = list(y_train)
validation_imgs = [img_to_array(load_img(img, target_size=IMAGE_SIZE)) for img in X_test]
validation_imgs = np.array(validation_imgs)
validation_labels = list(y_test)
```

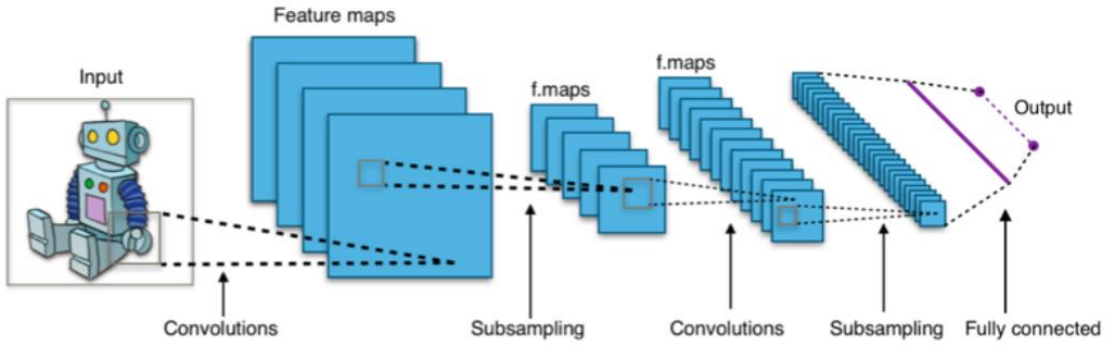
```
SHape of training dataset 188 and shape of validation set is 62
None
```

We will scale each image with pixel values between (0, 255) to between (0, 1) because deep learning models work really well with small input values.

```
print('Train dataset shape:', train_imgs.shape,
      '\tValidation dataset shape:', validation_imgs.shape)
train_imgs_scaled = train_imgs.astype('float32')
validation_imgs_scaled  = validation_imgs.astype('float32')
train_imgs_scaled /= 255
validation_imgs_scaled /= 255
```

```
Train dataset shape: (188, 224, 224, 3)         Validation dataset shape: (62, 224, 224, 3)
```

We will start by building a basic CNN model with three convolutional layers, coupled with max pooling for auto-extraction of features from our images and also down sampling the output convolution feature maps.



A Typical CNN (Source: Wikipedia)

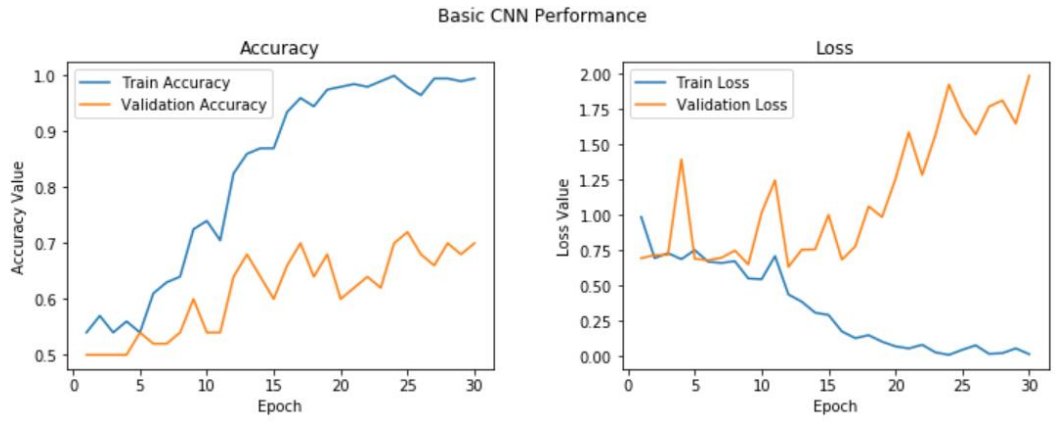
Let’s leverage Keras and build our CNN model architecture now.

```
from keras import backend as K

model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
model.summary()
```

We train our model with batch_size = 32, epochs = 30 settings on training data and validate it consequently on our validation dataset.

```
history = model.fit(x=train_imgs_scaled, y=train_labels,
                    validation_data=(validation_imgs_scaled, validation_labels),
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1)
```

After the model run, we saw that our model was overfitting and not doing that great on both training and validation datasets.



Let's improve upon our base CNN model by doing what we call regularization. In CNN, best way to do that is to use a drop out layer. We added two drop out layers with drop out rate of 0.5 after each hidden dense layer. Dropout randomly masks the outputs of a fraction of units from a layer by setting their output to zero (in our case, it is 50% of the units in our dense layers). But in the end we were still overfitting the model, though it took slightly longer and we also got a slightly better validation accuracy of around ~75% which was an improvement but not amazing.

The reason for model overfitting is because we have much less training data and the model keeps seeing the same instances over time across each epoch. A way to combat this would be to leverage

an image augmentation strategy to augment our existing training data with images that are slight variations of the existing images.

The idea behind image augmentation is that we follow a set process of taking in existing images from our training dataset and applying some image transformation operations to them, such as rotation, shearing, translation, zooming, and so on, to produce new, altered versions of existing images. Due to these random transformations, we don't get the same images each time, and we will leverage Python generators to feed in these new images to our model during training. The Keras framework has an excellent utility called ImageDataGenerator that can help us in doing all these operations.

Model with **Image Augmentation**

```
[ ]  # train_datagen = ImageDataGenerator(rescale=1./255, zoom_range=0.3, rotation_range=50,
     #                                    width_shift_range=0.2, height_shift_range=0.2, shear_range=0.2,
     #                                    horizontal_flip=True, fill_mode='nearest')

     train_datagen = ImageDataGenerator(rescale=1. / 255, shear_range=0.2, zoom_range=0.2, horizontal_flip=True)

     # this is the augmentation configuration we will use for testing:
     # only rescaling
     val_datagen = ImageDataGenerator(rescale=1./255)

     train_generator = train_datagen.flow(train_imgs_scaled, train_labels, batch_size=batch_size)
     val_generator = val_datagen.flow(validation_imgs_scaled, validation_labels, batch_size=batch_size//2)
```

A pre-trained model like the InceptionV3 is an already pre-trained model on a huge dataset (ImageNet) with a lot of diverse image categories. Considering this fact, the model should have learned a robust hierarchy of features, which are spatial, rotation, and translation invariant with regard to features learned by CNN models. Hence, the model, having learned a good representation of features for over a million images belonging to 1,000 different categories, can act as a good feature extractor for new images suitable for computer vision problems. These new images might never exist in the ImageNet dataset or might be of totally different categories, but the model should still be able to extract relevant features from these images.

This gives us an advantage of using pre-trained models as effective feature extractors for new images, to solve diverse and complex computer vision tasks, such as solving our model.

We will use Inception V3 as a simple feature extractor by freezing all the convolution blocks to make sure their weights don't get updated after each epoch.

```
from keras.models import Model
import keras
from keras.applications.inception_v3 import InceptionV3

inception = InceptionV3(weights='imagenet', include_top=False,
                        input_shape=(224, 224, 3))

output = inception.layers[-1].output
output = keras.layers.Flatten()(output)
incept_model = Model(inception.input, output)

incept_model.trainable = False
for layer in incept_model.layers:
    layer.trainable = False
```

A way to save time in model training is to use this model and extract out all the features from our training and validation datasets and then feed them as inputs to our classifier. Let's extract out the bottleneck features from our training and validation sets now.

```
def get_bottleneck_features(model, input_imgs):
    features = model.predict(input_imgs, verbose=0)
    return features

train_features_vgg = get_bottleneck_features(incept_model, train_imgs_scaled)
validation_features_vgg = get_bottleneck_features(incept_model, validation_imgs_scaled)

print('Train Bottleneck Features:', train_features_vgg.shape,
      '\tValidation Bottleneck Features:', validation_features_vgg.shape)
```

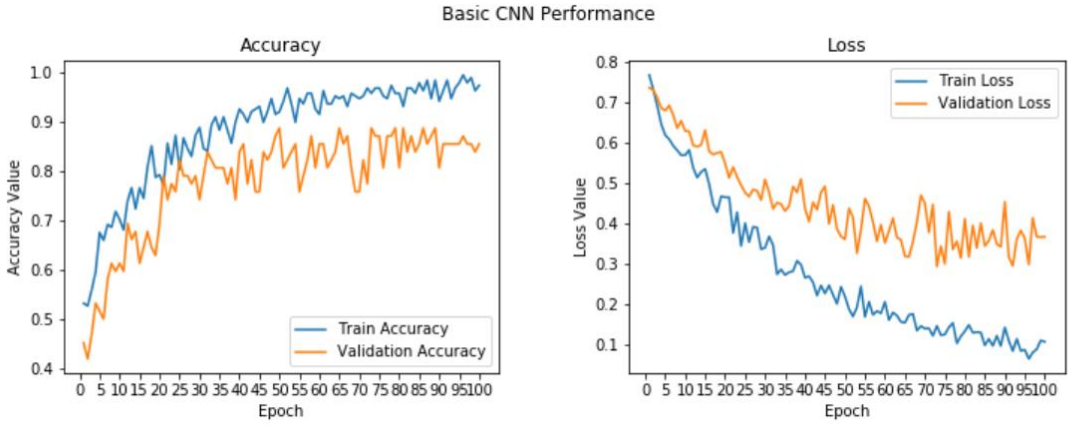
```
Train Bottleneck Features: (188, 51200)         Validation Bottleneck Features: (62, 51200)
```

Let's build the architecture of our deep neural network classifier now, which will take these features as input.

```
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, InputLayer
from keras.models import Sequential
from keras import optimizers

input_shape = incept_model.output_shape[1]

model = Sequential()
model.add(InputLayer(input_shape=(input_shape,)))
model.add(Dense(64, activation='relu', input_dim=input_shape))
model.add(Dropout(0.3))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['accuracy'])

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 64)                3276864
_________________________________________________________________
dropout_4 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                4160
_________________________________________________________________
dropout_5 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_7 (Dense)              (None, 32)                2080
_________________________________________________________________
dropout_6 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 33
=================================================================
Total params: 3,283,137
Trainable params: 3,283,137
Non-trainable params: 0
```

```
history = model.fit(x=train_features_vgg, y=train_labels,
                    validation_data=(validation_features_vgg, validation_labels),
                    batch_size=batch_size,
                    epochs=150,
                    verbose=1)
```



We can see that our model has an overall validation accuracy of ~88%, which is a huge improvement from our previous model, and also the train and validation accuracy are closer to each other, indicating that the model is not overfitting that.

We tested our model on 20 images and result is as follows:

```
meu.display_model_performance_metrics(true_labels=num2class_label_transformer(test_labels_enc), predicted_labels=predictions,
                                      classes=list(set(num2class_label_transformer(test_labels_enc))))
```

```
Model Performance metrics:
------------------------------
Accuracy: 0.9
Precision: 0.9167
Recall: 0.9
F1 Score: 0.899

Model Classification report:
------------------------------
              precision    recall  f1-score   support

     Healthy       1.00      0.80      0.89        10
   Defective       0.83      1.00      0.91        10

   micro avg       0.90      0.90      0.90        20
   macro avg       0.92      0.90      0.90        20
weighted avg       0.92      0.90      0.90        20


Prediction Confusion Matrix:
------------------------------
                  Predicted:
                     Healthy Defective
Actual: Healthy            8         2
        Defective          0        10
```

Considering the fact that we had such a small dataset to work with, this is a huge improvement.