Activity 1 – Classifying Grayscale Handwritten Digits

In this activity, we will learn:

- ❑ MNIST Handwritten Digit Dataset
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Define, compile and train a simple model
- ❑ Evaluate the model
- ❑ Improving the model with hidden layers

1. MNIST Handwritten Digit Dataset

a) The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It is a dataset of 70,000 small square 28 × 28 pixel grayscale images of handwritten single digits between 0 and 9.

b) Keras provides access to the MNIST dataset via the mnist.load_dataset() function. It returns two tuples, one with the input and output elements for the standard training dataset, and another with the input and output elements for the standard test dataset. The code snippet below loads the dataset and summarizes the shape of the loaded dataset.

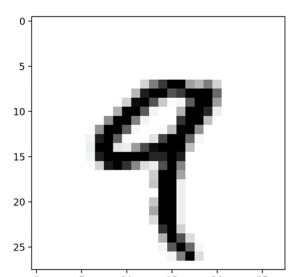
```
from keras import models
from keras import layers
from keras.datasets import mnist
from keras.utils import to_categorical

# Load the MNIST data
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
print('train_images shape:', train_images.shape)
print('test_images shape:', test_images.shape)
print('train_labels shape:', train_labels.shape)
```

Running the above loads the dataset and prints the shape of the input and output components of the train, test and train labels and test splits of images.

```
train_images shape: (60000, 28, 28)
test_images shape: (10000, 28, 28)
train_labels shape: (60000,)
```

c) The images are grayscale with a black background (0 pixel value) and the handwritten digits in white (pixel values near 255). The images are easier to review when we reverse the colours and plot the background as white and the handwritten digits in black.

```
# Display a sample
extract_digit = 4
digit = train_images[extract_digit]
label = train_labels[extract_digit]
print("Label =",label)
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```



d) The next preprocessing task we need to perform is to flatten the 2D images into a 1D array. Due to all the weights computation, we will need to normalise the pixel value by dividing them by 255.

```
# Process the data for the usage of ANN
train_images = train_images.reshape((60000, 28 * 28)) # Flatten the image
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

e) We then use keras utility function to_categorical() function convert an array of labelled data(from 0 to 9) to one-hot vector. For an indepth discussion on one-hot encoding, please refer to https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/

```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

f) Now we are ready to define our neural network. Keras has different APIs for defining neural networks that we can choose between. The most commonly used is the Sequential API. The Sequential API is the version in which we define it one layer at a time, in sequence. It's the easiest way to do it, so that's what we'll use too. So, first we create a new Sequential() model.

```
# Define the network
network = models.Sequential()
```

g) This is the model object to which we'll add our layers. Now we can define the layers of our neural network calling the network.add() function and passing in the kind of layer we want to add.
So far, we've only learned about the simplest kind of neural network layer where every node in the layer is connected to every node in the following layer. These are called dense layers because the nodes are densely connected.
To create a dense layer in Keras, we can create new Dense() objects and add them to the model. Here's how we'll define the first layer:

```
network.add(layers.Dense(10, activation='softmax', input_shape=(28 * 28,)))
```

Note: This network without any hidden layer cannot effectively classify the MNIST data.

h) We are using the Keras API to define our neural network, but it uses TensorFlow behind the scenes to do all the math. Now that we've declared all the layers, we need to tell Keras to construct the neural network inside of TensorFlow for us. To do that, we'll call the compile() function:

```
# Compile (Create) the network for TF
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

There are two important parameters that we need to specify:

- **loss** is the loss or cost function we are using to measure how wrong our neural network currently is. The Keras documentation lists out all the possible loss functions that you can choose from. Since we are predicting categorial values, we use categorical_crossentropy.
- **optimizer** is which numerical optimization algorithm we will use to train the neural network. We will use **rmsprop**, one of the many gradient descent optimization algorithms provided in keras. For an indepth discussion, please refer to https://ruder.io/optimizing-gradient-descent/

* You can use network.summary() to print out the details of your network.

i) Now we're ready to train the model. Keras models its syntax on scikit-learn, so its training function is also called fit().

```
# Train the network
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

First, we pass in the training data and the matching answers for each training example. Then we have several parameters that we can control:

- **epochs** is how many times we will loop through the entire training dataset before ending the gradient descent training process.
- **batch_size** controls how many training examples are considered at once during each gradient descent update pass.

j) After training, we can call the evaluate() function to have a feel of how our model perform.

```
#Evaluate the network
test_loss, test_acc = network.evaluate(test_images, test_labels)
print('test_acc:', test_acc)
```

k) Run the code. You should see something similar to the following:

```
  128/60000 [..............................] - ETA: 1s - loss: 0.3365 - acc: 0.9141
 5760/60000 [=>............................] - ETA: 0s - loss: 0.3018 - acc: 0.9151
13056/60000 [=====>........................] - ETA: 0s - loss: 0.2954 - acc: 0.9198
21120/60000 [=========>....................] - ETA: 0s - loss: 0.2932 - acc: 0.9197
27136/60000 [============>.................] - ETA: 0s - loss: 0.2955 - acc: 0.9190
31232/60000 [==============>...............] - ETA: 0s - loss: 0.2946 - acc: 0.9180
35712/60000 [================>.............] - ETA: 0s - loss: 0.2925 - acc: 0.9190
42624/60000 [====================>.........] - ETA: 0s - loss: 0.2939 - acc: 0.9184
50048/60000 [========================>.....] - ETA: 0s - loss: 0.2895 - acc: 0.9191
55168/60000 [==========================>...] - ETA: 0s - loss: 0.2882 - acc: 0.9197
58880/60000 [============================>.] - ETA: 0s - loss: 0.2878 - acc: 0.9194
60000/60000 [==============================] - 1s 9us/step - loss: 0.2882 - acc: 0.9194
Epoch 5/5

  128/60000 [..............................] - ETA: 1s - loss: 0.3450 - acc: 0.9297
 6400/60000 [==>...........................] - ETA: 0s - loss: 0.2851 - acc: 0.9230
12288/60000 [=====>........................] - ETA: 0s - loss: 0.2848 - acc: 0.9209
17536/60000 [=======>......................] - ETA: 0s - loss: 0.2824 - acc: 0.9215
21504/60000 [=========>....................] - ETA: 0s - loss: 0.2824 - acc: 0.9221
26880/60000 [============>.................] - ETA: 0s - loss: 0.2794 - acc: 0.9228
35072/60000 [================>.............] - ETA: 0s - loss: 0.2777 - acc: 0.9227
43008/60000 [====================>.........] - ETA: 0s - loss: 0.2816 - acc: 0.9217
48384/60000 [=======================>......] - ETA: 0s - loss: 0.2797 - acc: 0.9224
53632/60000 [=========================>....] - ETA: 0s - loss: 0.2806 - acc: 0.9224
60000/60000 [==============================] - 0s 8us/step - loss: 0.2798 - acc: 0.9223

   32/10000 [..............................] - ETA: 6s
 5920/10000 [================>.............] - ETA: 0s
10000/10000 [==============================] - 0s 10us/step
test_acc: 0.9236
```

l) To improve the result, you can add more hidden layers to the model.

```
# Define the network
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

The network above can be visualised as:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
```

m) Go ahead and add more layers with different number of neurons and check if the accuracy improves.

Activity wrap-up:
We learn
- ❑ MNIST Handwritten Digit Dataset
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Define, compile and train a simple model
- ❑ Evaluate the model
- ❑ Improving the model with hidden layers

Activity 2 –Classifying Fashion

<u>In this activity, we will learn:</u>
- ❑ Fashion-MNIST Dataset
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Define, compile and train a model
- ❑ Evaluate the model
- ❑ Tuning the model by changing the batch sizes and epoch

2. Fashion-MNIST Dataset

a) Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Zalando intends Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

b) Like MNIST, Keras provides access to the Fashion MNIST dataset via the fashion_mnist.load_data() function. It returns two tuples, one with the input and output elements for the standard training dataset, and another with the input and output elements for the standard test dataset. The code snippet below loads the dataset and summarizes the shape of the loaded dataset. The dataset will be downloaded if this is the first time you are using it.

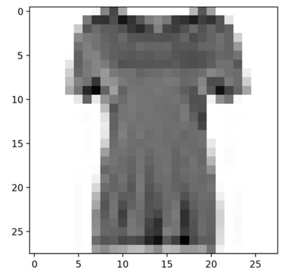
```
from keras import models
from keras import layers
from keras.datasets import fashion_mnist
from keras.utils import to_categorical

# Load the MNIST data
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
print('train_images shape:', train_images.shape)
print('test_images shape:', test_images.shape)
print('train_labels shape:', train_labels.shape)
```

Running the above loads the dataset and prints the shape of the input and output components of the train, test and train labels and test splits of images.

```
train_images shape: (60000, 28, 28)
test_images shape: (10000, 28, 28)
train_labels shape: (60000,)
```

c) The images are grayscale with a black background (0 pixel value) and the handwritten digits in white (pixel values near 255). The images are easier to review when we reverse the colours and plot the background as white and the handwritten digits in black.

```
# Display a sample
extract_digit = 10
digit = train_images[extract_digit]
label = train_labels[extract_digit]
print("Label =",label)
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```



d) The next preprocessing task we need to perform is to flatten the 2D images into a 1D array. Due to all the weights computation, we will need to normalise the pixel value by dividing them by 255.

```
# Process the data for the usage of ANN
train_images = train_images.reshape((60000, 28 * 28)) # Flatten the image
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

e) We then use keras utility function to_categorical() function convert an array of labelled data(from 0 to 9) to one-hot vector. For an in-depth discussion on one-hot encoding, please refer to https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/

```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

f) Now we are ready to define our neural network. We will use the improved model we created in Activity 1.

```
# Define the network
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

g) We will continue to compile, train and evaluate the model to get a feel of the accuracy:

```

network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

history = network.fit(train_images, train_labels, epochs=5, batch_size=128)

test_loss, test_acc = network.evaluate(test_images, test_labels, verbose=2)
print('test_acc:', test_acc)
print("------------------------------------")
```

Note: The history object is returned from calls to the fit() function used to train the model. Metrics are stored in a dictionary in the history member of the object returned. For an in-depth discussion on the history object, please refer to https://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/

h) Run the code. You should see something similar to the following:

```
50688/60000 [========================>.....] - ETA: 0s - loss: 0.2912 - acc: 0.8934
51968/60000 [========================>.....] - ETA: 0s - loss: 0.2904 - acc: 0.8937
52992/60000 [=========================>....] - ETA: 0s - loss: 0.2905 - acc: 0.8937
54016/60000 [==========================>...] - ETA: 0s - loss: 0.2906 - acc: 0.8939
55040/60000 [==========================>...] - ETA: 0s - loss: 0.2910 - acc: 0.8936
56192/60000 [===========================>..] - ETA: 0s - loss: 0.2908 - acc: 0.8936
57344/60000 [===========================>..] - ETA: 0s - loss: 0.2909 - acc: 0.8937
58368/60000 [============================>.] - ETA: 0s - loss: 0.2912 - acc: 0.8937
59392/60000 [============================>.] - ETA: 0s - loss: 0.2922 - acc: 0.8934
60000/60000 [==============================] - 3s 44us/step - loss: 0.2924 - acc: 0.8935
test_acc: 0.8632
------------------------------------
```

We see that although this network can classify MNIST data with high accuracy, but it fails to classify more complex images like FASHION_MNIST.

i) **Exercise:** Can adding more layers improve the accuracy? How about adding more nodes, epochs, and other setting?

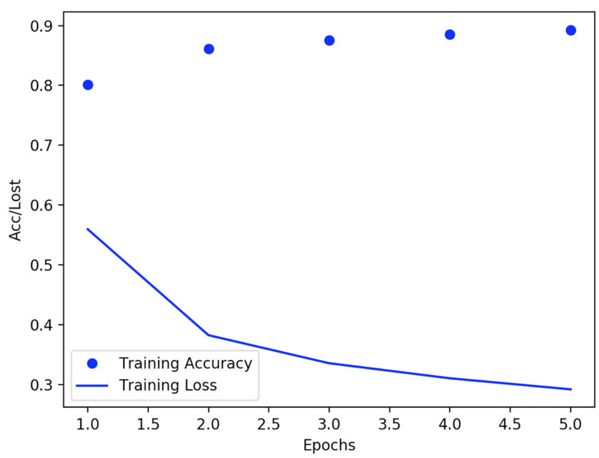
Basic Neural Network has its limitation in classifying images. An image processing technique, Image Convolution, is required to understand an image better. We will learn about the Convolution Neural Network in the next lesson.

j) Using the history object returned from the fit() function, we can plot the accuracy and loss over the epochs:

```
# list all data in history
print(history.history.keys())

```

```
# Plot the Learning curve
import matplotlib.pyplot as plt
history_dict = history.history
acc_values = history_dict['accuracy']
loss_values = history_dict['loss']
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, acc_values, 'bo', label='Training Accuracy')
plt.plot(epochs, loss_values, 'b', label='Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Acc/Lost')
plt.legend()
plt.show()
```

You should see a chat similar to the following:



<u>Activity wrap-up:</u>

We learn

- ❑ Fashion-MNIST Dataset
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Define, compile and train a model
- ❑ Evaluate the model
- ❑ Tuning the model by changing the batch sizes and epoch

Activity 3 –Classifying Fashion with CNN

In this activity, we will learn:

- ❑ Fashion-MNIST Dataset
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Define, compile and train a simple CNN model
- ❑ Evaluate the model
- ❑ Improving the model with hidden CNN layers

1. Fashion-MNIST Dataset

a) In this activity, we will use the same fashion-MNIST dataset. Instead of using a regular deep neural network, we will use a Convolution Neural Network instead.

b) Go to https://colab.research.google.com/ and start a new notebook.

c) Load up the necessary libraries and the Fashion-MNIST dataset.

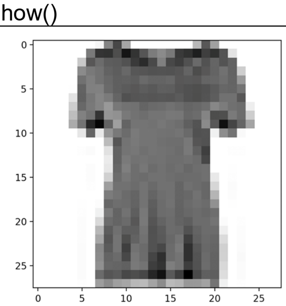
```
from keras import models
from keras import layers
from keras.datasets import fashion_mnist
from keras.utils import to_categorical
from keras import regularizers

# Load the MNIST data
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
print('train_images shape:', train_images.shape)
print('test_images shape:', test_images.shape)
print('train_labels shape:', train_labels.shape)
```

Running the above loads the dataset and prints the shape of the input and output components of the train, test and train labels and test splits of images.

```
train_images shape: (60000, 28, 28)
test_images shape: (10000, 28, 28)
train_labels shape: (60000,)
```

d) The images are grayscale with a black background (0 pixel value) and the handwritten digits in white (pixel values near 255). The images are easier to review when we reverse the colours and plot the background as white and the handwritten digits in black.

```
# Display a sample
extract_image = 10
digit = train_images[extract_image]
label = train_labels[extract_image]
print("Label =",label)
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```



e) Unlike the previous activity, for CNN, we will process the image in 2D. We will reshape the 2D images into a 3D array of images. Due to all the weights computation, we will need to normalise the pixel value by dividing them by 255.

```
# Process the data for the usage of ANN
train_images = train_images.reshape((60000, 28, 28, 1)) # Flatten the image
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28,28,1))
test_images = test_images.astype('float32') / 255
```

f) We then use keras utility function to_categorical() function convert an array of labelled data(from 0 to 9) to one-hot vector. For an in-depth discussion on one-hot encoding, please refer to https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/

```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

g) Now we are ready to define our convolutional neural network.

```
# Create the network
network = models.Sequential()
network.add(layers.Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)))
network.add(layers.Flatten())
network.add(layers.Dense(10, activation='softmax'))
```

We first add a Conv2D layer with 32 filters, kernel size of 3 x 3. We use the 'relu' activation function. Next, we flatten() the output before passing to the final Dense layer.

h) We will continue to compile, train and evaluate the model to get a feel of the accuracy:

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

history = network.fit(train_images, train_labels, validation_split=0.2, epochs=5, batch_size=128,
verbose=2)


test_loss, test_acc = network.evaluate(test_images, test_labels, verbose=2)
print('test_acc:', test_acc)
```

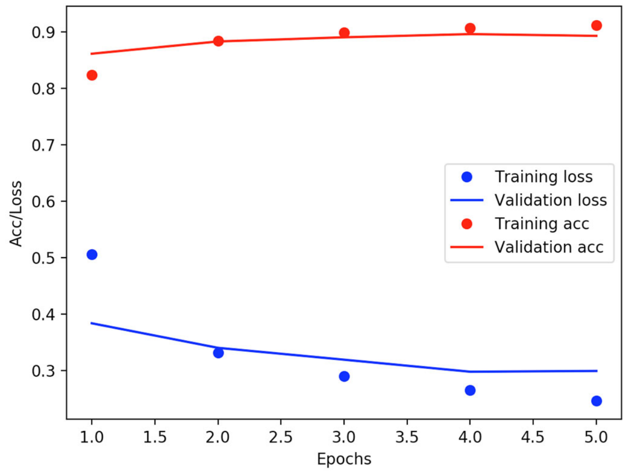
Notice in this case, we further split the training dataset into 80/20 for validation after every batch. We can also reduce the amount of information output by the specifying the verbose parameter to 2.

i) Run the code. You should see something similar to the following:

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/5
2020-03-05 20:20:15.295253: I tensorflow/core/platform/cpu_feature_guard.c
 - 7s - loss: 0.5054 - acc: 0.8244 - val_loss: 0.3834 - val_acc: 0.8616
Epoch 2/5
 - 7s - loss: 0.3308 - acc: 0.8840 - val_loss: 0.3398 - val_acc: 0.8833
Epoch 3/5
 - 7s - loss: 0.2894 - acc: 0.8993 - val_loss: 0.3188 - val_acc: 0.8908
Epoch 4/5
 - 7s - loss: 0.2644 - acc: 0.9075 - val_loss: 0.2974 - val_acc: 0.8964
Epoch 5/5
 - 7s - loss: 0.2462 - acc: 0.9124 - val_loss: 0.2988 - val_acc: 0.8932
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
test_acc: 0.8907
```

k) Using the history object returned from the fit() function, we can plot the accuracy and loss over the epochs:

```
# list all data in history
print(history.history.keys())

# Plot the Learning curve
import matplotlib.pyplot as plt
history_dict = history.history
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.plot(epochs, acc_values, 'ro', label='Training acc')
plt.plot(epochs, val_acc_values, 'r', label='Validation acc')
plt.xlabel('Epochs')
plt.ylabel('Acc/Loss')
plt.legend()
plt.show()
```



j) **Exercise:** Can adding more convolution layers to improve the accuracy? How about adding more filters, changing kernel size, epochs, and other setting?

Activity wrap-up:
We learn how to
- ❑ Fashion-MNIST Dataset
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Define, compile and train a simple CNN model
- ❑ Evaluate the model
- ❑ Improving the model with hidden CNN layers

Activity 4 –Classifying Pokemon with CNN

In this activity, we will learn:

- ❑ Custom Dataset - Pokemon
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Define, compile and train a CNN model
- ❑ Evaluate the model

1. Google Colab

As this series of activities will be very compute intensive, we will use colab's GPU instance instead of local laptop. Go to https://colab.research.google.com/ and start a new notebook.

2. Pokemon Dataset

a) 5 Pokemon characters were scrapped from the internet. 250 images for each character. We will be using this dataset to see how our improved CNN performs.

b) In order for the notebook to access the dataset, you can copy the dataset to your Google drive and mount it..

```
from google.colab import drive
drive.mount('/content/drive')
```

Follow the instruction on the screen to authorise Colab accessing your drive. On your Google Drive, you can access you files prepend with "/content/drive/My Drive/"

c) Next we will import the necessary libraries and initialise some constants and variables. Will rescale the image to 224 by 244 (this is the dimension used for the activity, so we just want to be consistent to make comparison)

```
import cv2
import numpy as np
import random
import os
from imutils import paths
from keras.preprocessing.image import img_to_array
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from keras import models
from keras import layers

# initialize the data and labels
data = []
labels = []

dataset_path = "/content/drive/My Drive/DATASETS/pokemon"
IMAGE_DIMS = (224, 224, 3)
```

d) Read and format our images and labels in to numpy array .

```
# grab the image paths and randomly shuffle them
print("[INFO] loading images...")
imagePaths = sorted(list(paths.list_images(dataset_path)))
random.seed(42)
random.shuffle(imagePaths)

# loop over the input images
for imagePath in imagePaths:
   # load the image, pre-process it, and store it in the data list
   print(imagePath)
   image = cv2.imread(imagePath)
   image = cv2.resize(image, (IMAGE_DIMS[1], IMAGE_DIMS[0]))
   image = img_to_array(image)
   data.append(image)

   # extract the class label from the image path and update the
   # labels list
   label = imagePath.split(os.path.sep)[-2]
   labels.append(label)

# scale the raw pixel intensities to the range [0, 1]
data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)
print("[INFO] data matrix: {:.2f}MB".format(
   data.nbytes / (1024 * 1000.0)))
```

e) Binarize our labels using scikit-learn's LabelBinarizer()

```
# binarize the labels
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
```

The following compares the difference between One-Hot and LabelBinarizer

```
Data:  ['cold' 'cold' 'warm' 'cold' 'hot' 'hot' 'warm' 'cold' 'warm' 'hot']
Label Encoder: [0 0 2 0 1 1 2 0 2 1]
OneHot Encoder: [[ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]
 [ 0.  1.  0.]]
Label Binarizer: [[1 0 0]
 [1 0 0]
 [0 0 1]
 [1 0 0]
 [0 1 0]
 [0 1 0]
 [0 0 1]
 [1 0 0]
 [0 0 1]
 [0 1 0]]
```

f) Next, we will split our dataset into train set and test set..

```
# partition the data into training and testing splits using 80% of
# the data for training and the remaining 20% for testing
(train_images, test_images, train_labels, test_labels) = train_test_split(data,labels, test_size=0.2,
random_state=42)
```

g) Now we are ready to define our convolutional neural network.

```
# Create the network
network = models.Sequential()
network.add(layers.Conv2D(32, (3, 3), activation="relu", input_shape=train_images.shape[1:]))
network.add(layers.Flatten())
network.add(layers.Dense(100, activation='relu'))
network.add(layers.Dense(5, activation='softmax'))

network.compile(optimizer='adam',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

In this case, we experiment with a 100 nodes Dense layer just before the final layer.

h) We can now proceed to train the model with our training dataset and evaluate with our test dataset to get a feel of the accuracy

```
history = network.fit(train_images, train_labels, epochs=50, batch_size=32, verbose=2)
test_loss, test_acc = network.evaluate(test_images, test_labels, verbose=2)
print('test_acc:', test_acc)
```

i) Run the code. You should see something similar to the following. It may take up to 5-10 mins to finish 50 epochs.

```
[8]  test_loss, test_acc = network.evaluate(test_images, test_labels, verbose=2)
     print('test_acc:', test_acc)

     test_acc: 0.22077922097274236
```

Our CNN is not able to process the Pokemon dataset. It is only doing slightly better than guessing.

<u>Activity wrap-up:</u>
We learn how to
- ❑ Custom Dataset - Pokemon
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Define, compile and train a CNN model
- ❑ Evaluate the model

Activity 5 –Classifying Pokemon with pre-trained model VGG16

In this activity, we will learn:

- ❑ Custom Dataset - Pokemon
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Use a pretrained VGG16 model
- ❑ Evaluate the model

a) In this activity, we will randomly choose an image from the Pokemon dataset and use a pre-trained VGG16 model trained using imagenet to make a prediction.

b) Start a new notebook on Colab and mount your google drive.

```
from google.colab import drive
drive.mount('/content/drive')
```

c) Import the necessary libraries and initialise some constants that we will be using.

```
import cv2
import numpy as np
from keras.preprocessing import image
from keras.preprocessing.image import img_to_array
from keras.applications import vgg16
dataset_path = "/content/drive/My Drive/DATASETS/pokemon"
IMAGE_DIMS = (224, 224, 3)
```

d) As VGG16 is designed to work with 224 x 224 images, we will need to resize our sample image from the Pokemon dataset. The following code will select an image from the dataset set, resize it and normalise it.

```
# load the image
image = cv2.imread(dataset_path + '/squirtle/00000000.png')
# pre-process the image for classification
image = cv2.resize(image, (IMAGE_DIMS[0], IMAGE_DIMS[1]))
image = image.astype("float") / 255.0
image = img_to_array(image)
image = np.expand_dims(image, axis=0)
# Normalize the input image's pixel values to the range used when training the neural network
x = vgg16.preprocess_input(image)
```

e) Next we load up the model.

```
# Load Keras' VGG16 model that was pre-trained against the ImageNet database
model = vgg16.VGG16()
```

f) Now we are ready to make a prediction by calling predict().

```
# Run the image through the deep neural network to make a prediction
predictions = model.predict(x)
```

Predict() returns a list of predictions and their likelihood.

g) To decode the predictions object, we use the decode-predictions() which returns the list containing each prediction with their imagenet id, name and the likelihood:

```
# Look up the names of the predicted classes. Index zero is the results for the first image.
predicted_classes = vgg16.decode_predictions(predictions)

print("Top predictions for this image:")

for imagenet_id, name, likelihood in predicted_classes[0]:
    print("Prediction: {} - {:2f}".format(name, likelihood))
```

h) Run the code. You should see something similar to the following.

```
Top predictions for this image:
Prediction: matchstick - 0.077864
Prediction: nematode - 0.050015
Prediction: lighter - 0.031218
Prediction: digital_clock - 0.030732
Prediction: spotlight - 0.024098
```

We see that this model although pretrained with imagenet, is not quite doing correctly for our Pokemon dataset.

Activity wrap-up:

We learn

- ❑ Custom Dataset - Pokemon

- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Use a pretrained VGG16 model
- ❑ Evaluate the model

Activity 6 –Classifying Pokemon with transfer learning using VGG16

In this activity, we will learn:

- ❑ Custom Dataset - Pokemon
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Use a pretrained VGG16 model as a feature extractor
- ❑ Add our customer classify layer
- ❑ Evaluate the model

a) In this activity, we will a pre-trained VGG16 model as a feature extractor and re-trained to classify the 5 characters in our Pokemon dataset.

b) Start a new notebook on Colab and mount your google drive.

```
from google.colab import drive
drive.mount('/content/drive')
```

c) Import the necessary libraries and initialise some constants that we will be using.

```
import os
import cv2
import numpy as np
import keras
import numpy as np
import random
from imutils import paths
from keras.preprocessing import image
from keras.preprocessing.image import img_to_array
from keras.applications import vgg16
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from keras.preprocessing.image import ImageDataGenerator
from keras import models
from keras import layers

dataset_path = "/content/drive/My Drive/DATASETS/pokemon"
IMAGE_DIMS = (224, 224, 3)
BS = 32
EPOCHS=100
print("keras version %s"%keras.__version__)
print("opencv version %s"%cv2.__version__)

# initialize the data and labels
data = []
labels = []
```

d) Read and format our images and labels in to numpy array .

```
# grab the image paths and randomly shuffle them
print("[INFO] loading images...")
imagePaths = sorted(list(paths.list_images(dataset_path)))
random.seed(42)
random.shuffle(imagePaths)

# loop over the input images
for imagePath in imagePaths:
   # load the image, pre-process it, and store it in the data list
   print(imagePath)
   image = cv2.imread(imagePath)
   image = cv2.resize(image, (IMAGE_DIMS[1], IMAGE_DIMS[0]))
   image = img_to_array(image)
   data.append(image)

   # extract the class label from the image path and update the
   # labels list
   label = imagePath.split(os.path.sep)[-2]
   labels.append(label)

# scale the raw pixel intensities to the range [0, 1]
data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)
print("[INFO] data matrix: {:.2f}MB".format(
   data.nbytes / (1024 * 1000.0)))
```

e) Binarize our labels using scikit-learn’s LabelBinarizer()

```
# binarize the labels
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
```

f) Next, we will split our dataset into train set and test set..

```
# partition the data into training and testing splits using 80% of
# the data for training and the remaining 20% for testing
(train_images, test_images, train_labels, test_labels) = train_test_split(data,labels, test_size=0.2, random_state=42)
```

g) Create our model based on VGG16. We will replace the top layer with our customised Dense layer that has only 5 nodes.

```
# initialize the model
print("[INFO] compiling model...")

# Load VGG16 model without the top layers
base_layers = vgg16.VGG16(include_top=False, input_shape=IMAGE_DIMS)
#base_layers = vgg16.VGG16(include_top=False)

# Allow fine tuning to go into the convolution layers
for layer in base_layers.layers:
     layer.trainable = False

# Create the network
network = models.Sequential(base_layers.layers)
network.add(layers.Flatten())
network.add(layers.Dense(5, activation='softmax'))

network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

network.summary()
```

Our network should be like this:

```
[INFO] compiling model...
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792      
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928     
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0         
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856     
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584    
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0         
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168    
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080    
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080    
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0         
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160   
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808   
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808   
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0         
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808   
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808   
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808   
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0         
_________________________________________________________________
flatten_2 (Flatten)          (None, 25088)             0         
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 125445    
=================================================================
Total params: 14,840,133
Trainable params: 125,445
Non-trainable params: 14,714,688
_________________________________________________________________
```

h) We can now proceed to train the model with our training dataset and evaluate with our test dataset to get a feel of the accuracy:

```
history = network.fit(train_images, train_labels, epochs=50, batch_size=32, verbose=2)
test_loss, test_acc = network.evaluate(test_images, test_labels, verbose=2)
print('test_acc:', test_acc)
```

i) Run the code. You should see something similar to the following.

```
Epoch 44/50
 - 5s - loss: 3.4909 - acc: 0.7744
Epoch 45/50
 - 5s - loss: 3.4661 - acc: 0.7777
Epoch 46/50
 - 5s - loss: 3.4898 - acc: 0.7766
Epoch 47/50
 - 4s - loss: 3.4879 - acc: 0.7744
Epoch 48/50
 - 4s - loss: 3.4766 - acc: 0.7744
Epoch 49/50
 - 4s - loss: 3.4546 - acc: 0.7755
Epoch 50/50
 - 4s - loss: 3.4822 - acc: 0.7755
test_acc: 0.7099567102147387
```

Not bad, considering how fast we can perform the training. [Other techniques can be used to improve the accuracy!]

j) Let's load up an image and see how the prediction work.

```
# load the image
image = cv2.imread(dataset_path + '/pikachu/00000000.jpg')

# pre-process the image for classification
image = cv2.resize(image, (IMAGE_DIMS[0], IMAGE_DIMS[1]))
image = image.astype("float") / 255.0
image = img_to_array(image)
image = np.expand_dims(image, axis=0)
# classify the input image
print("[INFO] classifying image...")
proba = network.predict(image)[0]
idx = np.argmax(proba)
label = lb.classes_[idx]

print(label)
```

Run the code. Your result may be different but you will see your model make a prediction. You can try with different images and see the results too.

```
[INFO] classifying image...
pikachu
```

Activity wrap-up:

We learn

- ❑ Custom Dataset - Pokemon
- ❑ Load and preprocess the dataset
- ❑ Split the dataset for training and testing
- ❑ Use a pretrained VGG16 model as a feature extractor
- ❑ Add our customer classify layer
- ❑ Evaluate the model