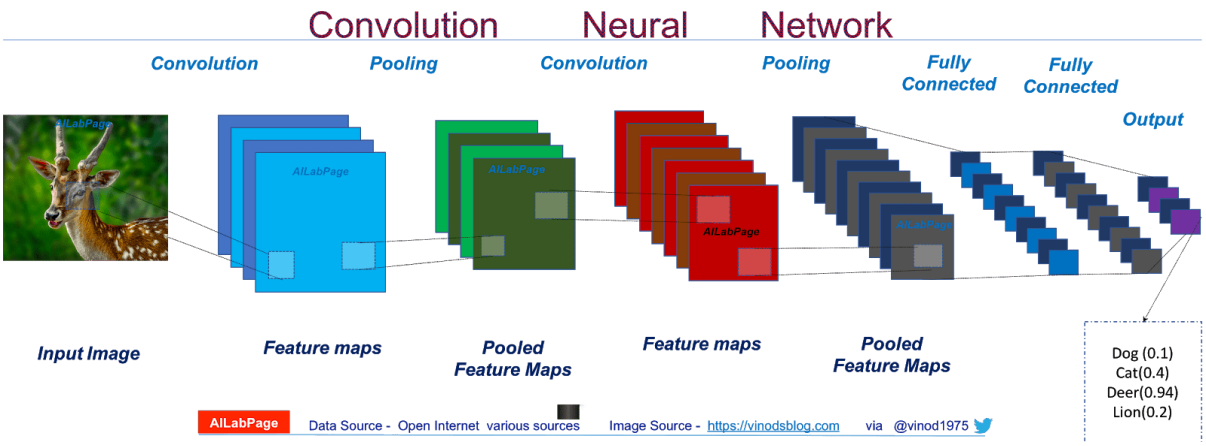
MNIST Handwritten Digit Classification Dataset



The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset.

It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9.

The task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively.

It is a widely used and deeply understood dataset and, for the most part, is "*solved*." Top-performing models are deep learning convolutional neural networks that achieve a classification accuracy of above 99%, with an error rate between 0.4 %and 0.2% on the hold out test dataset.

The below code is used to load the mnist dataset and reshape the image and scale it by dividing to 255.

```
import tensorflow as tf

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

img_rows, img_cols = 28, 28

# Reshaping the array to 4-dims so that it can work with the Keras API

x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)

x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)

input_shape = (img_rows, img_cols, 1)
```

```
# Making sure that the values are float so that we can get decimal points after division

x_train = x_train.astype('float32')

x_test = x_test.astype('float32')

# Normalizing the RGB codes by dividing it to the max RGB value.

x_train /= 255

x_test /= 255

print('x_train shape:', x_train.shape)

print('Number of images in x_train', x_train.shape[0])


print('Number of images in x_test', x_test.shape[0])
```

```
x_train shape: (60000, 28, 28, 1)
Number of images in x_train 60000
Number of images in x_test 10000
[5 0 4 ... 5 6 8]
```

Shape of mnist train and test sets.

Model Evaluation Methodology

The dataset already has a well-defined train and test dataset that we can use.

In order to estimate the performance of a model for a given training run, we can further split the training set into a train and validation dataset. Performance on the train and validation dataset over each run can then be plotted to provide learning curves and insight into how well a model is learning the problem.

The Keras API supports this by specifying the "*validation_data*" argument to the *model.fit()* function when training the model, that will, in turn, return an object that describes model performance for the chosen loss and metrics on each training epoch.

```
# Importing the required Keras modules containing model and layers

from keras.models import Sequential
```

```
from keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

# Creating a Sequential Model and adding the layers

model = Sequential()

model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

# Flattening the 2D arrays for fully connected layers

model.add(Dense(128, activation=tf.nn.relu))

model.add(Dropout(0.2))

model.add(Dense(10,activation=tf.nn.softmax))

model.compile(optimizer='adam',

        loss='sparse_categorical_crossentropy',

        metrics=['accuracy'])

model.fit(x=x_train,y=y_train, epochs=10)

print(model.evaluate(x_test, y_test))
```

```
======>..] - ETA: 3s 9696/10000 [============================>.] - ETA: 2s 972
8/10000 [============================>.] - ETA: 2s 9760/10000 [============================>.] - ETA: 2s
 9792/10000 [============================>.] - ETA: 1s 9824/10000 [============================>.] - ETA: 1
s 9856/10000 [============================>.] - ETA: 1s 9888/10000 [==========
==========>.] - ETA: 1s 9920/10000 [============================>.] - ETA: 0s
 9952/10000 [============================>.] - ETA: 0s 9984/10000 [============================>.] - ETA: 0s
10000/10000 [==============================] - 95s 10ms/step
[0.06351081106258362, 0.9847999811172485]
>>>
```

How to Develop a Baseline Model

The first step is to develop a baseline model.

This is critical as it both involves developing the infrastructure for the test harness so that any model we design can be evaluated on the dataset, and it establishes a baseline in model performance on the problem, by which all improvements can be compared.

The design of the test harness is modular, and we can develop a separate function for each piece. This allows a given aspect of the test harness to be modified or inter-changed, if we desire, separately from the rest.

We can develop this test harness with five key elements. They are the loading of the dataset, the preparation of the dataset, the definition of the model, the evaluation of the model, and the presentation of results.

Load Dataset

We know some things about the dataset. We are going to have 15 types and making it to 32* 32 to ease calculations.

```
img_rows, img_cols = 32, 32
DATASET_PATH = '15-Scene/'
one_hot_lookup = np.eye(15) # 15 classes
dataset_x = []
dataset_y = []
for category in sorted(os.listdir(DATASET_PATH)):
        print('loading category: '+str(int(category)))
        for fname in os.listdir(DATASET_PATH+category):
        img = cv2.imread(DATASET_PATH+category+'/'+fname, 2)
        img = cv2.resize(img, (img_rows,img_cols))
        dataset_x.append(np.reshape(img, [img_rows,img_cols,1]))
        dataset_y.append(int(category))
dataset_x = np.array(dataset_x)
```

Prepare Pixel Data

We know that the pixel values for each image in the dataset are unsigned integers in the range between black and white, or 0 and 255.

We do not know the best way to scale the pixel values for modeling, but we know that some scaling will be required.

A good starting point is to normalize the pixel values of grayscale images, e.g. rescale them to the range [0,1]. This involves first converting the data type from unsigned integers to floats, then dividing the pixel values by the maximum value.

```
# Reshaping the array to 4-dims so that it can work with the Keras API
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
# Making sure that the values are float so that we can get decimal points after division
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
# Normalizing the RGB codes by dividing it to the max RGB value.
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print('Number of images in x_train', x_train.shape[0])
print('Number of images in x_test', x_test.shape[0])
```

```
Using TensorFlow backend.
loading category: 0
loading category: 1
loading category: 2
loading category: 3
loading category: 4
loading category: 5
loading category: 6
loading category: 7
loading category: 8
loading category: 9
loading category: 10
loading category: 11
loading category: 12
loading category: 13
loading category: 14
[ 881  311 1359 ... 1272   55 4426]
x_train shape: (4037, 32, 32, 1)
Number of images in x_train 4037
Number of images in x_test 448
Epoch 1/10
```

Define Model

Next, we need to define a baseline convolutional neural network model for the problem.

The model has two main aspects: the feature extraction front end comprised of convolutional and pooling layers, and the classifier backend that will make a prediction.

For the convolutional front-end, we can start with a single convolutional layer with a small filter size (3,3) and a modest number of filters (32) followed by a max pooling layer. The filter maps can then be flattened to provide features to the classifier.

```
# Importing the required Keras modules containing model and layers

from keras.models import Sequential

from keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

# Creating a Sequential Model and adding the layers

model = Sequential()

model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten()) # Flattening the 2D arrays for fully connected layers

model.add(Dense(128, activation=tf.nn.relu))

model.add(Dropout(0.2))

model.add(Dense(15,activation=tf.nn.softmax))

model.compile(optimizer='adam',

        loss='sparse_categorical_crossentropy',

        metrics=['accuracy'])

model.fit(x=x_train,y=y_train, epochs=10)
```

Evaluate Model

After the model is defined, we need to evaluate it.

The training dataset is shuffled prior to being split, and the sample shuffling is performed each time, so that any model we evaluate will have the same train and test datasets in each fold, providing an apples-to-apples comparison between models.

```
"""shuffle dataset"""

p = np.random.permutation(len(dataset_x))

print(p)

dataset_x = dataset_x[p]

for i in p:

        ds_y.append(dataset_y[i])

x_test = dataset_x[:int(len(dataset_x)/10)]

y_test = ds_y[:int(len(dataset_x)/10)]

x_train = dataset_x[int(len(dataset_x)/10):]

y_train = ds_y[int(len(dataset_x)/10):
```

```
384/448 [=========================>.....] - ETA: 0s
448/448 [==============================]                    - 1s 1ms/
step
Miss Rate and Accuracy :  [1.660040625504085, 0.5200892686843872]
>>>
```