Deep Learning Project - Hieroglyphs

MBA Tech CE - 3rd Year

Project by:

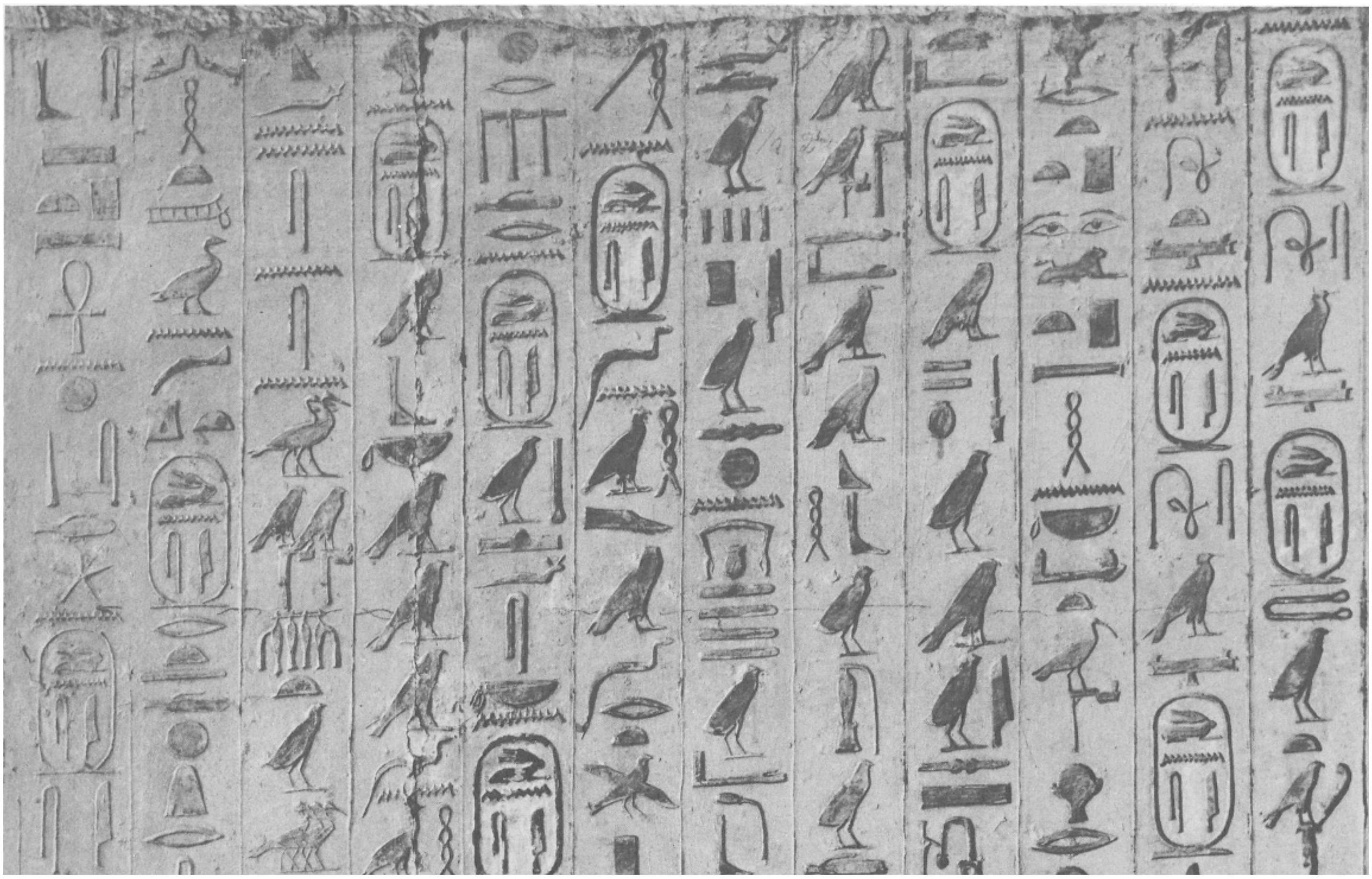
> Swapnil Singh (N041)
>
> Vaishnavi Singh (N048)
>
> Tarun Tanmay (N049)
>
> Vidhi Vaziani (N052)

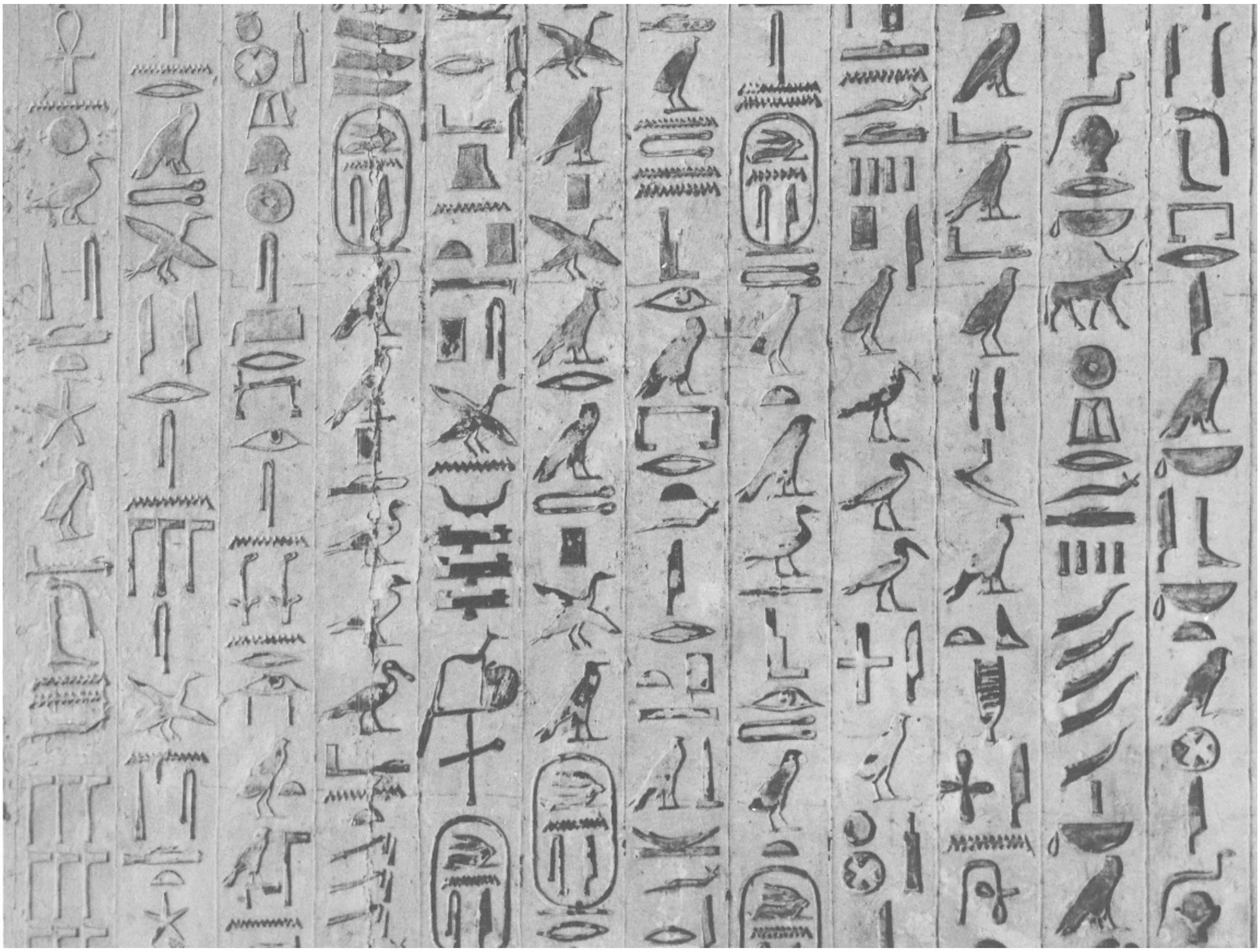
Introduction

- This project is a small part of a larger goal
- The aim of the project is to classify egyptian symbols or Hieroglyphs into the classes assigned by the Archeologist
- Other sections of the project including

> - Image processing to capture, filter, and segment the image to extract the individual symbols
> - Followed by this section where we classify the image to the annotations assigned to it
> - Then comes the concepts of NLP where we aim to translate the annotation to normal english for people to read

- Given below is an image from the Pyramid of Giza





Loading the dataset

1. **Accessing the drive for images**
2. **Loading the images**
3. **Making batches**

Vidhi Vazirani - N052

In [ ]:

```
from google.colab import drive
drive.mount("/content/drive", force_remount=True)
```

```
Mounted at /content/drive
```

In [ ]:

```
import numpy as np
from multiprocessing.pool import Pool
from keras.preprocessing import image
from keras.applications.inception_v3  import preprocess_input
```

In [ ]:

```
def loadImage(path):
    img = image.load_img(path, target_size=(299, 299))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)
    return x
```

In [ ]:

In [ ]:

```
def loadBatch(img_paths):
    with Pool(processes=8) as pool:
        imgs = pool.map(loadImage, img_paths)
        return np.vstack(imgs)
```

In [ ]:

```
def batchGenerator(img_paths, labels, batch_size):
    for i in range(0, len(img_paths), batch_size):
        batch_paths = img_paths[i:(i + batch_size)]
        batch_labels = labels[i:(i + batch_size)]
        batch_images = loadBatch(batch_paths)
        yield batch_images, batch_labels
```

Feature Extraction

Feature extraction is done using the Inception network.

The inception network works in 2 steps: Feature extraction and classification. But here, we just need the features. So we pop out the last layer of the network.

In [ ]:

```
from keras.applications.inception_v3 import InceptionV3
from sklearn.preprocessing import normalize
```

In [ ]:

```
class FeatureExtractor:
    def __init__(self):
        print("loading DeepNet (Inception-V3) ...")
        self.model = InceptionV3(weights='imagenet')

        # Initialise the model to output the second to last layer, which contains the de
eplearning featuers
        self.model.layers.pop() # Get rid of the classification layer
        self.model.outputs = [self.model.layers[-1].output]
        self.model.layers[-1].outbound_node = []

    def get_features(self, batch):
        features =  self.model.predict(batch)
        features = features.reshape(-1,features.shape[-1])
        return normalize(features, axis=1, norm='l2')
```

In [ ]:

```
import numpy as np
from os import listdir, path
from os.path import isdir, isfile, join, exists, dirname
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn import linear_model
from sklearn.externals import joblib
from urllib.request import urlopen
from zipfile import ZipFile
from io import BytesIO
import os
import cv2
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/externals/joblib/__init__.py:15: FutureWar
ning: sklearn.externals.joblib is deprecated in 0.21 and will be removed in 0.23. Please
import this functionality directly from joblib, which can be installed with: pip install
joblib. If this warning is raised when loading pickled models, you may need to re-seriali
ze those models with scikit-learn 0.21+.
  warnings.warn(msg, category=FutureWarning)
```

In [ ]:

```
dataPath          = '/content/drive/MyDrive/Dataset'
stelePath         = join(dataPath, "Manual/Preprocessed")
examplePath       = join(dataPath, "Examples")
featurePath       = "features.npy"
labelsPath        = "labels.npy"
svmPath           = "svm.pkl"
image_paths       = []
labels            = []
batch_size        = 2000
```

In [ ]:

```
print("indexing images...")
Steles = [ join(stelePath,f) for f in listdir(stelePath) if isdir(join(stelePath,f)) ]
for stele in Steles:
    imagePaths = [ join(stele,f) for f in listdir(stele) if isfile(join(stele,f)) ]
    for path in imagePaths:
        image_paths.append(path)
        labels.append(path[(path.rfind("_") + 1): path.rfind(".")])

featureExtractor = FeatureExtractor()
features = []
print("computing features...")
for idx, (batch_images, _) in enumerate(batchGenerator(image_paths, labels, batch_size))
:
    print("{}/{}".format((idx+1) * batch_size, len(labels)))
    features_ = featureExtractor.get_features(batch_images)
    features.append(features_)
features = np.vstack(features)

labels = np.asarray(labels)
print("saving features...")
np.save(featurePath, features)
np.save(labelsPath, labels)
```

```
indexing images...
loading DeepNet (Inception-V3) ...
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/incept
ion_v3/inception_v3_weights_tf_dim_ordering_tf_kernels.h5
96116736/96112376 [==============================] - 1s 0us/step
computing features...
2000/4210
4000/4210
6000/4210
saving features...
```

In [ ]:

```
features
```

Out[ ]:

```
array([[1.5229225e-04, 5.5373140e-04, 3.0603906e-04, ..., 2.1803525e-04,
        1.3289934e-04, 6.4251979e-04],
       [6.4444571e-04, 4.5223985e-04, 5.2867125e-04, ..., 5.3320249e-04,
        3.7295202e-04, 2.8969211e-04],
       [2.0304059e-04, 2.7304085e-04, 3.5118556e-03, ..., 9.1406232e-04,
        4.3884842e-04, 4.9880863e-04],
       ...,
       [5.6005852e-06, 2.2427563e-05, 8.1763294e-04, ..., 1.7655144e-05,
        3.6304511e-05, 1.3711563e-05],
       [1.7347759e-04, 2.4198461e-04, 5.2247121e-04, ..., 1.5125928e-03,
        1.8769264e-04, 2.7925041e-04],
       [8.8263077e-05, 1.8308988e-04, 7.6894701e-04, ..., 4.7692665e-05,
        1.2536222e-04, 1.5528710e-04]], dtype=float32)
```

Filtering the dataset

Removing images that do not have a class associated with it.

Splitting the dataset into training and testing samples (80%-20%)

In [ ]:

```
tobeDeleted = np.nonzero(labels == "UNKNOWN")
features = np.delete(features,tobeDeleted, 0)
labels = np.delete(labels,tobeDeleted, 0)
numImages = len(labels)
trainSet, testSet, trainLabels, testLabels = train_test_split(features, labels, test_siz
e=0.20, random_state=42)
```

Training basic Classifiers

Training some machine learning classifers like Logistic Regression and XGBoost.

Training a deep learning classifier - MLPClassifier.

The above classifiers yield low accuracy scores.

In [ ]:

```
#Logistic regression is a classification algorithm, used when the value of the target var
iable is categorical in nature.
# most commonly used when the output variable is categorical in nature
print("training SVM...")
if 0: # optional; either train 1 classifier fast, or search trough the parameter space b
y training multiple classifiers to sqeeze out that extra 2%
    clf = linear_model.LogisticRegression(C=10000) #This class implements regularized lo
gistic regression
else:
    svr = linear_model.LogisticRegression(max_iter=10000)
    parameters = {'C':[0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000]}
    clf = GridSearchCV(svr, parameters, n_jobs=8) #Exhaustive search over specified para
meter values for an estimator.
                                                  #Important members are fit, predict.
                                                   #GridSearchCV implements a "fit" and
a "score" method
clf.fit(trainSet, trainLabels) # fitting the training set to the above classifier

print(clf)
print("finished training! saving...")
joblib.dump(clf, 'clf.pkl', compress=1) # saving the model

prediction = clf.predict(testSet) #predict() function enables us to predict the labels of
the data values on the basis of the trained model.
accuracy = np.sum(testLabels == prediction) / float(len(prediction)) # calculating the a
ccuracy reflected by the above classifier manually

# for idx, pred in enumerate(prediction):
#     print("%-5s --> %s" % (testLabels[idx], pred))
print("accuracy = {}%".format(accuracy*100))
```

```
training SVM...
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:667: UserWarning
: The least populated class in y has only 1 members, which is less than n_splits=5.
  % (min_groups, self.n_splits)), UserWarning)
```

```
GridSearchCV(cv=None, error_score=nan,
             estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
```

```
                                                fit_intercept=True,
                                                intercept_scaling=1, l1_ratio=None,
                                                max_iter=10000, multi_class='auto',
                                                n_jobs=None, penalty='l2',
                                                random_state=None, solver='lbfgs',
                                                tol=0.0001, verbose=0,
                                                warm_start=False),
             iid='deprecated', n_jobs=8,
             param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
finished training! saving...
accuracy = 64.56009913258984%
```

In [ ]:

```
from xgboost import XGBClassifier
```

In [ ]:

```
#XGBoost is a powerful machine learning algorithm especially where speed and accuracy are
concerned
xgb = XGBClassifier()
xgb.fit(trainSet, trainLabels)  # fitting the training set to the above classifier

joblib.dump(xgb, svmPath, compress=1)  # saving the model

prediction = xgb.predict(testSet) #predict() function enables us to predict the labels of
the data values on the basis of the trained model.
accuracy = np.sum(testLabels == prediction) / float(len(prediction))  # calculating the
accuracy reflected by the above classifier manually

# for idx, pred in enumerate(prediction):
#     print("%-5s --> %s" % (testLabels[idx], pred))
print("accuracy = {}%".format(accuracy*100))
```

```
accuracy = 68.02973977695167%
```

In [ ]:

```
from sklearn.neural_network import MLPClassifier
```

In [ ]:

```
#Multi-layer Perceptron classifier.

# multilayer perceptron (MLP) is a class of feedforward artificial neural network (ANN)
mlp = MLPClassifier(max_iter=1000, hidden_layer_sizes=(1000, 1000, 500, 171) , alpha=0.0
001, solver='adam') #alpha= regularization term ,adam' refers to a stochastic gradient-b
ased optimizer
#The default solver 'adam' works pretty well on relatively large datasets
#(with thousands of training samples or more) in terms of both training time and validati
on score.
# For small datasets, however, 'lbfgs' can converge faster and perform better.
mlp.fit(trainSet, trainLabels)

joblib.dump(mlp, "mlp.pkl", compress=1) # saving

prediction = mlp.predict(testSet) #predict() function enables us to predict the labels of
the data values on the basis of the trained model.
accuracy = np.sum(testLabels == prediction) / float(len(prediction)) # calculating accur
acy manually

# for idx, pred in enumerate(prediction):
#     print("%-5s --> %s" % (testLabels[idx], pred))
print("accuracy = {}%".format(accuracy*100))
```

```
accuracy = 64.80793060718712%
```

In [ ]:

```
import os
```

Testing XGBoost

Since XGBoost gave the highest accuracy, as seen above, we can test using some samples and check the accuracy.

In [ ]:

```
inputPath = examplePath
if isdir(inputPath):
    imagePaths = [join(inputPath, f) for f in listdir(inputPath) if f.endswith(('.png',
'.jpg'))]
else:
    imagePaths = [inputPath,]

print("loading images...")
Images = loadBatch(imagePaths)
print("loading SVM model...")
clf = joblib.load(svmPath);

print("Extracting features, this may take a while for large collections of images...") #
should probably use batches for this as well
extractor = FeatureExtractor() #This class allows you to extract features of an image via
a pre-trained model and re-train that model with new data.

features  = extractor.get_features(Images) #Extract set of features from a time series(m
atrix of dataframe).

classes = xgb.classes_
print("Predicting the Hieroglyph type...")
prob = np.array(xgb.predict_proba(features))
top5_i = np.argsort(-prob)[:,0]
top5_s = np.array([prob[row,top5_i[row]] for row, top5_i_row in enumerate(top5_i)])
top5_n = classes[top5_i]

print("{:<25} ::: {}".format("image name", "top 5 best matching hieroglyphs"))
for idx, path in enumerate(imagePaths):
    print("{:<25} --> {}".format(os.path.basename(path), top5_n[idx]))
```

```
loading images...
loading SVM model...
Extracting features, this may take a while for large collections of images...
loading DeepNet (Inception-V3) ...
Predicting the Hieroglyph type...
image name                ::: top 5 best matching hieroglyphs
200000_S29.png            --> S29
200001_V13.png            --> V13
200002_V13.png            --> V13
200003_G43.png            --> G43
200004_D21.png            --> D21
200005_O50.png            --> O50
200006_X1.png             --> X1
200007_M23.png            --> M23
200008_G43.png            --> G43
200009_S29.png            --> S29
200010_V13.png            --> V13
200011_M23.png            --> M23
200012_G43.png            --> G43
200013_D21.png            --> D21
200014_O50.png            --> O50
200015_V13.png            --> V13
200016_G43.png            --> G43
```

Preparing data for CNN

Swapnil Singh - N041

In [ ]:

```
# importing the required libraries for reading the images
from keras.preprocessing.image import ImageDataGenerator
import cv2
```

Augmenting the data

In [ ]:

```
# defining a function for augmenting and storing the images.
def augment_data(file_dir, n_generated_samples, save_to_dir, taking):

    data_gen = ImageDataGenerator(rotation_range=10,
                                  width_shift_range=0.1,
                                  height_shift_range=0.1,
                                  shear_range=0.1,
                                  brightness_range=(0.3, 1.0),
                                  horizontal_flip=True,
                                  vertical_flip=True,
                                  fill_mode='nearest'
                                 )
# defining the type of augmentation we want.
# we defined a rotation range of 10, width shift of 0.1, height shift of 0.1, shear range
of 0.1, brightness range of 0.3-1.0
# horizontal flip and vertical flips were set to true.

    for filename in taking:
            # load the image
            image = cv2.imread(file_dir + '/' + filename)
            # reshape the image
            image = image.reshape((1,)+image.shape)
            # prefix of the names for the generated sampels.
            save_prefix = 'aug_' + filename[:filename.rfind('.')]
            print(save_prefix)
            # generate 'n_generated_samples' sample images
            i=0
            for batch in data_gen.flow(x=image, batch_size=1, save_to_dir=save_to_dir,
                                          save_prefix=save_prefix, save_format='png
'):
                i += 1
                if i > n_generated_samples:
                    break
# in the above code we are reading the images, resizing them, and then augmenting and sav
ing the images with the prefic aug_
```

In [ ]:

```
# DON'T RUN THIS CELL AGAIN.
augmented_data_path = '/content/drive/MyDrive/GlyphDataset/Augmented_hieroglyphs'
# declaring the augmentation path

# augment data for the examples with label equal to 'yes' representing tumurous examples
augment_data(file_dir='/content/drive/MyDrive/GlyphDataset/hieroglyphs', n_generated_sam
ples=10, save_to_dir=augmented_data_path, taking=taking)
# calling the function for saving the augmented images in the given path, here we have ma
de 10 images for each image passed
```

In [ ]:

```
# declaring paths
image_paths1       = []
labels1            = []
features1          = []
batch_size         = 2000
```

Manually Preprocessed Data

In [ ]:

```
# storing the path for images and storing lables in the declared arrays
print("indexing images...")
Steles = [ join(stelePath,f) for f in listdir(stelePath) if isdir(join(stelePath,f)) ]
for stele in Steles:
    imagePaths1 = [ join(stele,f) for f in listdir(stele) if isfile(join(stele,f)) ]
    for path in imagePaths1:
        image_paths1.append(path)
        #print(path)
        labels1.append(path[(path.rfind("_") + 1): path.rfind(".")])
for filename in os.listdir("/content/drive/MyDrive/Dataset/Augmented"):
    image_paths1.append(join('/content/drive/MyDrive/Dataset/Augmented', filename))
    labels1.append(path[(path.rfind("_") + 1): path.rfind(".")])
print(len(labels1))
```

```
indexing images...
63680
```

In [ ]:

```
# storing the path for images and storing lables in the declared arrays for the second te
st dataset
print("indexing images...")
image_paths2 = []
labels2 = []
for filename in taking:
    image_paths2.append(join('/content/drive/MyDrive/GlyphDataset/hieroglyphs', filename
))
    labels2.append(filename[:filename.rfind(".")])
for filename in os.listdir('/content/drive/MyDrive/GlyphDataset/Augmented_hieroglyphs'):
    image_paths2.append(join('/content/drive/MyDrive/GlyphDataset/Augmented_hieroglyphs'
, filename))
    labels2.append(filename[4 : (filename.rfind("_")-2)])
print(len(labels2))
```

```
indexing images...
1967
```

In [ ]:

```
taking = []
```

In [ ]:

```
# adding non augmented images to the dataset
for filename in os.listdir('/content/drive/MyDrive/GlyphDataset/hieroglyphs'):
  if filename[:filename.rfind('.')] in labels1:
    taking.append(filename)
```

In [ ]:

```
len(np.unique(labels1))
```

Out[ ]:

```
171
```

In [ ]:

```
# removing images whose class is not known
tobeDeleted = np.nonzero(labels1 == "UNKNOWN") # Remove the Unknown class from the datab
ase
image_paths1 = np.delete(image_paths1,tobeDeleted, 0)
labels1 = np.delete(labels1,tobeDeleted, 0)
numImages = len(labels1)
```

In [ ]:

```
# lable encoding and then tranforming the labels.
label_encoder = LabelEncoder()
label_encoder.fit(labels1)
labels2_encoded = label_encoder.transform(labels2)
```

In [ ]:

```
# checking the encoded labels
labels2_encoded
```

Out[ ]:

```
array([150, 143, 145, ...,  81,  81,  81])
```

In [ ]:

```
# checking the classes in encoded dataset
np.unique(labels2_encoded)
```

Out[ ]:

```
array([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,
        13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,
        26,  27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,
        39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
        52,  53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,
        65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  76,  77,  78,
        79,  80,  81,  82,  84,  85,  86,  87,  88,  89,  90,  91,  92,
        93,  94,  95,  96,  97,  99, 100, 101, 102, 104, 105, 106, 107,
       108, 110, 111, 112, 113, 114, 115, 116, 117, 118, 120, 121, 122,
       123, 124, 125, 126, 127, 128, 129, 131, 132, 133, 134, 135, 136,
       137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149,
       150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162,
       163, 164, 165, 166, 167, 168, 169, 170])
```

In [ ]:

```
# manually one hot encoding images for the second dataset
labels2_one_hot_encoded = []
for i in labels2_encoded:
  a = np.zeros(171,)
  a[i] = 1
  labels2_one_hot_encoded.append(a)
```

In [ ]:

```
# verifying the count of the classes
target = label_encoder.classes_
len(target)
```

Out[ ]:

```
171
```

In [ ]:

```
# one hot encoding the labels of the training dataset
```

```
labels1 = to_categorical(labels1)
```

In [ ]:

```
#iterating in training set of data and reading and resizing the images
rawImages2 = []
for i in image_paths2:
    img = cv2.imread(i, cv2.IMREAD_GRAYSCALE)
    #stores the raw pixel values of this image after resizing
    pixels = getPixels(img,(32,32))
    #stores the raw pixel values of images
    rawImages2.append(pixels)
```

In [ ]:

```
rawImages2 = np.asarray(rawImages2) # converting to numpy array
labels2_one_hot_encoded = np.asarray(labels2_one_hot_encoded) # converting to numpy arra
y
```

In [ ]:

```
labels2_one_hot_encoded.shape # checking the shape of the encoded labels
```

Out[ ]:

```
(1967, 171)
```

In [ ]:

```
np.save('/content/drive/MyDrive/Dataset/rawimages_ejypt.npy', rawImages1) # saving the d
ataset read to save time in the future.
 # next time use np.load('/content/drive/MyDrive/Dataset/rawimages_ejypt.npy')
```

Convolution Neural Network (CNN)

Tarun Tanmay - N049

In [ ]:

```
from keras.utils import to_categorical
```

Preparing Training and testing data

In [ ]:

```
X_train1,X_test1,y_train1,y_test1 = train_test_split(rawImages1,labels1,test_size = 0.2,
random_state=42)
```

In [ ]:

```
X_train1 = X_train1.reshape(-1, 32, 32, 1)
X_test1 = X_test1.reshape(-1, 32, 32, 1)
```

In [ ]:

```
rawImages2 = rawImages2.reshape(-1, 32, 32, 1)
rawImages2 = rawImages2 / 255.0
```

In [ ]:

```
X_train1 = X_train1 / 255.0
X_test1 = X_test1 / 255.0
```

In [ ]:

```
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
```

In [ ]:

```
from tensorflow.keras import layers, models, utils, datasets
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, Dropout, BatchNormalizati
on
from keras.layers import LeakyReLU
```

Custom CNN

In [ ]:

```
#creating A Convolutional Neural Network
model1=models.Sequential()                                    # sequential neural netwo
rk
model1.add(Conv2D(32, (5, 5), input_shape = (32, 32, 1)))     # 5*5 size filter, input l
ayer has 32 neurons
model1.add(LeakyReLU(alpha=0.1))                              # alpha means learning rat
e
model1.add(MaxPooling2D(pool_size = (2, 2)))                  # max-pooling
model1.add(Conv2D(128, (5, 5)))
model1.add(LeakyReLU(alpha=0.1))
model1.add(Conv2D(64, (5, 5)))
model1.add(LeakyReLU(alpha=0.1))
model1.add(Conv2D(32, (5, 5)))
model1.add(LeakyReLU(alpha=0.1))
model1.add(MaxPooling2D(pool_size = (2, 2)))
model1.add(Flatten())                                         # flattening o/p of cnn
model1.add(Dense(1000))                                       # Dense layer (Fully conn
ected layer)
model1.add(LeakyReLU(alpha=0.1))
model1.add(Dropout(0.5))                                      # Dropout to reducde overf
itting
model1.add(Dense(500))
model1.add(LeakyReLU(alpha=0.1))
model1.add(Dropout(0.5))
model1.add(Dense(250))
model1.add(LeakyReLU(alpha=0.1))
model1.add(Dense(171, activation = 'softmax'))                # softmax activation funct
ion used for measuring predictions
#output layer has 171 neurons, which represnet the 171 classes of the dataset
model1.summary()                                              # summarising what is in
the designed model
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 128)       102528
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 10, 10, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 6, 6, 64)          204864
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 2, 2, 32)          51232
_________________________________________________________________
```

```
leaky_re_lu_3 (LeakyReLU)    (None, 2, 2, 32)          0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 1, 1, 32)          0
_________________________________________________________________
flatten (Flatten)            (None, 32)                0
_________________________________________________________________
dense (Dense)                (None, 1000)              33000
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 1000)              0
_________________________________________________________________
dropout (Dropout)            (None, 1000)              0
_________________________________________________________________
dense_1 (Dense)              (None, 500)               500500
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 500)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_2 (Dense)              (None, 250)               125250
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 250)               0
_________________________________________________________________
dense_3 (Dense)              (None, 171)               42921
=================================================================
Total params: 1,061,127
Trainable params: 1,061,127
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

```
#we use the adam optimizer to handle sparse gradient for nosiy problems, it combines the
benefits of rmsprop and adagrad algorithms
model1.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
               #we use categorical_entropy
              #If we use this loss, we will train a CNN to output a probability over th
e C classes for each image.
              #It is used for multi-class classification

history1 = model1.fit(X_train1, y_train1, epochs= 50, #training the model for 50 epochs
                    validation_split = 0.1)
# using the default batch size of 32
```

```
Epoch 1/50
1429/1429 [==============================] - 14s 5ms/step - loss: 0.5253 - accuracy: 0.93
25 - val_loss: 0.3001 - val_accuracy: 0.9404
Epoch 2/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2916 - accuracy: 0.938
9 - val_loss: 0.2266 - val_accuracy: 0.9531
Epoch 3/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2191 - accuracy: 0.950
2 - val_loss: 0.2127 - val_accuracy: 0.9547
Epoch 4/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1977 - accuracy: 0.954
4 - val_loss: 0.1667 - val_accuracy: 0.9646
Epoch 5/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1604 - accuracy: 0.961
3 - val_loss: 0.1433 - val_accuracy: 0.9709
Epoch 6/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1458 - accuracy: 0.965
6 - val_loss: 0.1757 - val_accuracy: 0.9652
Epoch 7/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1448 - accuracy: 0.966
8 - val_loss: 0.1450 - val_accuracy: 0.9736
Epoch 8/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1315 - accuracy: 0.967
```

```
8 - val_loss: 0.1184 - val_accuracy: 0.9774
Epoch 9/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1394 - accuracy: 0.966
4 - val_loss: 0.1246 - val_accuracy: 0.9736
Epoch 10/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1221 - accuracy: 0.969
4 - val_loss: 0.1133 - val_accuracy: 0.9797
Epoch 11/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1448 - accuracy: 0.968
6 - val_loss: 0.2061 - val_accuracy: 0.9596
Epoch 12/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1496 - accuracy: 0.967
1 - val_loss: 0.1346 - val_accuracy: 0.9760
Epoch 13/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1034 - accuracy: 0.974
9 - val_loss: 0.1532 - val_accuracy: 0.9732
Epoch 14/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1316 - accuracy: 0.969
5 - val_loss: 0.1718 - val_accuracy: 0.9715
Epoch 15/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1541 - accuracy: 0.967
4 - val_loss: 0.1192 - val_accuracy: 0.9789
Epoch 16/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1088 - accuracy: 0.973
3 - val_loss: 0.1172 - val_accuracy: 0.9787
Epoch 17/50
1429/1429 [==============================] - 7s 5ms/step - loss: 3.6909 - accuracy: 0.921
7 - val_loss: 0.4414 - val_accuracy: 0.9376
Epoch 18/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.4344 - accuracy: 0.933
5 - val_loss: 0.3276 - val_accuracy: 0.9406
Epoch 19/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.3686 - accuracy: 0.936
9 - val_loss: 0.3165 - val_accuracy: 0.9404
Epoch 20/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.3356 - accuracy: 0.937
5 - val_loss: 0.2768 - val_accuracy: 0.9421
Epoch 21/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.3583 - accuracy: 0.938
2 - val_loss: 0.4114 - val_accuracy: 0.9376
Epoch 22/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2693 - accuracy: 0.944
6 - val_loss: 0.2467 - val_accuracy: 0.9610
Epoch 23/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2832 - accuracy: 0.949
9 - val_loss: 0.3593 - val_accuracy: 0.9392
Epoch 24/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.3753 - accuracy: 0.936
3 - val_loss: 0.2503 - val_accuracy: 0.9569
Epoch 25/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2299 - accuracy: 0.950
7 - val_loss: 0.2319 - val_accuracy: 0.9632
Epoch 26/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2041 - accuracy: 0.954
6 - val_loss: 0.2167 - val_accuracy: 0.9638
Epoch 27/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2001 - accuracy: 0.957
9 - val_loss: 0.2283 - val_accuracy: 0.9632
Epoch 28/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.5929 - accuracy: 0.942
9 - val_loss: 0.3437 - val_accuracy: 0.9427
Epoch 29/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2959 - accuracy: 0.940
6 - val_loss: 0.2292 - val_accuracy: 0.9604
Epoch 30/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2613 - accuracy: 0.947
7 - val_loss: 0.2609 - val_accuracy: 0.9553
Epoch 31/50
```

```
Epoch 31/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2185 - accuracy: 0.952
9 - val_loss: 0.1931 - val_accuracy: 0.9654
Epoch 32/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1848 - accuracy: 0.958
0 - val_loss: 0.5031 - val_accuracy: 0.9419
Epoch 33/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2599 - accuracy: 0.947
8 - val_loss: 0.1827 - val_accuracy: 0.9673
Epoch 34/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1943 - accuracy: 0.957
2 - val_loss: 0.2247 - val_accuracy: 0.9634
Epoch 35/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1781 - accuracy: 0.962
5 - val_loss: 0.1714 - val_accuracy: 0.9717
Epoch 36/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1478 - accuracy: 0.967
8 - val_loss: 0.1683 - val_accuracy: 0.9736
Epoch 37/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1744 - accuracy: 0.963
8 - val_loss: 0.1670 - val_accuracy: 0.9671
Epoch 38/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1517 - accuracy: 0.965
2 - val_loss: 0.1655 - val_accuracy: 0.9744
Epoch 39/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1475 - accuracy: 0.967
8 - val_loss: 0.1666 - val_accuracy: 0.9665
Epoch 40/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1470 - accuracy: 0.967
0 - val_loss: 0.1607 - val_accuracy: 0.9726
Epoch 41/50
1429/1429 [==============================] - 7s 5ms/step - loss: 3.9259 - accuracy: 0.926
6 - val_loss: 0.5860 - val_accuracy: 0.9376
Epoch 42/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.4212 - accuracy: 0.932
5 - val_loss: 0.2822 - val_accuracy: 0.9567
Epoch 43/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2567 - accuracy: 0.946
7 - val_loss: 0.3919 - val_accuracy: 0.9427
Epoch 44/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2598 - accuracy: 0.946
3 - val_loss: 0.2666 - val_accuracy: 0.9646
Epoch 45/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2153 - accuracy: 0.954
1 - val_loss: 0.3079 - val_accuracy: 0.9490
Epoch 46/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2804 - accuracy: 0.945
6 - val_loss: 0.3669 - val_accuracy: 0.9535
Epoch 47/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2634 - accuracy: 0.950
0 - val_loss: 0.2404 - val_accuracy: 0.9657
Epoch 48/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2046 - accuracy: 0.957
4 - val_loss: 0.1993 - val_accuracy: 0.9622
Epoch 49/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.1997 - accuracy: 0.957
7 - val_loss: 0.2294 - val_accuracy: 0.9642
Epoch 50/50
1429/1429 [==============================] - 7s 5ms/step - loss: 0.2743 - accuracy: 0.952
5 - val_loss: 0.1802 - val_accuracy: 0.9719
```

Performance evaluation

In [ ]:

```
print("Loss of the model is - " , model1.evaluate(X_test1,y_test1)[0])
```

```
print("Accuracy of the model is - " , model1.evaluate(X_test1,y_test1)[1]*100 , "%")
```

```
397/397 [==============================] - 1s 3ms/step - loss: 0.1721 - accuracy: 0.9708
Loss of the model is -  0.17208269238471985
397/397 [==============================] - 1s 3ms/step - loss: 0.1721 - accuracy: 0.9708
Accuracy of the model is -  97.07897305488586 %
```

In [ ]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, cla
ssification_report, confusion_matrix
```

In [ ]:

```
predict = model1.predict(X_test1)
```

In [ ]:

```
y_classes = [np.argmax(y, axis=None, out=None) for y in y_test1]
```

In [ ]:

```
yp_classes = [np.argmax(y, axis=None, out=None) for y in predict]
```

In [ ]:

```
accuracy_score(y_classes, yp_classes)
```

Out[ ]:

```
0.9707897015982994
```

In [ ]:

```
# micro-average will aggregate the contributions of all classes to compute the average me
tric
precision_score(y_classes, yp_classes, average='micro')
```

Out[ ]:

```
0.9707897015982994
```

In [ ]:

```
recall_score(y_classes, yp_classes, average='micro')
```

Out[ ]:

```
0.9707897015982994
```

In [ ]:

```
f1_score(y_classes, yp_classes, average='micro')
```

Out[ ]:

```
0.9707897015982994
```

In [ ]:

```
model1.save('hierogylyphs.h5')
```

In [ ]:

```
model1 = models.load_model('hierogylyphs.h5')
```

In [ ]:

```
print("Loss of the model is - " , model1.evaluate(rawImages2,labels2_one_hot_encoded)[0]
```

```
)
print("Accuracy of the model is - " , model1.evaluate(rawImages2,labels2_one_hot_encoded
)[1]*100 , "%")
```

```
62/62 [==============================] - 2s 31ms/step - loss: 6175.4155 - accuracy: 0.006
1
Loss of the model is -  6175.41552734375
62/62 [==============================] - 2s 32ms/step - loss: 6175.4155 - accuracy: 0.006
1
Accuracy of the model is -  0.6100661121308804 %
```

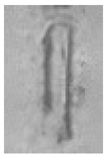
Conclusion and Summary:

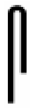
1. **The dataset was loaded from google drive.**
2. **Feature extraction was done using *Inception Neural Network.***
3. **SVM, XGBoost and MLV Classifiers were tarined and tested.**

> - **SVM = 64% (Approx)**
> - **XGBoost = 68% (Approx)**
> - **MLV Classifier = 64% (Approx)**

4. **Some Examples were tested in the XGBoost Classifier.**
5. **Data was augmented for the neural network and manually preprocessed data was loaded.**
6. **A Custom CNN was defined and trained.**

> - **Training accuracy = 95%**
> - **Testing accuracy = 97%**





As we see in the images, the two dataset have completely different image quality, the first image is a part of the image from the Piramid of Giza, where as the second image is a hand writen image. So this explains why our model fails on the second dataset.