

Data Exploration

For this project, the primary source of data was the Sign Language MNIST from Kaggle. The American Sign Language letter database of hand gestures represent a multi-class problem with 24 classes of letters (excluding J and Z which require motion).

The dataset format is patterned to match closely with the classic MNIST. Each training and test case represents a label (0-25) as a one-to-one map for each alphabetic letter A-Z (and no cases for 9=J or 25=Z because of gesture motions). The training data (27,455 cases) and test data (7172 cases) are approximately half the size of the standard MNIST but otherwise similar with a header row of label, pixel1, pixel2.... pixel784 which represents a single 28x28 pixel image with grayscale values between 0-255. The original hand gesture image data represented multiple users repeating the gesture against different backgrounds. The Sign Language MNIST data came from greatly extending the small number (1704) of the color images included as not cropped around the hand region of interest.



```
[ ] pd.DataFrame({
    'X': ['Shape', 'Different number of labels', 'Different number of labels (Sum)'],
    'Training Set': [train.shape, train.label.unique(), len(train.label.unique())],
    'Test Set': [test.shape, test.label.unique(), len(test.label.unique())],
})
```

	X	Training Set	Test Set
0	Shape	(27455, 785)	(7172, 785)
1	Different number of labels	[3, 6, 2, 13, 16, 8, 22, 18, 10, 20, 17, 19, 2...	[6, 5, 10, 0, 3, 21, 14, 7, 8, 12, 4, 22, 2, 1...
2	Different number of labels (Sum)	24	24

Proposed Methodology

System implementation turns models into functional systems or new applications using newly developed designs. This project involves reshaping the dataset, data preprocessing, testing, and training the system. These steps are necessary to ensure that the system functions correctly and accurately recognize the alphabet from the hand sign given as input.

A. System Architecture

Convolutional neural network

Convolutional neural networks (CNN) is one of the most commonly used deep learning methods to analyze visual imagery. CNN involves less preprocessing compared to other image classification algorithms. The network learns the filters that are normally hand-engineered in other systems. The use of a CNN reduces the images into a format that is easier to process while preserving features that are essential for making accurate predictions.

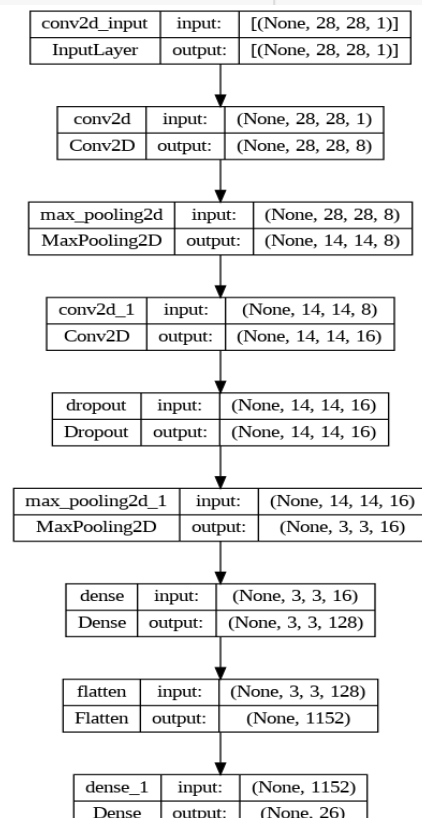
Model 1:

```
classifier = Sequential()
classifier.add(Conv2D(filters=8, kernel_size=(3,3),strides=(1,1),padding='same',
                    ,input_shape=(28,28,1),activation='relu', data_format='channels_last'))
classifier.add(MaxPooling2D(pool_size=(2,2)))
classifier.add(Conv2D(filters=16, kernel_size=(3,3),strides=(1,1),padding='same',activation='relu'))
classifier.add(Dropout(0.5))
classifier.add(MaxPooling2D(pool_size=(4,4)))
classifier.add(Dense(128, activation='relu'))
classifier.add(Flatten())
classifier.add(Dense(26, activation='softmax'))
classifier.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 8)	80
max_pooling2d (MaxPooling2D)	(None, 14, 14, 8)	0
conv2d_1 (Conv2D)	(None, 14, 14, 16)	1168
dropout (Dropout)	(None, 14, 14, 16)	0
max_pooling2d_1 (MaxPooling2D)	(None, 3, 3, 16)	0
dense (Dense)	(None, 3, 3, 128)	2176
flatten (Flatten)	(None, 1152)	0
dense_1 (Dense)	(None, 26)	29978

=====
Total params: 33,402
Trainable params: 33,402
Non-trainable params: 0



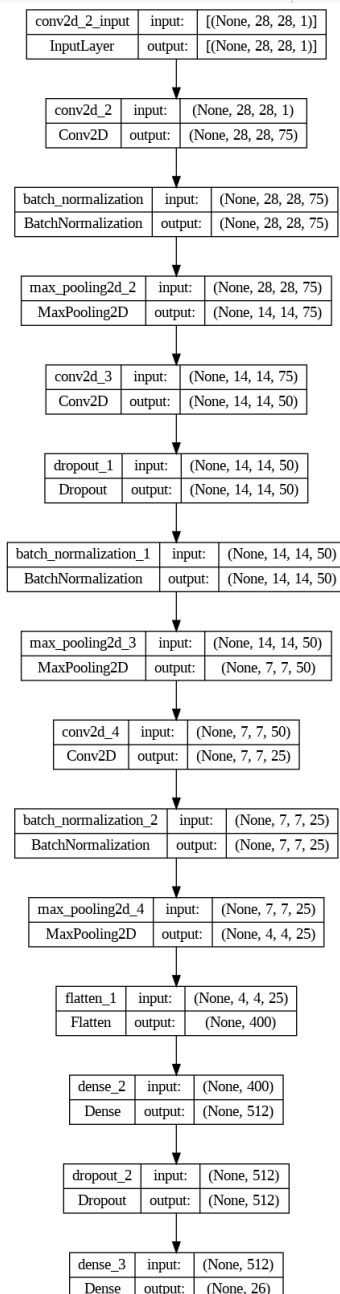
Summary of Model 1

Model 2:

```
model = Sequential()
model.add(Conv2D(75 , (3,3) , strides = 1 , padding = 'same' ,
                activation = 'relu' , input_shape = (28,28,1)))
model.add(BatchNormalization())
model.add(MaxPool2D((2,2) , strides = 2 , padding = 'same'))
model.add(Conv2D(50 , (3,3) , strides = 1 , padding = 'same' , activation = 'relu'))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(MaxPool2D((2,2) , strides = 2 , padding = 'same'))
model.add(Conv2D(25 , (3,3) , strides = 1 , padding = 'same' , activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPool2D((2,2) , strides = 2 , padding = 'same'))
model.add(Flatten())
model.add(Dense(units = 512 , activation = 'relu'))
model.add(Dropout(0.3))
model.add(Dense(units = 26 , activation = 'softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
=====		
conv2d_2 (Conv2D)	(None, 28, 28, 75)	750
batch_normalization (Batch Normalization)	(None, 28, 28, 75)	300
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 75)	0
conv2d_3 (Conv2D)	(None, 14, 14, 50)	33800
dropout_1 (Dropout)	(None, 14, 14, 50)	0
batch_normalization_1 (Batch Normalization)	(None, 14, 14, 50)	200
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 50)	0
conv2d_4 (Conv2D)	(None, 7, 7, 25)	11275
batch_normalization_2 (Batch Normalization)	(None, 7, 7, 25)	100
max_pooling2d_4 (MaxPooling2D)	(None, 4, 4, 25)	0
flatten_1 (Flatten)	(None, 400)	0
dense_2 (Dense)	(None, 512)	205312
dropout_2 (Dropout)	(None, 512)	0
dense_3 (Dense)	(None, 26)	13338
=====		
Total params: 265,075		
Trainable params: 264,775		
Non-trainable params: 300		



Summary of Model 2

The first model has two convolutional layers with 8 and 16 filters, respectively, followed by max pooling, dropout regularization, two fully connected layers, and a softmax activation function for the output layer. The first convolutional layer uses a (3,3) kernel size and the input shape is (28,28,1). The second convolutional layer uses a (3,3) kernel size with a dropout regularization rate of 0.5. The model uses the softmax cross-entropy loss function and the Adam optimizer.

The second model has three convolutional layers with 75, 50, and 25 filters, respectively, followed by batch normalization, max pooling, dropout regularization, and two fully connected layers. The first convolutional layer uses a (3,3) kernel size and the input shape is (28,28,1). The 'same' padding ensures that the output feature maps have the same spatial dimensions as the input image. The ReLU activation function introduces non-linearity into the model, allowing it to learn complex patterns and features. Batch Normalization layer normalizes the activations of the previous layer to ensure that they have zero mean and unit variance, improving the stability and performance of the model. Then, MaxPool2D layer with a pool size of (2,2), 'same' padding, and stride of 2 performs downsampling on the output feature maps of the previous layer, reducing their spatial dimensions by half. The second convolutional layer uses a (3,3) kernel size with dropout regularization of 0.2, while the third convolutional layer uses a (3,3) kernel size. Followed by a Flatten layer which flattens the output feature maps of the previous layer into a one-dimensional vector, which is then fed into the fully connected layers of the model. Dense layer with 24 units and softmax activation: This layer is the output layer of the model, with 24 units representing the 24 classes of hand signs to be recognized. The softmax activation function normalizes the output of the layer to a probability distribution over the 24 classes.

The model uses the categorical cross-entropy loss function and the Adam optimizer.

Some differences between these models include:

1. Number and size of convolutional filters: The second model has more convolutional filters and larger kernel sizes than the first model, which may allow it to extract more complex and high-level features from the input images.
2. Regularization techniques: The second model uses batch normalization in addition to dropout regularization, while the first model only uses dropout. Batch normalization can help to reduce overfitting and improve the generalization performance of the model.
3. Size of fully connected layers: The second model has larger fully connected layers with 512 units, while the first model has a smaller fully connected layer with 128 units. This may affect the capacity of the model to learn and represent complex patterns in the data.

Evaluation of the above two models:

```
metrics = classifier.evaluate(x=X_test,y=y_test,batch_size=32)
print("Loss of model 1: ",metrics[0])
print("Accuracy of model 1: ",metrics[1]*100 , "%")

225/225 [=====] - 1s 3ms/step - loss: 0.2705 - accuracy: 0.9465
Loss of model 1: 0.2704564929008484
Accuracy of model 1: 94.64584589004517 %

metrics = model.evaluate(x=X_test,y=y_test,batch_size=32)
print("Loss of model 2: ",metrics[0])
print("Accuracy of model 2: ",metrics[1]*100 , "%")

225/225 [=====] - 1s 3ms/step - loss: 0.1922 - accuracy: 0.9668
Loss of model 2: 0.19215944409370422
Accuracy of model 2: 96.68154120445251 %
```

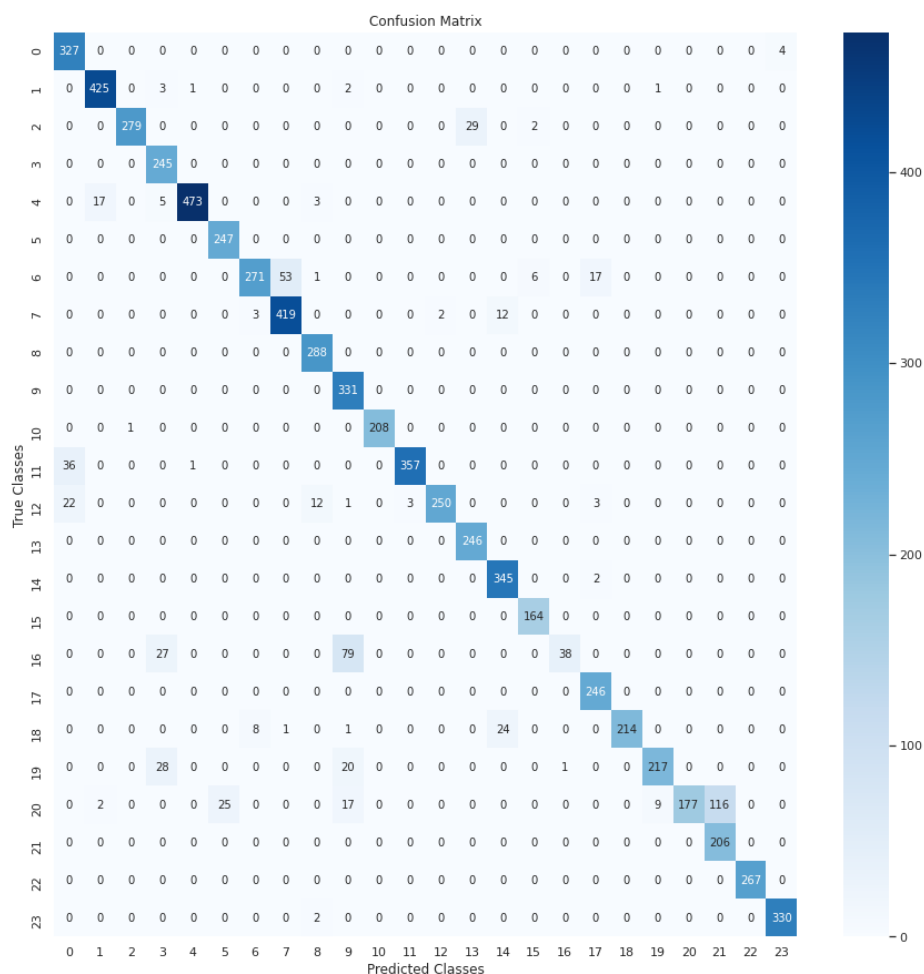
The outputs above show the evaluation metrics of the two models on a test set. The first model has a loss of 0.2705 and an accuracy of 94.65%, while the second model has a lower loss of 0.1922 and a higher accuracy of 96.68%. The evaluation metrics indicate that the second model

performs better than the first model on the test set, as it has a lower loss and a higher accuracy.

Overall we can say that, the CNN model 2 with its multiple convolutional layers, batch normalization, max pooling, dropout regularization, and dense layers, is designed to learn and extract hierarchical features from the input images, making it a powerful architecture for hand sign recognition tasks. Hence we use the second model in further implementation of OpenCV.

Testing the model with test dataset:

We generated a confusion matrix to evaluate the performance of a classification model. It uses the predicted classes and true classes of the test set data to create a matrix that shows the number of true positives, true negatives, false positives, and false negatives for each class. The matrix is displayed as a heatmap using the Seaborn library, with the predicted classes on the x-axis and the true classes on the y-axis. The diagonal elements represent correct predictions, while the off-diagonal elements represent incorrect predictions. This allows us to identify which classes are being misclassified and to adjust the model accordingly.



B. Implementation of OpenCV

We developed a Python script that uses OpenCV and Keras to implement a real-time sign language recognition system. It loads a pre-trained model using the Keras API, captures frames from the webcam using OpenCV, crops the image to isolate the region of interest (ROI), applies Gaussian blur to the ROI, resizes the image to 28x28 pixels, converts the image to grayscale,

and feeds it to the pre-trained model for prediction. The predicted output is then displayed in real-time on the captured frame. The script runs indefinitely until the user presses the 'q' key to quit.