Documentation project for:

Selected -2

Faculty Name:

Computers and artificial intelligence Helwan

TEAMWORK

	NAME	ID
1	Mohamed Ashraf Khalifa	202000729
2	Mohamed Zakaria Kamel	202000761
3	Mohamed Emadeline Abdelhamid	202000806
4	Ahmed Ashraf Taha	202000013
5	Mostafa Kamal Fahmy	202000907
6	Mostafa AlaaElden Mostafa	202000906

Name of the dataset

Arabic Handwritten Character Recognition

NUMBER OF CLASSES:

28

"ا","ب","ت","ث","ج","ح","خ", "

د,"ذ","ر","ز","س","ش","ص","ض", ط,

[illegible]

Total Number of Samples :

Tot: 16800 sample

65% Train: 10702 sample

16% Cross Validation: 2688 sample

19% Test: 3460 sample

Images Size :

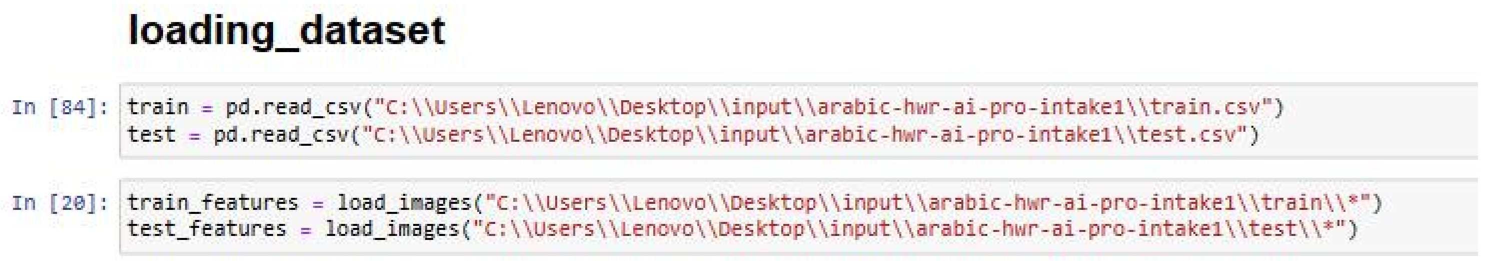
32*32*3

Dataset link :

Arabic Handwritten Characters Recognition | Kaggle

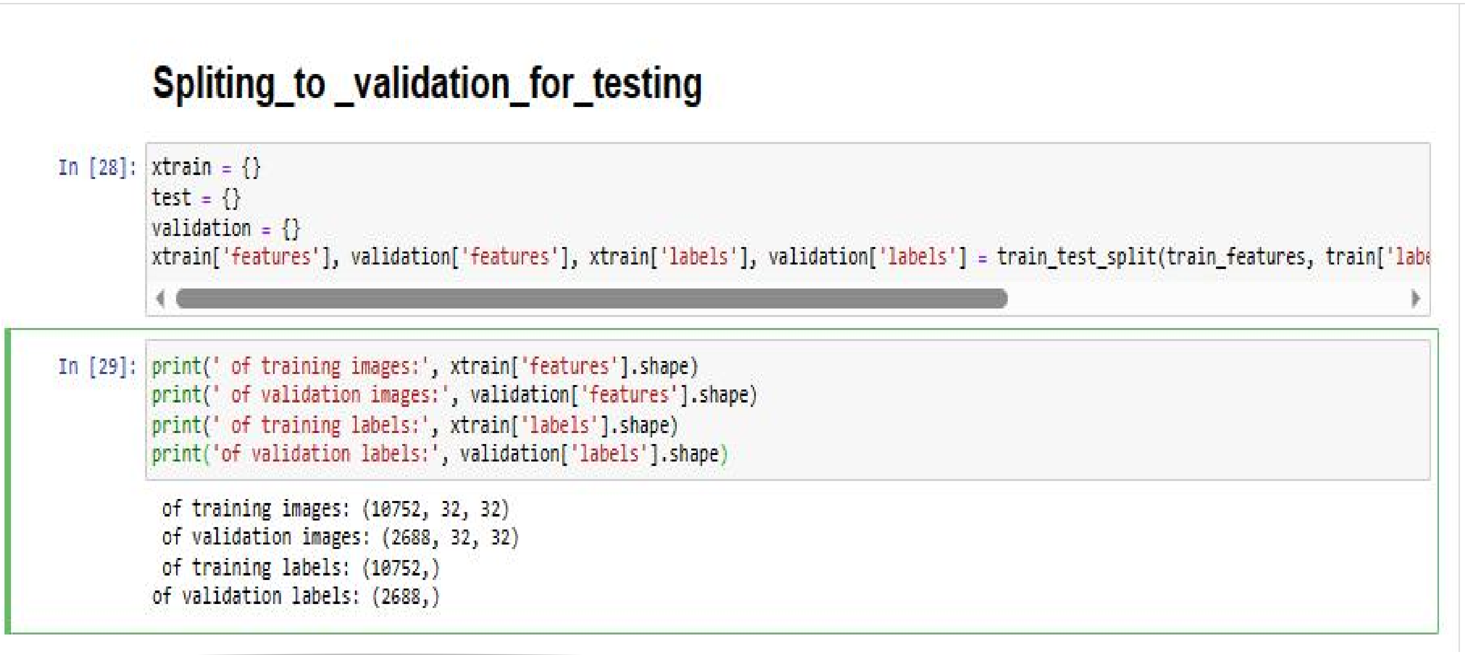
The First Step Preparing the data :

We define A variable train and validation to load the dataset then we split the data into features and target using .Iloc so we can put the features in X and the target in y.



loading_dataset

```
In [84]: train = pd.read_csv("C:\\Users\\Lenovo\\Desktop\\input\\arabic-hwr-ai-pro-intake1\\train.csv")
         test = pd.read_csv("C:\\Users\\Lenovo\\Desktop\\input\\arabic-hwr-ai-pro-intake1\\test.csv")

In [20]: train_features = load_images("C:\\Users\\Lenovo\\Desktop\\input\\arabic-hwr-ai-pro-intake1\\train\\*")
         test_features = load_images("C:\\Users\\Lenovo\\Desktop\\input\\arabic-hwr-ai-pro-intake1\\test\\*")
```



Spliting_to _validation_for_testing

```
In [28]: xtrain = {}
         test = {}
         validation = {}
         xtrain['features'], validation['features'], xtrain['labels'], validation['labels'] = train_test_split(train_features, train['labe
```

```
In [29]: print(' of training images:', xtrain['features'].shape)
         print(' of validation images:', validation['features'].shape)
         print(' of training labels:', xtrain['labels'].shape)
         print('of validation labels:', validation['labels'].shape)

          of training images: (10752, 32, 32)
          of validation images: (2688, 32, 32)
          of training labels: (10752,)
         of validation labels: (2688,)
```

Converting: we converted the output which was numbers representing the letters into actual letters so we can

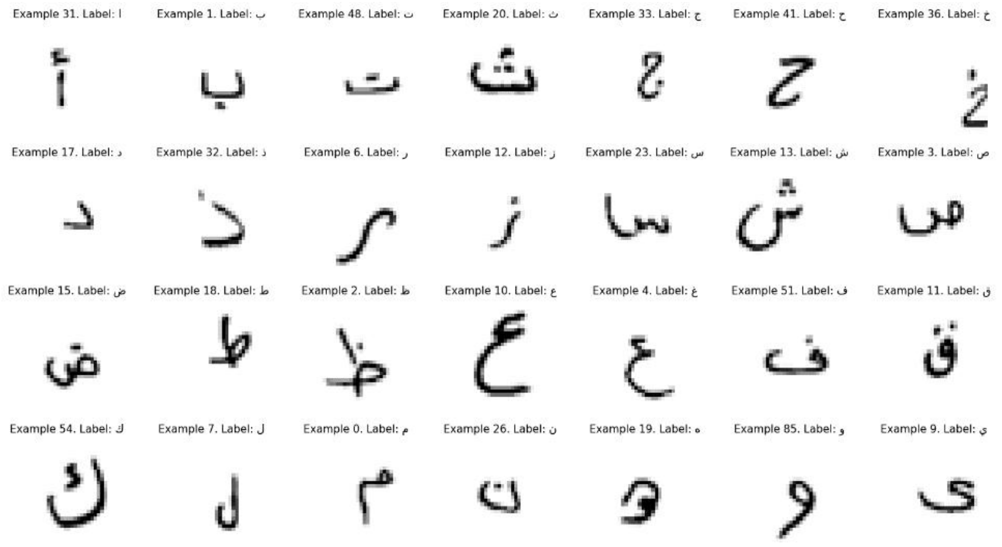
```
In [45]: characters = ["ا","ب","ت","ث","ج","ح","خ","د","ذ","ر","ز","س","ش","ص","ض","ط","ظ","ع","غ","ف","ق","ك","ل","م","ن","ه","و","ي"]

         characters_dict = dict(zip(np.arange(0,len(characters)), characters))
         characters_dict
```

```
Out[45]: {0: 'ا',
          1: 'ب',
          2: 'ت',
          3: 'ث',
          4: 'ج',
          5: 'ح',
          6: 'خ',
          7: 'د',
          8: 'ذ',
          9: 'ر',
          10: 'ز',
          11: 'س',
          12: 'ش',
          13: 'ص',
          14: 'ض',
          15: 'ط',
          16: 'ظ',
          17: 'ع',
          18: 'غ',
          19: 'ف',
          20: 'ق',
          21: 'ك',
          22: 'ل',
          23: 'م',
          24: 'ن',
          25: 'ه',
          26: 'و',
          27: 'ي'}
```

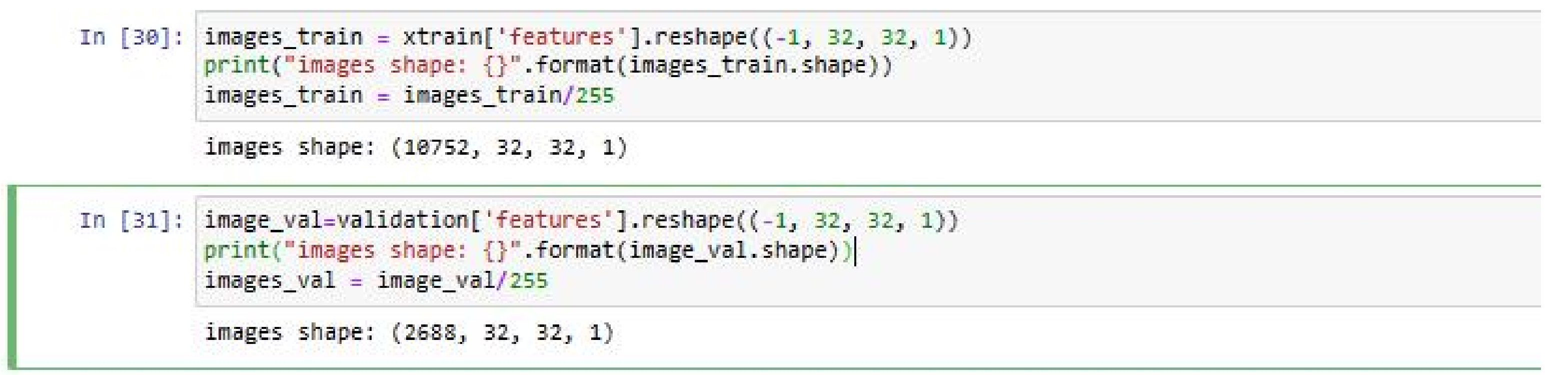
We visulaise the classes sam

Then we divided the X_train and X_test by 255 to normalize them and by this the computation becomes easier and faster since all the numbers became in range of 0 and 1

Then we reshape the data before entering the model because NN train faster on small images A larger input image requires the neural network to learn from four times as many pixels, and this increase the training time for the architecture.



Normalize_and_reshaping



In [30]:
```
images_train = xtrain['features'].reshape((-1, 32, 32, 1))
print("images shape: {}".format(images_train.shape))
images_train = images_train/255
```

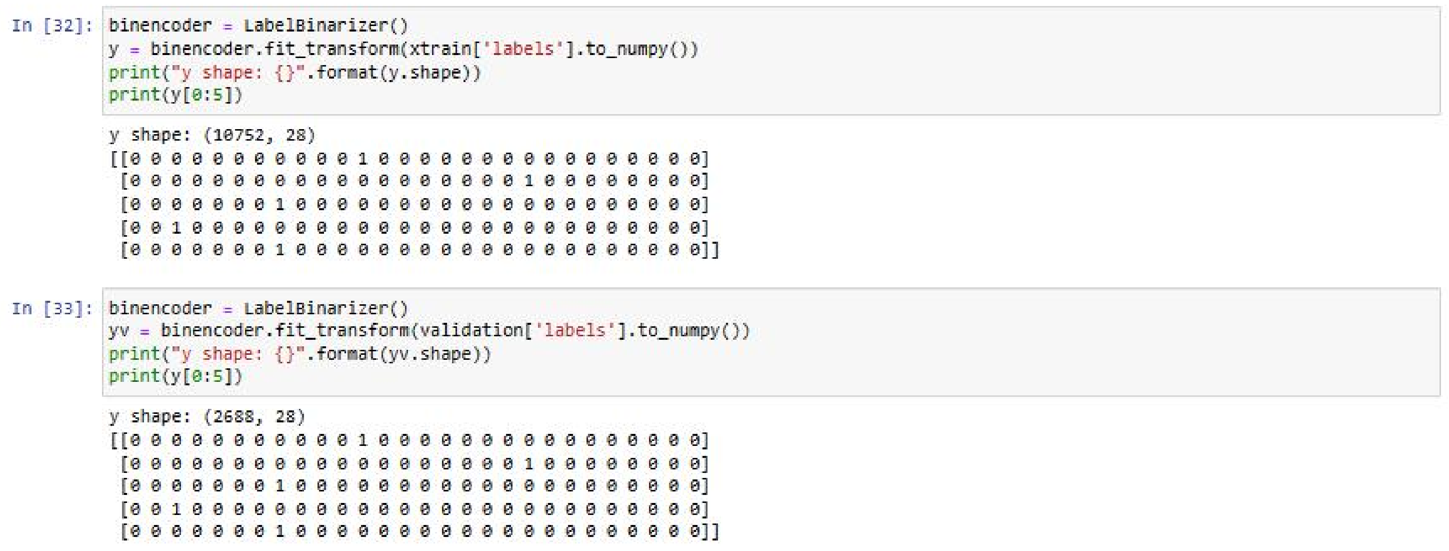
```
images shape: (10752, 32, 32, 1)
```

In [31]:
```
image_val=validation['features'].reshape((-1, 32, 32, 1))
print("images shape: {}".format(image_val.shape))
images_val = image_val/255
```

```
images shape: (2688, 32, 32, 1)
```

Labels are 28 digits numbers from 0 to 27. We need to encode these lables to one hot vectors (ex : 2 -> [0,0,1,0,0,0,0,0,0,0,0,0,0,......].

Encoding_label



In [32]:
```
binencoder = LabelBinarizer()
y = binencoder.fit_transform(xtrain['labels'].to_numpy())
print("y shape: {}".format(y.shape))
print(y[0:5])
```

```
y shape: (10752, 28)
[[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

In [33]:
```
binencoder = LabelBinarizer()
yv = binencoder.fit_transform(validation['labels'].to_numpy())
print("y shape: {}".format(yv.shape))
print(y[0:5])
```

```
y shape: (2688, 28)
[[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

Second Step Building the model:

Sequential() is to create a sequential model because we have many layers that

neurons of each layer are connected to the neurons of the next layer.

We used the Keras Sequential API, where you have just to add one layer at a time, starting from the input.

The first is the convolutional (Conv2D) layer. It is like a set of learnable filters. I choosed to set 16 filters for the firsts conv2D layers and 32 in the second and 64 filters for the third and 128 filters for the last ones,kernal size in all=3, activation is relu in all . Each filter transforms a part of the image (defined by the kernel size) using the kernel filter. The kernel filter matrix is applied on the whole image. Filters can be seen as a transformation of the image.

The CNN can isolate features that are useful everywhere from these transformed images (feature maps).

Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

The second important layer in CNN is the pooling (MaxPool2D) layer. This layer simply acts as a downsampling filter. It looks at the 2 neighboring pixels and picks the maximal value. These are used to reduce computational cost, and to some extent also reduce overfitting. We have to choose the pooling size (i.e the area size pooled each time) more the pooling dimension is high, more the downsampling is important.

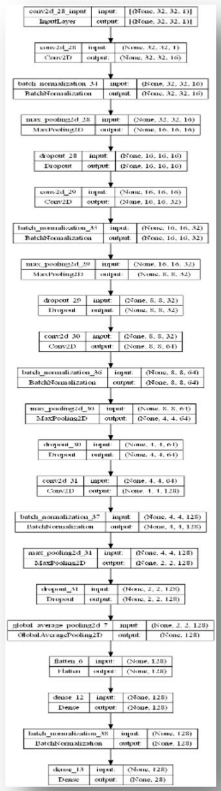
Combining convolutional and pooling layers, CNN are able to combine local features and learn more global features of the image.

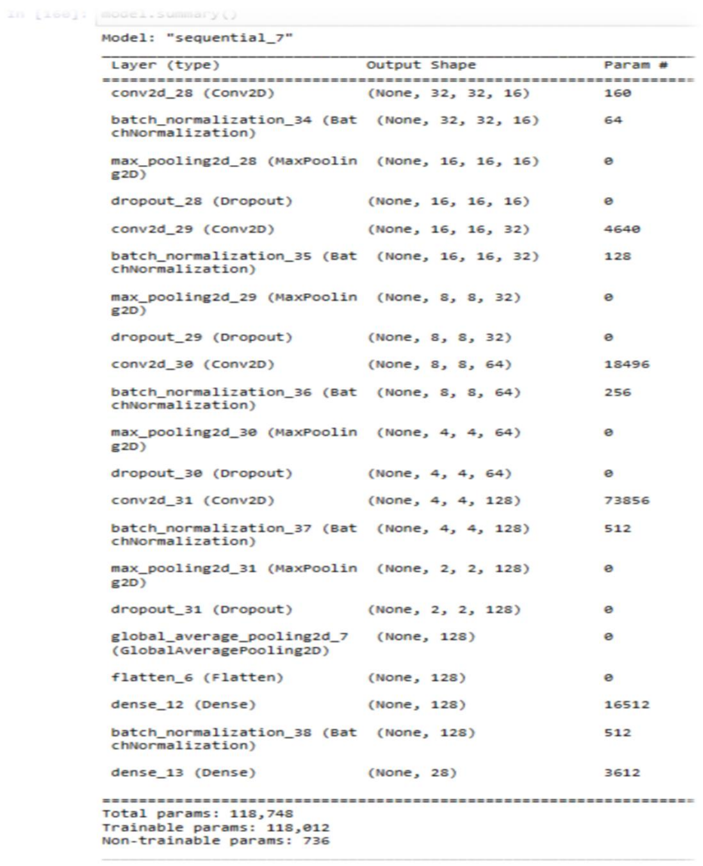
The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by 1/(1 - rate) such that the sum over all inputs is unchanged Dropout is a regularization method, where a proportion of nodes in the layer are randomly ignored (setting their wieghts to zero) for each training sample. This drops randomly a propotion of the network and forces the network to learn features in a distributed way. This technique also improves generalization and reduces the overfitting.

'relu' is the rectifier (activation function max(0,x). The rectifier activation function is used to add non linearity to the network.

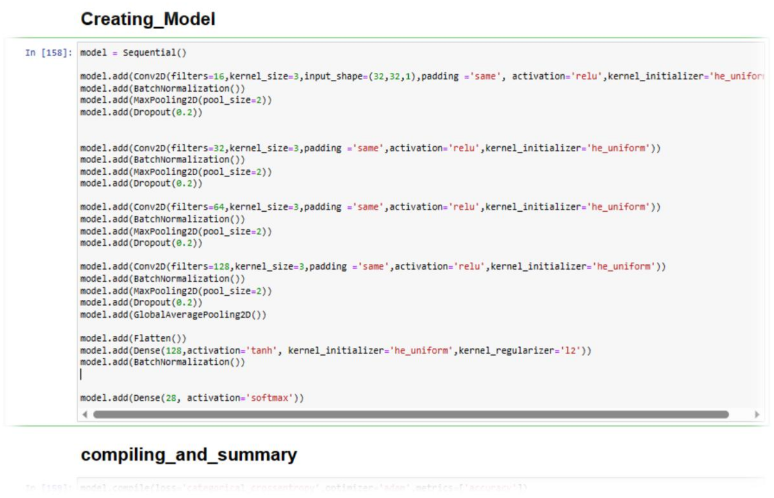
The Flatten layer is use to convert the final feature maps into a one single 1D vector. This flattening step is needed so that you can make use of fully connected layers after some convolutional/maxpool layers. It combines all the found local features of the previous convolutional layers.

In the end i used the features in two fully-connected (Dense) layers which is just artificial an neural networks (ANN) classifier. In the last layer(Dense(28,activation="softmax")) the net outputs distribution of probability of each class.


conv2d_28_input | input: | [(None, 32, 32, 1)]
InputLayer | output: | [(None, 32, 32, 1)]
conv2d_28 | input: | (None, 32, 32, 1)
Conv2D | output: | (None, 32, 32, 16)
batch_normalization_34 | input: | (None, 32, 32, 16)
BatchNormalization | output: | (None, 32, 32, 16)
max_pooling2d_28 | input: | (None, 32, 32, 16)
MaxPooling2D | output: | (None, 16, 16, 16)
dropout_28 | input: | (None, 16, 16, 16)
Dropout | output: | (None, 16, 16, 16)
conv2d_29 | input: | (None, 16, 16, 16)
Conv2D | output: | (None, 16, 16, 32)
batch_normalization_35 | input: | (None, 16, 16, 32)
BatchNormalization | output: | (None, 16, 16, 32)
max_pooling2d_29 | input: | (None, 16, 16, 32)
MaxPooling2D | output: | (None, 8, 8, 32)
dropout_29 | input: | (None, 8, 8, 32)
Dropout | output: | (None, 8, 8, 32)
conv2d_30 | input: | (None, 8, 8, 32)
Conv2D | output: | (None, 8, 8, 64)
batch_normalization_36 | input: | (None, 8, 8, 64)
BatchNormalization | output: | (None, 8, 8, 64)
max_pooling2d_30 | input: | (None, 8, 8, 64)
MaxPooling2D | output: | (None, 4, 4, 64)
dropout_30 | input: | (None, 4, 4, 64)
Dropout | output: | (None, 4, 4, 64)
conv2d_31 | input: | (None, 4, 4, 64)
Conv2D | output: | (None, 4, 4, 128)
batch_normalization_37 | input: | (None, 4, 4, 128)
BatchNormalization | output: | (None, 4, 4, 128)
max_pooling2d_31 | input: | (None, 4, 4, 128)
MaxPooling2D | output: | (None, 2, 2, 128)
dropout_31 | input: | (None, 2, 2, 128)
Dropout | output: | (None, 2, 2, 128)
global_average_pooling2d_7 | input: | (None, 2, 2, 128)
GlobalAveragePooling2D | output: | (None, 128)
flatten_6 | input: | (None, 128)
Flatten | output: | (None, 128)
dense_12 | input: | (None, 128)
Dense | output: | (None, 128)
batch_normalization_38 | input: | (None, 128)
BatchNormalization | output: | (None, 128)
dense_13 | input: | (None, 128)
Dense | output: | (None, 28)



```
In [160]: model.summary()

Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_28 (Conv2D)          (None, 32, 32, 16)        160

 batch_normalization_34 (Bat  (None, 32, 32, 16)       64
 chNormalization)

 max_pooling2d_28 (MaxPoolin  (None, 16, 16, 16)       0
 g2D)

 dropout_28 (Dropout)        (None, 16, 16, 16)        0

 conv2d_29 (Conv2D)          (None, 16, 16, 32)        4640

 batch_normalization_35 (Bat  (None, 16, 16, 32)       128
 chNormalization)

 max_pooling2d_29 (MaxPoolin  (None, 8, 8, 32)         0
 g2D)

 dropout_29 (Dropout)        (None, 8, 8, 32)          0

 conv2d_30 (Conv2D)          (None, 8, 8, 64)          18496

 batch_normalization_36 (Bat  (None, 8, 8, 64)         256
 chNormalization)

 max_pooling2d_30 (MaxPoolin  (None, 4, 4, 64)         0
 g2D)

 dropout_30 (Dropout)        (None, 4, 4, 64)          0

 conv2d_31 (Conv2D)          (None, 4, 4, 128)         73856

 batch_normalization_37 (Bat  (None, 4, 4, 128)        512
 chNormalization)

 max_pooling2d_31 (MaxPoolin  (None, 2, 2, 128)        0
 g2D)

 dropout_31 (Dropout)        (None, 2, 2, 128)         0

 global_average_pooling2d_7   (None, 128)              0
 (GlobalAveragePooling2D)

 flatten_6 (Flatten)         (None, 128)               0

 dense_12 (Dense)            (None, 128)               16512

 batch_normalization_38 (Bat  (None, 128)              512
 chNormalization)

 dense_13 (Dense)            (None, 28)                3612

=================================================================
Total params: 118,748
Trainable params: 118,012
Non-trainable params: 736
_________________________________________________________________
```



Creating_Model

```
In [158]: model = Sequential()

          model.add(Conv2D(filters=16,kernel_size=3,input_shape=(32,32,1),padding ='same', activation='relu',kernel_initializer='he_unifor
          model.add(BatchNormalization())
          model.add(MaxPooling2D(pool_size=2))
          model.add(Dropout(0.2))


          model.add(Conv2D(filters=32,kernel_size=3,padding ='same',activation='relu',kernel_initializer='he_uniform'))
          model.add(BatchNormalization())
          model.add(MaxPooling2D(pool_size=2))
          model.add(Dropout(0.2))

          model.add(Conv2D(filters=64,kernel_size=3,padding ='same',activation='relu',kernel_initializer='he_uniform'))
          model.add(BatchNormalization())
          model.add(MaxPooling2D(pool_size=2))
          model.add(Dropout(0.2))

          model.add(Conv2D(filters=128,kernel_size=3,padding ='same',activation='relu',kernel_initializer='he_uniform'))
          model.add(BatchNormalization())
          model.add(MaxPooling2D(pool_size=2))
          model.add(Dropout(0.2))
          model.add(GlobalAveragePooling2D())

          model.add(Flatten())
          model.add(Dense(128,activation='tanh', kernel_initializer='he_uniform',kernel_regularizer='l2'))
          model.add(BatchNormalization())


          model.add(Dense(28, activation='softmax'))
```

compiling_and_summary

```
In [159]: model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
```

- **Input Layer**: It represent input image data. It will reshape image into single diminsion array. Example your image is 32x32 = 1024, it will convert to (1024,1) array.
- **Conv Layer**: This layer will extract features from image.
- **Pooling Layer**: This layerreduce the spatial volume of input image after convolution.
- **Fully Connected Layer**: It connect the network from a layer to another layer
- **Output Layer**: It is the predicted values layer.

Hyperparameters used:

1. Learning Rate = 0.2

2. Optimizer : Adam TensorFlow

3. Regularization : Overfit the Dataset :

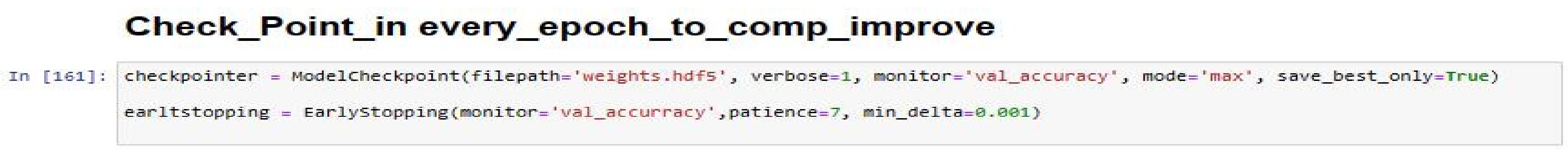
    a. Training Not Big=10702

    b. Fewer Layer=3 layer

    c. Dropout=0.2

    d. earltstopping

.



Check_Point_in every_epoch_to_comp_improve

```
In [161]: checkpointer = ModelCheckpoint(filepath='weights.hdf5', verbose=1, monitor='val_accuracy', mode='max', save_best_only=True)
          earltstopping = EarlyStopping(monitor='val_accurracy',patience=7, min_delta=0.001)
```

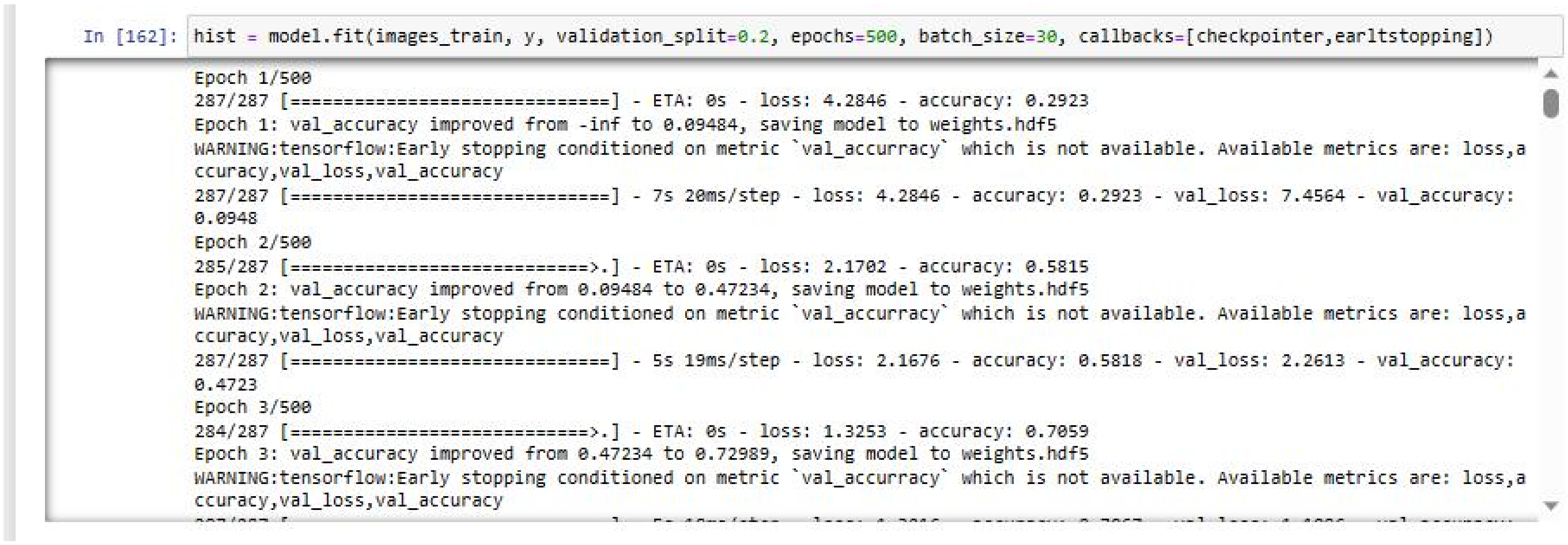
4. Batch Size = 30

5. Number of epochs = 500

method that is based on adaptive estimation of first-order and second-order moments.

The optimizer generates the hyperparameters and reached the best values for the best

accuracy, save them in weights.h5 verbose is 1 to see the



```
In [162]: hist = model.fit(images_train, y, validation_split=0.2, epochs=500, batch_size=30, callbacks=[checkpointer,earltstopping])
```

```
Epoch 1/500
287/287 [==============================] - ETA: 0s - loss: 4.2846 - accuracy: 0.2923
Epoch 1: val_accuracy improved from -inf to 0.09484, saving model to weights.hdf5
WARNING:tensorflow:Early stopping conditioned on metric `val_accurracy` which is not available. Available metrics are: loss,a
ccuracy,val_loss,val_accuracy
287/287 [==============================] - 7s 20ms/step - loss: 4.2846 - accuracy: 0.2923 - val_loss: 7.4564 - val_accuracy:
0.0948
Epoch 2/500
285/287 [============================>.] - ETA: 0s - loss: 2.1702 - accuracy: 0.5815
Epoch 2: val_accuracy improved from 0.09484 to 0.47234, saving model to weights.hdf5
WARNING:tensorflow:Early stopping conditioned on metric `val_accurracy` which is not available. Available metrics are: loss,a
ccuracy,val_loss,val_accuracy
287/287 [==============================] - 5s 19ms/step - loss: 2.1676 - accuracy: 0.5818 - val_loss: 2.2613 - val_accuracy:
0.4723
Epoch 3/500
284/287 [============================>.] - ETA: 0s - loss: 1.3253 - accuracy: 0.7059
Epoch 3: val_accuracy improved from 0.47234 to 0.72989, saving model to weights.hdf5
WARNING:tensorflow:Early stopping conditioned on metric `val_accurracy` which is not available. Available metrics are: loss,a
ccuracy,val_loss,val_accuracy
```

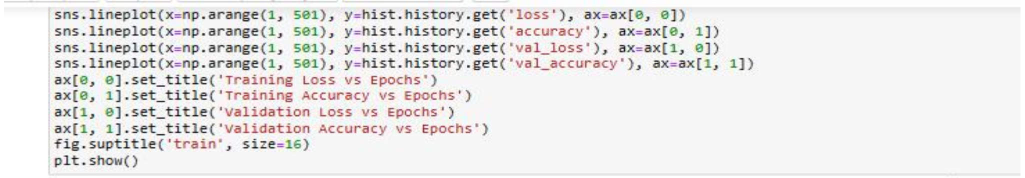
MODEL VALIDATION ACCURACY

	EPOCH	ACCURACY
1	57	improved from 0.95026 to 0.95165
2	102	improved from 0.96048 to 0.96513
3	336	improved from 0.96792 to 0.96978
4	439	improved from 0.96978 to 0.97025

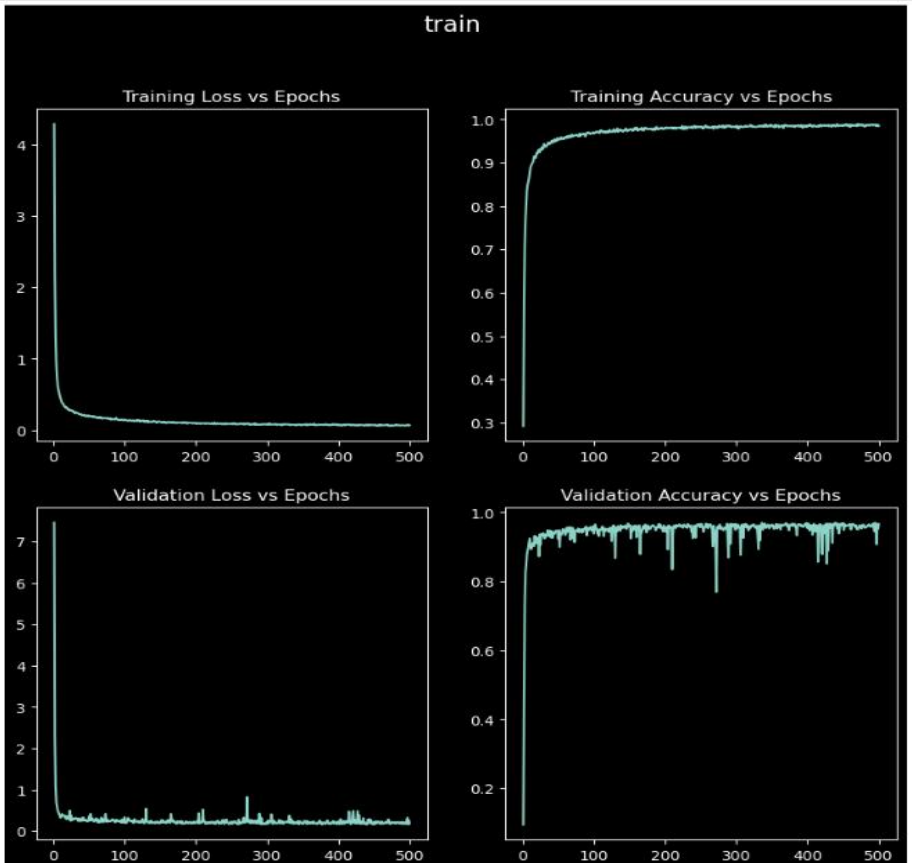
we store all the data about model training in history variable to visualization it using

matplotlib.pyplot library or we can use it to know any information about model training

like loss, val_loss and accuracy



```
sns.lineplot(x=np.arange(1, 501), y=hist.history.get('loss'), ax=ax[0, 0])
sns.lineplot(x=np.arange(1, 501), y=hist.history.get('accuracy'), ax=ax[0, 1])
sns.lineplot(x=np.arange(1, 501), y=hist.history.get('val_loss'), ax=ax[1, 0])
sns.lineplot(x=np.arange(1, 501), y=hist.history.get('val_accuracy'), ax=ax[1, 1])
ax[0, 0].set_title('Training Loss vs Epochs')
ax[0, 1].set_title('Training Accuracy vs Epochs')
ax[1, 0].set_title('Validation Loss vs Epochs')
ax[1, 1].set_title('Validation Accuracy vs Epochs')
fig.suptitle('train', size=16)
plt.show()
```

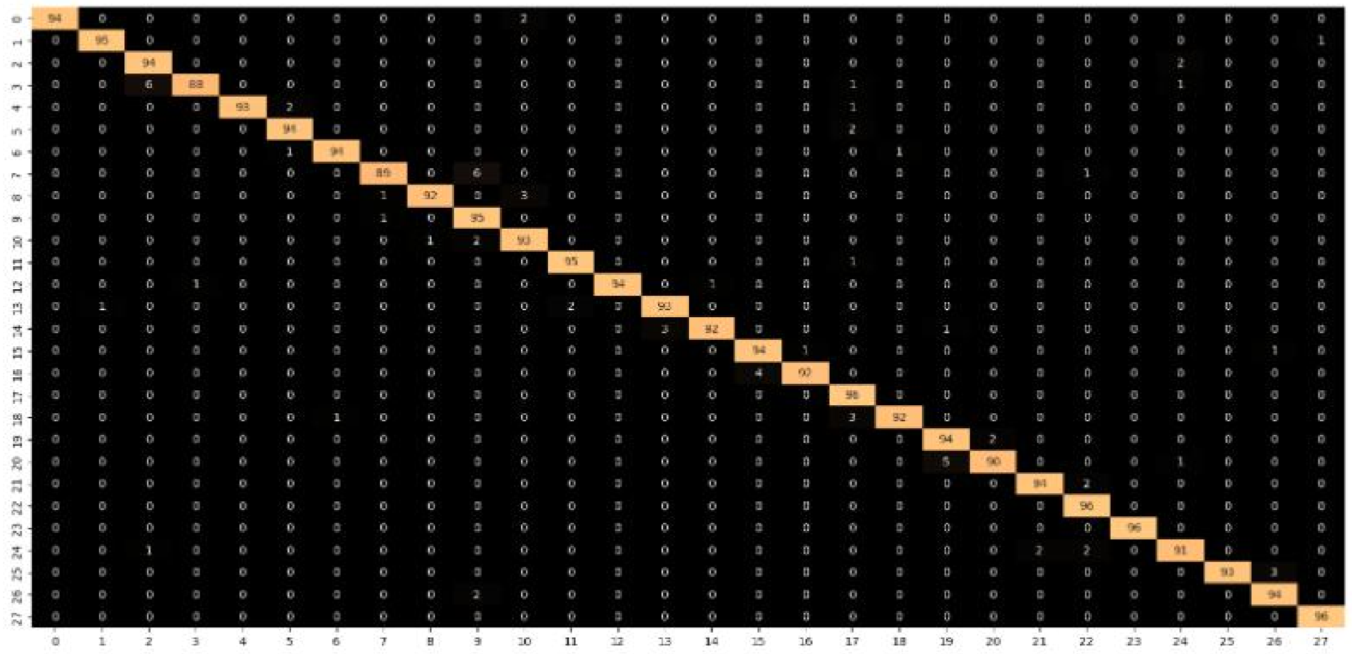


Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class, or vice versa – both variants are found in the literature.[12] The name stems from the fact that it makes it easy to see whether the system is confusing two classes (i.e. commonly mislabeling one as another).

Confusion_Matrix

In [40]:
```
import seaborn as sns
from sklearn.metrics import confusion_matrix
fig,ax=plt.subplots(figsize=(20,10))
cm = confusion_matrix(validation['labels'],pred)
sns.heatmap(cm,annot=True,cmap="copper",fmt="d",cbar=False,ax=ax)
```

Out[40]:
```
<Axes: >
```



True Positives (TPs): 1	False Positives (FPs): 1
False Negatives (FNs): 8	True Negatives (TNs): 90

$$\text{Precision} = \frac{TP}{TP+FP} = \frac{1}{1+1} = 0.5$$

Our model has a precision of 0.5—in other words, when it predicts a tumor is malignant, it is correct 50% of the time.

Recall

Recall attempts to answer the following question:

> What proportion of actual positives was identified correctly?

Mathematically, recall is defined as follows:

$$\text{Recall} = \frac{TP}{TP+FN}$$

Reporting_Accuracy

```
In [26]: y_pred = new_model.predict(images_val)

         pred = np.argmax(y_pred, axis=1) + 1
         ground = np.argmax(yv, axis=1) + 1

         print(classification_report(ground,pred))
```

```
84/84 [==============================] - 0s 4ms/step
              precision    recall  f1-score   support

           1       1.00      0.98      0.99        96
           2       0.99      0.99      0.99        96
           3       0.93      0.98      0.95        96
           4       0.99      0.92      0.95        96
           5       1.00      0.97      0.98        96
           6       0.97      0.98      0.97        96
           7       0.99      0.98      0.98        96
           8       0.98      0.93      0.95        96
           9       0.99      0.96      0.97        96
          10       0.90      0.99      0.95        96
          11       0.95      0.97      0.96        96
          12       0.98      0.99      0.98        96
          13       1.00      0.98      0.99        96
          14       0.97      0.97      0.97        96
          15       0.99      0.96      0.97        96
          16       0.96      0.98      0.97        96
          17       0.99      0.96      0.97        96
          18       0.92      1.00      0.96        96
          19       0.99      0.96      0.97        96
          20       0.94      0.98      0.96        96
          21       0.98      0.94      0.96        96
          22       0.98      0.98      0.98        96
          23       0.95      1.00      0.97        96
          24       1.00      1.00      1.00        96
          25       0.96      0.95      0.95        96
          26       1.00      0.97      0.98        96
          27       0.96      0.98      0.97        96
          28       0.99      1.00      0.99        96

    accuracy                           0.97      2688
   macro avg       0.97      0.97      0.97      2688
weighted avg       0.97      0.97      0.97      2688
```

A ROC curve is constructed by plotting the true positive rate (TPR) against the false positive rate (FPR). The true positive rate is the proportion of observations that were correctly predicted to be positive out of all positive observations (TP/(TP + FN)). Similarly, the false positive rate is the proportion of observations that are incorrectly predicted to be positive out of all negative observations (FP/(TN + FP)). For example, in medical testing, the true positive rate is the rate in which people are correctly identified to test positive for the disease in question.

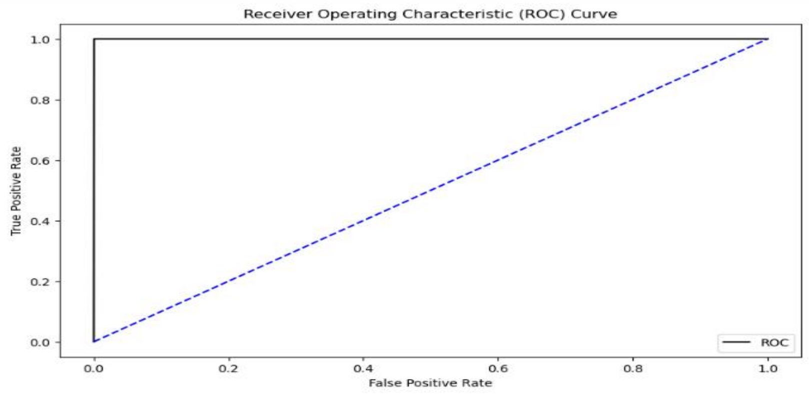
The ROC curve shows the trade-off between sensitivity (or TPR) and specificity (1 – FPR). Classifiers that give curves closer to the top-left corner indicate a better performance. As a baseline, a random classifier is expected to give points lying along the diagonal (FPR = TPR). The closer the curve comes to the 45-degree diagonal of the ROC space, the less accurate the test.

ACC

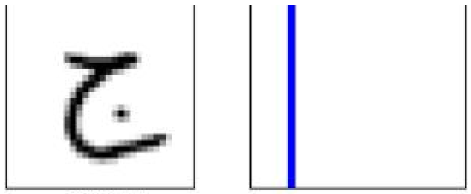
```
In [60]: fpr_keras, tpr_keras, thresholds = roc_curve(ground.ravel(), pred.ravel(),pos_label=28)
         auc_keras = auc(fpr_keras, tpr_keras)
         auc_keras
Out[60]: 0.9998070987654321
```

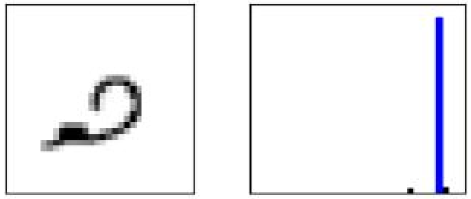
ROC_Curve

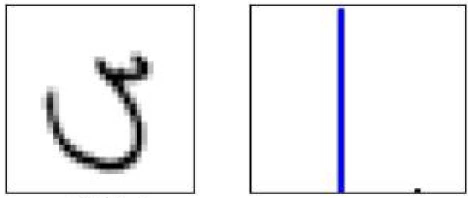
```
In [63]: plot_roc_curve(fpr_keras, tpr_keras)
```

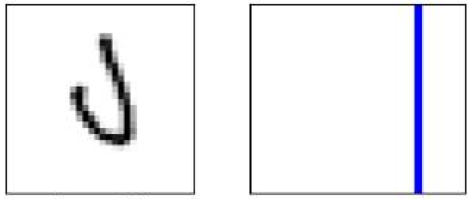


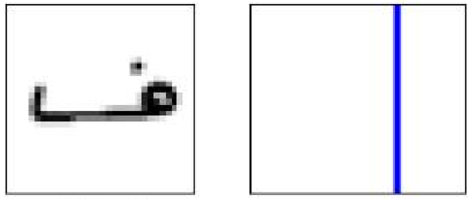
Let's try the model to see some pictures and how to expect them


ج 100% (ج)


ه 93% (ه)


س 98% (س)


ل 100% (ل)


ف 100% (ف)