TRAINING CNN

```
original_folder=r"C:\Users\square\Desktop\TOOBA\UNI\SEM6\ML\Final Project\Gesture Image
Data/9"
train_folder= r"C:\Users\square\Desktop\TOOBA\UNI\SEM6\ML\Final Project\train\Z"
validate_folder=r"C:\Users\square\Desktop\TOOBA\UNI\SEM6\ML\Final Project\validate\9"
test_folder= r"C:\Users\square\Desktop\TOOBA\UNI\SEM6\ML\Final Project\test\Z"


filenames= [f"{i}.jpg" for i in range(1, 1201)]
for filename in filenames:
    src_path = os.path.join(original_folder, filename)
    dst_path = os.path.join(train_folder, filename)
    shutil.copy(src_path, dst_path)
filenames= [f"{i}.jpg" for i in range(1051, 1201)]
for filename in filenames:
    src_path = os.path.join(original_folder, filename)
    dst_path = os.path.join(validate_folder, filename)
    shutil.copy(src_path, dst_path)
filenames= [f"{i}.jpg" for i in range(1201, 1501)]
for filename in filenames:
    src_path = os.path.join(original_folder, filename)
    dst_path = os.path.join(test_folder, filename)
    shutil.copy(src_path, dst_path)
```

Splitting of dataset into test, train and validate is being performed with the help of loops.

```
train_path= r"C:\Users\square\Desktop\TOOBA\UNI\SEM6\ML\Final Project\train"
validate_path= r"C:\Users\square\Desktop\TOOBA\UNI\SEM6\ML\Final Project\validate"
test_path= r"C:\Users\square\Desktop\TOOBA\UNI\SEM6\ML\Final Project\test"
```

```
train_batches =
ImageDataGenerator(preprocessing_function=tf.keras.applications.vgg16.preprocess_input).flo
w_from_directory(directory=train_path,target_size=(64,64), class_mode='categorical',
batch_size=10,shuffle=True)
validate_batches =
ImageDataGenerator(preprocessing_function=tf.keras.applications.vgg16.preprocess_input).flo
w_from_directory(directory=validate_path,target_size=(64,64), class_mode='categorical',
batch_size=10,shuffle=True)
test_batches =
ImageDataGenerator(preprocessing_function=tf.keras.applications.vgg16.preprocess_input).flo
```

```
w_from_directory(directory=test_path,target_size=(64,64), class_mode='categorical',
batch_size=10, shuffle=True)
```

Train path, validate path, test path = three variables created storing all the data

Image data generator(___) : Preprocessing the images using VGG16 model and then creating batches of them
vgg16.preprocess_input will convert the input images from RGB to BGR, then will zero-center each color channel with respect to the ImageNet dataset, without scaling.

flow_from_directory(___) : passing the data here. Directory is set as the variable we stored our data in. Target size: Height, width of images. Setting class, mode, setting batch size, setting shuffle

```
imgs, labels = next(train_batches)

#Plotting the images...
def plotImages(images_arr):
    fig, axes = plt.subplots(1, 10, figsize=(30,20))
    axes = axes.flatten()
    for img, ax in zip( images_arr, axes):
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        ax.imshow(img)
        ax.axis('off')
    plt.tight_layout()
    plt.show()

plotImages(imgs)
print(imgs.shape)
print(labels)
```

line 1: Batch of images and labels are taken from the train dataset, batch size is 10 so this will be 10 images with their labels
plotImages function : from tensorflow, used to simply plot the images. The images shown will be processed using the VGG16 model

```
model = Sequential()

model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=
(64,64,3)))
```

```
model.add(MaxPool2D(pool_size=(2, 2), strides=2))
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding = 'same'))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))
model.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding = 'valid'))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))

model.add(Flatten())

model.add(Dense(64,activation ="relu"))
model.add(Dense(128,activation ="relu"))
model.add(Dense(128,activation ="relu"))
model.add(Dense(36,activation ="softmax"))
```

keras sequential model used (A sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.)
Model.add : adding layers to the sequential model. Layers:

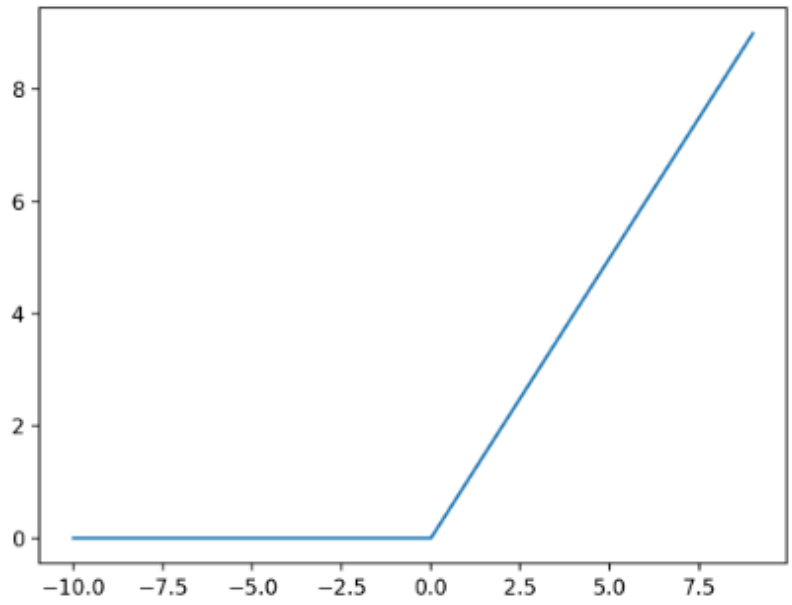
- **Conv2D layer** : standard convolutional layer that accepts image data.

[Convolution means applying a filter or kernel on the input data which results in a feature map (output)
Think of an image as a big matrix and a kernel as tiny matrix
We are sliding the kernel from left-to-right and top-to-bottom along the original image. At each (x, y)-coordinate of the original image, we stop and examine the neighborhood of pixels located at the center of the image kernel. We then take this neighborhood of pixels, convolve them with the kernel, and obtain a single output value. This output value is then stored in the output image]
PARAMETERS:

1. Filters: the number of output filters in the convolution
2. Kernel size: specifying the height and width of the 2D convolution window
3. Activation: activation function you want to apply *after* performing the convolution. An activation function in a neural network defines how the weighted sum of the input is transformed into an output.

    Activation = ReLU:

    ReLU (Rectified Linear Unit) For positive input: ReLU is linear function, output is same as input. For negative input: ReLU is non linear function, Output is zero



4. Padding: two types: 1) "valid" -> means non-zero padding, dimensions are allowed to reduce // 2) "same" -> zero padding, dimensions aren't reduced
5. Input shape: telling the model the shape of our input data. 64x64 is size, 3 is RGB colours

- **Max Pooling Layer**:
  max pooling reduces the spatial dimensionality of images by reducing the number of pixels in the output from the previous convolutional layer, is always added after a convolutional layer

[How it works: take the first (n x n) region/ block on the input image, calculate the max value from each value in the block, store this value in the output channel, then move over by the number of pixels are defined in stride size, repeat. Once we reach the edge over on the far right, we then move down]

PARAMETERS:

1. Pool size: size of pooling window
2. Stride: determines how many units the filter slides across the input image

ADVANTAGES:

1. Reducing Computational Load
2. Reduces Overfitting

- **Flatten Layer**

flattens the multi-dimensional input tensors into a single dimension, so you can model your input layer and build your neural network model, then pass those data into every single neuron of the model effectively.

- **Dense Layer:** for changing the dimension of the output by performing matrix vector multiplication, used to classify image based on output from convolutional layers

[Dense Layer is simple layer of neurons in which each neuron receives input from all the neurons of previous layer.
The neurons in the layer performs the matrix multiplication and results in an output which is then passed through the activation (ReLU) function.]
Operation implemented:
output = activation(dot(input, kernel) + bias)

PARAMETERS:

1. Units: the size (dimensionality) of the output from the dense layer
2. Activation: function that is used for the transformation of the input values

Softmax activation function is used when we have 2 or more than 2 classes
Softmax function use in last output layer
mathematical function that converts a vector of numbers into a vector of probabilities

If we have total 10 classes, then the number of neurons in the output layer will be 10 . Each neuron represents one class.
All 10 neurons will return probabilities of the input image for the respective class. *Class with highest probability will be considered as output for that image*
we need two layers with non linear activation (sigmoid, tanh or RELU) - that is universal aproximator. In many cases we use three layers, two with non linear activation and the third with Softmax to provide probabilities.

```
model.compile(optimizer=SGD(learning_rate=0.001), loss='categorical_crossentropy', metrics=
['accuracy'])
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=1, min_lr=0.0005)
early_stop = EarlyStopping(monitor='val_loss', min_delta=0, patience=2, verbose=0,
mode='auto')
```

- **Model.compile(___) :** Configures the model for training

PARAMETERS:

1. Optimizer: optimizers are used for improving speed and performance for training a specific model
    - SDG optimizer: Optimizer that implements the SGD algorithm
        - SGD algorithm: SGD randomly picks one data point from the whole data set at each iteration to reduce the computations enormously. involves gradient descent methodology.
2. Loss: Loss function is used to find error or deviation in the learning process
   Categorical Corssentropy: used when The output label is assigned one-hot category encoding value in form of 0s and 1. The model uses the categorical crossentropy to learn to give a high probability to the correct digit and a low probability to the other digits.
3. Metrics: Metrics is used to evaluate the performance of classification of your model
   accuracy is the fraction of predictions our model got right

* Gradient Descent is designed to find the local minimum of a differential function, hence to find the best parameters that minimize the model's cost function
*Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the loss gradient

- **ReduceLROnPlateau(___):** Reduce learning rate when a metric has stopped improving. Monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

PARAMETERS:

1. monitor: quantity to be monitored.
2. factor: factor by which the learning rate will be reduced. new_lr = lr * factor.
3. patience: number of epochs with no improvement after which learning rate will be reduced.

- **EarlyStopping(___):** Stop training when a monitored metric has stopped improving. Assuming the goal of a training is to minimize the loss

PARAMETERS:

1. monitor: Quantity to be monitored.
2. min_delta: Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
3. patience: Number of epochs with no improvement after which training will be stopped.
4. verbose: Verbosity mode, 0 or 1. Mode 0 is silent, and mode 1 displays messages when the callback takes an action.
5. mode: One of {"auto", "min", "max"}. In min mode, training will stop when the quantity monitored has stopped decreasing; in "max" mode it will stop when the quantity monitored has stopped increasing; in "auto" mode, the direction is automatically inferred from the name of the monitored quantity.

```
history2 = model.fit(train_batches, epochs=10, callbacks=[reduce_lr, early_stop],
validation_data = test_batches)
history2.history
```

- **Model.fit(___):** Trains the model for a fixed number of epochs (iterations on a dataset).

PARAMETERS:
Train batches: variable storing our data
Epochs: iteration. Number of epochs to train a model
Callback: to be applied during training, passed to keras methods in order to hook into the various stages of the model training and inference lifecycle.
Validation data: Data on which to evaluate the loss and any model metrics at the end of each epoch

```
# For getting next batch of validating imgs...
imgs_v, labels_v = next(validate_batches)
scores_v = model.evaluate(imgs_v , labels_v , verbose=0)
```

```
print(f'{model.metrics_names[0]} of {scores_v[0]}; {model.metrics_names[1]} of
{scores_v[1]*100}%')
# Once the model is fitted we save the model using model.save()  function.
model.save('best_model_dataflair2.h5')

word_dict_v =
{0:'0',1:'1',2:'2',3:'3',4:'4',5:'5',6:'6',7:'7',8:'8',9:'9',10:'A',11:'B',12:'C',13:'D',14
:'E',15:'F',16:'G',17:'H',18:'I',19:'J',20:'K',21:'L',22:'M',23:'N',24:'O',25:'P',26:'Q',27
:'R',28:'S',29:'T',30:'U',31:'V',32:'W',33:'X',34:'Y',35:'Z'}

predictions_v = model.predict(imgs_v, verbose=0)
print("predictions on a small set of validation data--")
print("")
for ind, i in enumerate(predictions_v):
    print(word_dict_v[np.argmax(i)], end='   ')
plotImages(imgs)
print('Actual labels')
for i in labels:
    print(word_dict_v[np.argmax(i)], end='   ')
```

Getting the next batch of images from the validate data set & evaluating the model on the validate set to obtain a better result on test data set and printing the accuracy and loss scores
data from dataset is taken, picture store in variable imgs, the labels stored in variable labels

```
# For getting next batch of testing imgs...
imgs, labels = next(test_batches)
scores = model.evaluate(imgs, labels, verbose=0)
print(f'{model.metrics_names[0]} of {scores[0]}; {model.metrics_names[1]} of
{scores[1]*100}%')
#Once the model is fitted we save the model using model.save()  function.
model.save('best_model_dataflair3.h5')
```

Getting the next batch of images from the test data & evaluating the model on the test set and printing the accuracy and loss scores
data from dataset is taken, picture store in variable imgs, the labels stored in variable labels

- **Model.evaluate (___):** to check whether the model is best fit for the given problem and corresponding data,

PARAMAETERS

1. imgs: the input data

2. labels: the target data
3. Verbose: 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line (we are using 0)

- The attribute **model.metrics_names** will give you the display labels for the scalar outputs

- **Model.save(___):** Saves the model in a single HDF5 file. models can be re instantiated in the exact same state, without any of the code used for model definition or training.
    HDF5 file format (Hierarchical Data Format version 5): is an open source file format that supports large, complex, heterogeneous data.
        here are two important terms used in HDF5 format.
        Groups – Folder like element within the HDF5 file which can contain subgroups or datasets.
        Dataset – Actual data contained within the HDF5 file. (Numpy arrays etc.)

```
word_dict =
{0:'0',1:'1',2:'2',3:'3',4:'4',5:'5',6:'6',7:'7',8:'8',9:'9',10:'A',11:'B',12:'C',13:'D',14
:'E',15:'F',16:'G',17:'H',18:'I',19:'J',20:'K',21:'L',22:'M',23:'N',24:'O',25:'P',26:'Q',27
:'R',28:'S',29:'T',30:'U',31:'V',32:'W',33:'X',34:'Y',35:'Z'}
predictions = model.predict(imgs, verbose=0)
print("predictions on a small set of test data--")
print("")
for ind, i in enumerate(predictions):
    print(word_dict[np.argmax(i)], end='   ')
plotImages(imgs)
print('Actual labels')
for i in labels:
    print(word_dict[np.argmax(i)], end='   ')
```

Here we are visualizing and making a small test on the model to check if everything is working as we expect it to while detecting on the live cam feed.

- **word_dict :** is the dictionary containing label names for the various labels predicted. That is of numbers from 1 to 9 and alphabets from A to Z

- **Model.predict(___):** Generates output predictions for the input samples. Computation is done in batches. This method is designed for batch processing of large numbers of inputs.

PARAMAETERS

1. imgs: the input data
2. Verbose: 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = single line (we are using 0)

- **Enumerate(predictions)**: The enumerate function in Python converts a data collection object into an enumerate object. Enumerate returns an object that contains a counter as a key for each value within an object, making items within the collection easier to access. This function is applied to our predicted results to convert them to numerical key values
- **np.argmax(___):** np.argmax() is a built-in Numpy function that is used to get the indices of the maximum element from an array (single-dimensional array). Argmax is an operation that finds the argument that gives the maximum value from a target function.

We are printing the images and their predicted labels

PREDICTION

```
import numpy as np
import cv2
import keras
from keras.preprocessing.image import ImageDataGenerator
import tensorflow as tf
```

import all the essential packages/ libraries

- Numpy: for working with arrays
- Cv2: to solve computer vision problems
- keras: for developing and evaluating deep learning models
- ImageDataGenerator: to augment your images in real-time
- tensorflow: for fast numerical computing of Deep Learning models

```
word_dict =
{0:'0',1:'1',2:'2',3:'3',4:'4',5:'5',6:'6',7:'7',8:'8',9:'9',10:'A',11:'B',12:'C',13:'D',14
```

```
:'E',15:'F',16:'G',17:'H',18:'I',19:'J',20:'K',21:'L',22:'M',23:'N',24:'O',25:'P',26:'Q',27
:'R',28:'S',29:'T',30:'U',31:'V',32:'W',33:'X',34:'Y',35:'Z'}
model = keras.models.load_model(r"C:\Users\square\Desktop\best_model_dataflair3.h5")
background = None
accumulated_weight = 0.5
ROI_top = 100
ROI_bottom = 300
ROI_right = 150
ROI_left = 350
```

We are loading our previously saved model
Setting background variable and accumulated weight (variables we will be needing)
Setting values of ROI (Region of Interest)

```
def cal_accum_avg(frame, accumulated_weight):
    global background

    if background is None:
        background = frame.copy().astype("float")
        return None
    cv2.accumulateWeighted(frame, background, accumulated_weight)
```

- **cal_accum_avg:**
  This function is defined for the purpose of background subtraction. Background subtraction is a technique for separating foreground elements from the background and is done by generating a foreground mask. This technique is used for detecting dynamically moving objects from static cameras.
    The running average over the current frame and the previous frames is computed. This gives us the background model and any new object introduced during the sequencing of the video becomes part of the foreground.
    If the background model is None (i.e if it is the first frame), then initialize it with the current frame.

    PARAMETERS:

the current frame and its accumulated weight

- **cv2.accumulateWeighted:** within the above function
  This is predefined and is used to do the calculation

```
def segment_hand(frame, threshold=25):
    global background

    diff = cv2.absdiff(background.astype("uint8"), frame)

_ , thresholded = cv2.threshold(diff, threshold, 255,
cv2.THRESH_BINARY)

     #Fetching contours in the frame (These contours can be of hand
or any other object in foreground)
    image, contours, hierarchy =
    cv2.findContours(thresholded.copy(), cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE)
    # If length of contours list = 0, means we didn't get any
    contours...
    if len(contours) == 0:
        return None
    else:
        # The largest external contour should be the hand
        hand_segment_max_cont = max(contours, key=cv2.contourArea)

        # Returning the hand segment(max contour) and the
  thresholded image of hand...
        return (thresholded, hand_segment_max_cont)
```

- **segment_hand:** This function is used to segment the hand region from the video sequence

PARAMETERS:

1. Frame: current frame
2. Threshold value: threshold indicates "spam"; a value below indicates "not spam."

- **cv2.absdiff():** we find the absolute difference (distance regardless of positive or negative) between the background model and the current frame

- **cv2.threshold():** Thresholding is the binarization of an image. In general, we seek to convert a grayscale image to a binary image, where the pixels are either *0* or *255 (max value)*.

PARAMETERS:

1. DIff: Input image (we are using the resulting image of the above absolute difference function)
2. Threshold: threshold value indicating spam
3. max value: 255, this is the max value assigned to a pixel is value is more than threshold

- **cv2.findContours:** To find contours. Contours can be explained simply as a curve joining all the continuous points (along the boundary), having same color or intensity

PARAMETERS:

1. Input image: the image returned after thresholding
2. contour retrieval mode: RETR_EXTERNAL : retrieves only the extreme outer contours.
3. contour approximation method: CHAIN_APPROX_SIMPLE: algorithm compresses horizontal, vertical, and diagonal segments along the contour and leaves only their end points

```
cam = cv2.VideoCapture(0)
num_frames =0
while True:
    ret, frame = cam.read()
    # flipping the frame to prevent inverted image of captured
    frame...

    frame = cv2.flip(frame, 1)
    frame_copy = frame.copy()
    # ROI from the frame
    roi = frame[ROI_top:ROI_bottom, ROI_right:ROI_left]
    gray_frame = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
    gray_frame = cv2.GaussianBlur(gray_frame, (9, 9), 0)
    if num_frames < 70:

        cal_accum_avg(gray_frame, accumulated_weight)

        cv2.putText(frame_copy, "FETCHING BACKGROUND...PLEASE WAIT",
```

```
  (80, 400), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0,0,255), 2)

    else:
        # segmenting the hand region
        hand = segment_hand(gray_frame)

        # Checking if we are able to detect the hand...
        if hand is not None:

            thresholded, hand_segment = hand
            # Drawing contours around hand segment
            cv2.drawContours(frame_copy, [hand_segment + (ROI_right,
      ROI_top)], -1, (255, 0, 0),1)

            cv2.imshow("Thesholded Hand Image", thresholded)

            thresholded = cv2.resize(thresholded, (64, 64))
            thresholded = cv2.cvtColor(thresholded,
 cv2.COLOR_GRAY2RGB)
            thresholded = np.reshape(thresholded,
(1,thresholded.shape[0],thresholded.shape[1],3))

            pred = model.predict(thresholded)
            cv2.putText(frame_copy, word_dict[np.argmax(pred)],
(170, 45), cv2.FONT_HERSHEY_SIMPLEX, 1, (0,0,255), 2)

    # Draw ROI on frame_copy
    cv2.rectangle(frame_copy, (ROI_left, ROI_top), (ROI_right,
    ROI_bottom), (255,128,0), 3)
    # incrementing the number of frames for tracking
    num_frames += 1
    # Display the frame with segmented hand
    cv2.putText(frame_copy, "DataFlair hand sign recognition_ _ _",
    (10, 20), cv2.FONT_ITALIC, 0.5, (51,255,51), 1)
    cv2.imshow("Sign Detection", frame_copy)
    # Close windows with Esc
    k = cv2.waitKey(1) & 0xFF
    if k == 27:
        break
# Release the camera and destroy all the windows
cam.release()
cv2.destroyAllWindows()
```

- **cv2.VideoCapture:** start reading a Video from the camera frame by frame

- **cam.read():** frame variable stores frames from camera
- **cv2.flip:** frames are flipped to prevent inverted image
- ROI is extracted from the frame
- **cv2.cvtColor:** OI is converted to greyscale
- **cv2.GaussianBlur:** ROI is blurred
- If frames are less than 70, the running average is calculated and message is printed to please wait, else hand is segmented
- When hand is detected, image is thresholded, contours are drawn, image is resized, colours are converted to greyscale, image is reshaped
- The image is now passed for prediction
- Prediction is printed
- ROI is drawn and the frame is displayed with segmented hand